

**z angielskiego przekąły
Zdzisław Płoski**

Abraham Silberschatz
Peter B. Galvin

PODSTAWY SYSTEMÓW OPERACYJNYCH

wydanie czwarte



WYDAWNICTWA NAUKOWO-TECHNICZNE
WARSZAWA



PRZEDMOWA

Systemy operacyjne są zasadniczą częścią systemu komputerowego. Analogicznie, kurs systemów operacyjnych stanowi zasadniczą część edukacji informatycznej. Niniejsza książka ma służyć jako podręcznik do podstawowego wykładu z systemów operacyjnych dla studentów młodszych i starszych lat studiów dyplomowych lub pierwszego roku studiów podyplomowych. Zawiera przejrzystą prezentację wiedzy leżącej u podstawa systemów operacyjnych.

W tej książce nie kładziemy szczególnego nacisku na żaden konkretny system operacyjny lub sprzęt. Omawiamy w niej natomiast fundamentalne koncepcje odnoszące się do wielu różnych systemów. Przedstawiamy dużą liczbę przykładów nawiązujących zwłaszcza do systemu UNIX, jak również do innych, popularnych systemów operacyjnych. Odwołujemy się w szczególności do systemu operacyjnego Solaris 2 firmy Sun Microsystems – odmiany systemu UNIX, który ostatnio został przekształcony w nowoczesny system operacyjny, wyposażony w wątki na poziomie jądra i użytkownika, symetryczne wieloprocessorowanie i planowanie w czasie rzeczywistym. Inne przykłady dotyczą systemów MS-DOS, Windows i Windows NT firmy Microsoft, systemu Linux, IBM OS/2, Apple Macintosh Operating System, systemu VMS firmy DEC oraz systemu TOPS.

Wymagane wiadomości

Przymajemy jako niezbędné założenie, że Czytelnik jest zaznajomiony z ogólną budową komputerów oraz z językiem wysokiego poziomu, takim jak Pascal. Zagadnienia dotyczące sprzętu, które są potrzebne do zrozumienia sy-

stemów operacyjnych, przedstawiamy w rozdz. 2. Przykłady kodu wyrażamy w języku pseudopascalowym, ale zrozumienie omawianych algorytmów nie wymaga znajomości języka Pascal.

Zawartość książki

Książka jest podzielona na siedem głównych części:

- **Przegląd** (rozdz. od 1 do 3). W rozdziałach tych wyjaśniamy, czym są systemy operacyjne, co robią, jak są pomyślane i zbudowane. Opisujemy, jak rozwijała się koncepcja systemu operacyjnego, jakie są jego najbardziej oczywiste cechy, co robi dla użytkownika, a co dla operatora systemu komputerowego. Unikamy rozoważań o tym, jak rzeczy mają się od wewnętrz. Dzięki temu opisane tu problemy są przystępne dla słuchaczy niższych lat studiów oraz dla indywidualnych Czytelników, którzy chcą się dowiedzieć, czym jest system operacyjny, bez wnikania w szczegóły wewnętrznych algorytmów. W rozdziale 2 omawiamy zagadnienia sprzętowe, ważne do zrozumienia systemów operacyjnych. Czytelnicy mający dobre rozeznanie w problematyce sprzętu, a w szczególności w działaniu urządzeń wejścia-wyjścia, organizacji bezpośredniego dostępu do pamięci (DMA) oraz w operacjach dyskowych, mogą zapoznać się z treścią tego rozdziału pobiędnie lub wręcz go pominąć.
- **Zarządzanie procesami** (rozdz. od 4 do 7). Pojęcia procesu i współprzeźności leżą w samym sercu nowoczesnych systemów operacyjnych. Proces stanowi cząstkę pracy w systemie. System jest zbiorem procesów wykonywanych *współbieżnie*, z których część jest procesami systemowymi (wykonującymi rozkazy kodu samego systemu), a reszta to procesy użytkownika (wykonujące kod dostarczony przez użytkownika). W wymienionych rozdziałach omawiamy różnorakie metody planowania procesów, komunikacji międzyprocesowej, synchronizacji procesów oraz obsługi ich zakleszczeń. W tej grupie zagadnień mieści się także przedstawienie wątków.
- **Zarządzanie pamięcią** (rozdz. od 8 do 11). Podczas wykonywania procesu, lub przynajmniej jego części, musi znajdować się w pamięci operacyjnej (głównej). W celu polepszania stopnia wykorzystania procesora, a także szybkości, z jaką odpowiada on swoim użytkownikom, komputer musi przechowywać w pamięci wiele procesów. Istnieje znaczna liczba różnych schematów zarządzania pamięcią operacyjną. Odzwierciedlają one różnorakie podejście do zarządzania pamięcią, przy czym efektywność

poszczególnych algorytmów zależy od konkretnej sytuacji. Pamięć główna jest na ogół za mała, aby pomieścić wszystkie dane i programy; nie można w niej również przechowywać danych nieustannie. Dlatego system komputerowy musi rozporządzać pamięcią pomocniczą, aby składować w niej zawartość pamięci głównej. Większość nowoczesnych systemów komputerowych używa dysków jako podstawowego nośnika magazynowania informacji, umożliwiającego dostęp bezpośredni (zarówno do programów, jak i do danych). Mechanizmy zawarte w systemie plików umożliwiają bezpośrednie magazynowanie i dostęp zarówno w odniesieniu do danych, jak i do programów przebywających na dysku. W tych rozdziałach zajmujemy się klasycznymi algorytmami wewnętrznymi i strukturami zarządzania pamięcią. Umożliwiają one solidne, praktyczne zrozumienie stosowanych algorytmów – ich własności, zalet i wad.

- **Systemy wejścia-wyjścia** (rozdz. od 12 do 14). Urządzenia przyłączane do komputera różnią się pod wieloma względami. W wielu przypadkach są one także najwolniej działającymi elementami komputera. Z powodu dużych różnic w urządzeniach system operacyjny musi udostępniać szeroki wybór funkcji, które umożliwiają aplikacjom wszechstronne sterowanie urządzeniami. W tej części książki omawiamy dogłębnie system wejścia-wyjścia, poświęcając uwagę jego konstrukcji, interfejsom oraz wewnętrznym strukturam i funkcjom. Ponieważ urządzenia są wąskim gardłem wydajności, analizujemy też zagadnienia dotyczące optymalizacji ich działania. Wyjaśniamy również kwestie związane z pamięciami pomocniczymi i trzeciorzędnymi.
- **Systemy rozproszone** (rozdz. od 15 do 18). *System rozproszony* składa się ze zbioru procesorów, które nie używają wspólnej pamięci ani zegara. System taki zarządza najróżniczszymi zasobami, które udostępnia swoim użytkownikom. Korzystanie z zasobów dzielonych przyspiesza obliczenia oraz polepsza dostępność i niezawodność danych. System tego rodzaju oddaje również do dyspozycji użytkownika rozproszony system plików, w którym użytkownicy, serwery i urządzenia magazynowania informacji znajdują się w różnych miejscach systemu rozproszonego. System rozproszony musi zawierać różnorodne mechanizmy do synchronizacji i komunikacji procesów, obsługi zakleszczeń i rozlicznych błędów, nie napotykanych w systemie scentralizowanym.
- **Ochrona i bezpieczeństwo** (rozdz. 19 i 20). Rozmaite procesy w systemie operacyjnym należy chronić przed wzajemnym oddziaływaniem. Z tego powodu istnieją mechanizmy, z których można skorzystać, aby zapewnić, że pliki, segmenty pamięci, procesory i inne zasoby będą uzy-

wane tylko przez te procesy, które uzyskały właściwe upoważnienie od systemu operacyjnego. Przez ochronę rozumie się mechanizm kontroliowania dostępu programów, procesów lub użytkowników do zasobów zdefiniowanych w systemie komputerowym. Mechanizm ten musi dostarczać możliwości do określania wymaganej kontroli oraz środki jej egzekwowania. Bezpieczeństwo ma na celu ochronę informacji przechowywanej w systemie (zarówno danych, jak i kodu) oraz ochronę zasobów fizycznych systemu komputerowego przed nieupoważnionym dostępem, złożliwymi uszkodzeniami lub zmianami, a także przed przypadkowym wprowadzaniem niespójności.

- **Przykłady konkretnych systemów** (rozdz. od 21 do 24). Różnorodne koncepcje opisane w tej książce łączymy w całość, prezentując rzeczywiste systemy operacyjne. Szczegółowo omawiamy trzy takie systemy: wersję systemu UNIX 4.3BSD z Berkeley, system Linux oraz system Microsoft Windows NT. System 4.3BSD z Berkeley oraz system Linux wybraliśmy dla tego, że UNIX był w swoim czasie na tyle mały, że można go było zrozumieć, a jednocześnie nie był „zabawkowym” systemem operacyjnym. Większość jego wewnętrznych algorytmów dobrano ze względu na ich prostotę, a nie szybkość lub wyrafinowanie. Zarówno system 4.3BSD, jak i Linux są łatwo osiągalne na wydziałach informatyki; wielu więc studentów ma do nich dostęp. System Windows NT wybraliśmy z tego powodu, że daje możliwość zapoznania się z nowoczesnym systemem operacyjnym, który w konstrukcji i implementacji zasadniczo różni się od systemów uniksowych. W rozdziale 24 omawiamy pokróćte kilka innych systemów, które miały wpływ na rozwój dziedziny systemów operacyjnych.

O piątym wydaniu książki *

W związku z poprzednimi wydaniami skierowanymi do nas wiele komentarzy i sugestii. Uwagi te, wraz z naszymi obserwacjami, skłoniły nas do opracowania niniejszego, piątego wydania. W szczególności dokonaliśmy zasadniczych zmian w materiale dotyczącym systemów wejścia-wyjścia; dodaliśmy też dwa nowe rozdziały poświęcone nowoczesnym systemom operacyjnym. Przepisaliśmy na nowo treść kilku rozdziałów, które dotyczą urządzeń pamięci (rozdz. 1, 2, 3 i 8), uaktualniając starszy tekst i usuwając te fragmenty, które przestały być interesujące. Dodaliśmy nowe rozdziały o systemach wejścia-wyjścia

* Trzecie wydanie polskie jest przekładem piątego wydania amerykańskiego. — Przyp. tłum.

oraz o pamięci trzeciorzędnej, jak również rozdziały o systemach Linux i Windows NT.

Istotne uaktualnienia i zmiany wprowadziliśmy w następujących rozdziałach:

- **Rozdział 3.** Dodaliśmy nowy punkt dotyczący maszyny wirtualnej języka Java (ang. *Java Virtual Machine*).
- **Rozdział 4.** Dodaliśmy nowy punkt zawierający opis komunikacji międzyprocesowej w systemie Windows NT.
- **Rozdział 12.** Ten nowy rozdział jest poświęcony architekturze wejścia-wyjścia systemu operacyjnego, w tym – strukturze jądra, metodom przesyłania, metodom powiadamiania oraz zagadnieniom wydajności.
- **Rozdział 13.** Jest to stary rozdz. 12*. Dokonaliśmy w nim istotnych uaktualnień materiału.
- **Rozdział 14.** W tym nowym rozdziale przedstawiliśmy systemy pamięci trzeciorzędnych.
- **Rozdział 19.** Jest to stary rozdz. 13.
- **Rozdział 20.** Jest to stary rozdz. 14.
- **Rozdział 21.** Jest to stary rozdz. 19.
- **Rozdział 22.** Jest to nowy rozdział dotyczący systemu Linux.
- **Rozdział 23.** Jest to nowy rozdział dotyczący systemu Windows NT.
- **Rozdział 24.** Rozdział ten odpowiada staremu rozdz. 21. Dodaliśmy w nim punkt zawierający opis wczesnych systemów wsadowych (poprzednio umieszczony w rozdz. 1) oraz punkt z podsumowaniem nowych właściwości systemu operacyjnego Mach.

Omówienie systemu operacyjnego Mach (stary rozdz. 20) – nowoczesnego systemu operacyjnego zachowującego zgodność z systemem 4.3BSD – jest dostępne w sieci Internet. Opis systemu Nachos również jest dostępny bezpośrednio w sieci. Zapoznanie się z nim to dobry sposób na pogłębienie wiedzy na temat współczesnych systemów operacyjnych. Umożliwia studentom wgląd w jego kod źródłowy, analizę działania systemu na niskim poziomie, samodzielne opracowanie istotnych części systemu operacyjnego i zaobserwowanie skutków własnej pracy.

* Ta numeracja dotyczy czwartego wydania amerykańskiego. —Przyp. tłum.

Listy korespondencyjne i uzupełnienia

Dysponujemy obecnie stroną WWW poświęconą tej książce, zawierającą takie informacje, jak zestaw przeroczy uzupełniających książkę, pliki w formacie postscript z rozdziałami o systemach Mach i Nachos oraz najnowszy wykaz dostrzeżonych błędów. Udostępniany także środowisko, w którym użytkownicy mogą komunikować się ze sobą i z nami. Utworzyliśmy listę korespondencyjną użytkowników naszej książki pod adresem: os-book@research.bell-labs.com. Każdy, kto ma ochotę być na nią wpisany, jest proszony o wysłanie komunikatu pod adresem avi@bell-labs.com z podaniem nazwiska i adresu poczty elektronicznej.

Aby uzyskać informacje dotyczące uzupełniających materiałów dydaktycznych i materiałów dostępnych bezpośrednio w sieci, wystarczy zajrzeć pod adres URL <http://www.awl.com/cseng/books/osc5e>. Wykaz uzupełnień i szczegółowe informacje dotyczące kontaktu na terenie USA i poza wysypane są automatycznie w odpowiedzi na list zaadresowany do osc@aw.com. W celu otrzymania dodatków, których upowszechnianie podlega ograniczeniom, należy skontaktować się z lokalnym punktem sprzedaży.

Errata

Doliołyśmy starań, aby w tym nowym wydaniu nie było wszystkich wcześniejszych błędów, lecz – jak to się zdarza w systemach operacyjnych – trochę ukrytych pozostało. Będziemy wdzięczni za powiadomienie nas o jakichkolwiek błędach lub brakach. Z zadowoleniem powitamy też propozycje ulepszeń lub ćwiczeń. Wszelką korespondencję prosimy przesyłać pod adresem: Avi Silberschatz, Director, Information Sciences Research Center, MH 2T-210, Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974 (avi@bell-labs.com).

Podziękowania

Niniejsza książka wywodzi się z poprzednich wydań. Współautorem trzech pierwszych z nich był James Peterson. Do grona innych osób, które pomogły nam w przygotowaniu poprzednich wydań należą: Randy Bentson, David Black, Joseph Boykin, Jeff Erumfield, Gael Buckley, P. C. Capon, John Carpenter, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Thomas Doeppner, Caleb Drake, M. Raşit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Mark Holli-

day, Richard Kieburz, Carol Kroll, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Yoichi Muraoka, Jim M. Ng, Ed Posnak, Boris Putanec, Charles Qualline, John Quartermann, John Stankovic, Adam Staufer, Steven Stepanek, Hal Stern, Louis Stevens, David Umbaugh, Steve Vinoski, John Werth i J. S. Weston.

Przeredagowanie książki jest dziełem Lyn Dupré, a opracowanie techniczne dziełem Cliffa Wilkessa. Sara Strandtman zredagowała nasz tekst w formacie Latex. W opracowaniu rysunków pomogła Marylin Tumamian, Debbie Lafferty, Lynne Doran Cote i Patricia Unubun okazały pomoc w przygotowaniu książki do produkcji.

Bruce Hillyer przejrzał przepisane na nowo rozdziały 2, 12, 13 i 14, pomagając w ich opracowaniu. Rozdział 14 pochodzi z artykułu Hillyera i Silberschatza z roku 1996, a rozdz. 17 z artykułu Levy'ego i Silberschatza [254]. Rozdział 19 opracowano na podstawie artykułu Quartermanna i in. [339]. Rozdział 22 pochodzi z niepublikowanego rękopisu Stephena Tweediego, a rozdz. 23 z niepublikowanego rękopisu Cliffa Martina. Dodatkowo rozdział ten zredukował Bruce Hillyer.

Dziękujemy również osobom, które przejrzały obecne wydanie książki. Są to: Hamid Arabnia, Sudarshan K. Dhall, Steven Stepanek, Pete Thomas, L. David Umbaugh i Tommy Wagner.

Abraham Silberschatz, 1997, Murray Hill, NJ

Peter Baer Galvin, 1997, Norton, MA



SPIS TREŚCI

CZĘŚĆ 1 ■ PRZEGŁĄD

Rozdział 1 Wstęp / 3

- | | |
|--|--------------------------------------|
| 1.1 Co to jest system operacyjny? / 3 | 1.6 Systemy równoległe / 17 |
| 1.2 Proste systemy wsadowe / 6 | 1.7 Systemy rozproszone / 20 |
| 1.3 Wieloprogramowane systemy
wsadowe / 9 | 1.8 Systemy czasu rzeczywistego / 22 |
| 1.4 Systemy z podziałem czasu / 11 | 1.9 Podsumowanie / 23 |
| 1.5 Systemy operacyjne dla komputerów
osobistych / 14 | Ćwiczenia / 25 |
| | Uwagi bibliograficzne / 26 |

Rozdział 2 Struktury systemów komputerowych / 29

- | | |
|--|--------------------------------------|
| 2.1 Działanie systemu komputerowego / 29 | 2.6 Ogólna architektura systemu / 54 |
| 2.2 Struktura wejścia-wyjścia / 33 | 2.7 Podsumowanie / 56 |
| 2.3 Struktura pamięci / 38 | Ćwiczenia / 57 |
| 2.4 Hierarchia pamięci / 44 | Uwagi bibliograficzne / 59 |
| 2.5 Ochrona sprzętowa / 48 | |

Rozdział 3 Struktury systemów operacyjnych / 61

- | | |
|--------------------------------------|---|
| 3.1 Składowe systemu / 61 | 3.7 Projektowanie i implementacja
systemu / 96 |
| 3.2 Usługi systemu operacyjnego / 68 | 3.8 Generowanie systemu / 99 |
| 3.3 Funkcje systemowe / 71 | 3.9 Podsumowanie / 101 |
| 3.4 Programy systemowe / 82 | Ćwiczenia / 102 |
| 3.5 Struktura systemu / 84 | Uwagi bibliograficzne / 103 |
| 3.6 Maszyny wirtualne / 91 | |

CZĘŚĆ 2 ■ ZARZĄDZANIE PROCESAMI

Rozdział 4 Procesy / 107

- | | |
|----------------------------------|---------------------------------------|
| 4.1 Koncepcja procesu / 107 | 4.6 Komunikacja międzyprocesowa / 129 |
| 4.2 Planowanie procesów / 111 | 4.7 Podsumowanie / 143 |
| 4.3 Działania na procesach / 116 | Ćwiczenia / 144 |
| 4.4 Procesy współpracujące / 120 | Uwagi bibliograficzne / 145 |
| 4.5 Wątki / 122 | |

Rozdział 5 Planowanie przydziału procesora / 147

- | | |
|--|-----------------------------|
| 5.1 Pojęcia podstawowe / 147 | 5.6 Ocena algorytmów / 172 |
| 5.2 Kryteria planowania / 152 | 5.7 Podsumowanie / 177 |
| 5.3 Algorytmy planowania / 154 | Ćwiczenia / 178 |
| 5.4 Planowanie wieloprocesorowe / 167 | Uwagi bibliograficzne / 181 |
| 5.5 Planowanie w czasie rzeczywistym / 169 | |

Rozdział 6 Synchronizowanie procesów / 183

- | | |
|---|--|
| 6.1 Podstawy / 183 | 6.8 Synchronizacja w systemie
Solaris 2 / 220 |
| 6.2 Problem sekcji krytycznej / 186 | 6.9 Transakcje niepodzielne / 221 |
| 6.3 Sprzętowe środki synchronizacji / 193 | 6.10 Podsumowanie / 233 |
| 6.4 Semafory / 196 | Ćwiczenia / 234 |
| 6.5 Klasyczne problemy synchronizacji / 202 | Uwagi bibliograficzne / 238 |
| 6.6 Regiony krytyczne / 207 | |
| 6.7 Monitory / 212 | |

Rozdział 7 Zakleszczenia / 241

- | | |
|--|---|
| 7.1 Model systemu / 242 | 7.7 Likwidowanie zakleszczenia / 264 |
| 7.2 Charakterystyka zakleszczenia / 243 | 7.8 Mieszane metody postępowania
z zakleszczeniami / 266 |
| 7.3 Metody postępowania
z zakleszczeniami / 247 | 7.9 Podsumowanie / 268 |
| 7.4 Zapobieganie zakleszczeniom / 249 | Ćwiczenia / 269 |
| 7.5 Unikanie zakleszczeń / 252 | Uwagi bibliograficzne / 272 |
| 7.6 Wykrywanie zakleszczenia / 260 | |

CZĘŚĆ 3 ■ ZARZĄDZANIE ZASOBAMI PAMIĘCI

Rozdział 8 Zarządzanie pamięcią / 277

- | | |
|--|---|
| 8.1 Podstawy / 277 | 8.6 Segmentacja / 315 |
| 8.2 Logiczna i fizyczna przestrzeń
adresowa / 284 | 8.7 Segmentacji ze stronicowaniem / 323 |
| 8.3 Wymiana / 286 | 8.8 Podsumowanie / 328 |
| 8.4 Przydział ciągły / 290 | Ćwiczenia / 330 |
| 8.5 Stronicowanie / 299 | Uwagi bibliograficzne / 333 |

Rozdział 9 Pamięć wirtualna / 335

9.1 Podstawy / 335	9.7 Szamotanie / 367
9.2 Stronicowanie na żądanie / 338	9.8 Inne rozważania / 374
9.3 Sprawność stronicowania na żądanie / 344	9.9 Segmentacja na żądanie / 382
9.4 Zastępowanie stron / 347	9.10 Podsumowanie / 383
9.5 Algorytmy zastępowania stron / 351	Ćwiczenia / 385
9.6 Przydział ramek / 363	Uwagi bibliograficzne / 390

Rozdział 10 Interfejs systemu plików / 393

10.1 Pojęcie pliku / 393	10.5 Semantyka spójności / 427
10.2 Metody dostępu / 404	10.6 Podsumowanie / 428
10.3 Struktura katalogowa / 408	Ćwiczenia / 430
10.4 Ochrona / 422	Uwagi bibliograficzne / 431

Rozdział 11 Implementacja systemu plików / 433

11.1 Budowa systemu plików / 433	11.6 Rekonstrukcja / 458
11.2 Metody przydziału miejsca na dysku / 438	11.7 Podsumowanie / 460
11.3 Zarządzanie wolną przestrzenią / 449	Ćwiczenia / 461
11.4 Implementacja katalogu / 452	Uwagi bibliograficzne / 463
11.5 Efektywność i wydajność / 454	

CZĘŚĆ 4 ■ SYSTEMY WEJŚCIA-WYJŚCIA**Rozdział 12 Systemy wejścia-wyjścia / 467**

12.1 Przegląd / 467	12.6 Wydajność / 500
12.2 Sprzęt wejścia-wyjścia / 468	12.7 Podsumowanie / 504
12.3 Użytkowy interfejs wejścia-wyjścia / 481	Ćwiczenia / 505
12.4 Podsystem wejścia-wyjścia w jądrze / 489	Uwagi bibliograficzne / 507
12.5 Przekształcanie zamówień wejścia-wyjścia na operacje sprzętowe / 496	

Rozdział 13 Struktura pamięci pomocniczej / 509

13.1 Struktura dysku / 509	13.6 Implementowanie pamięci trwałej / 527
13.2 Planowanie dostępu do dysku / 510	13.7 Podsumowanie / 529
13.3 Zarządzanie dyskiem / 518	Ćwiczenia / 530
13.4 Zarządzanie obszarem wymiany / 522	Uwagi bibliograficzne / 535
13.5 Niezawodność dysku / 526	

Rozdział 14 Struktura pamięci trzeciorzędnej / 537

14.1 Urządzenia pamięci trzeciorzędnej / 537	14.4 Podsumowanie / 551
14.2 Zadania systemu operacyjnego / 541	Ćwiczenie / 551
14.3 Zagadnienia dotyczące wydajności / 545	Uwagi bibliograficzne / 555

CZEŚĆ 5 ■ SYSTEMY ROZPROSZONE

Rozdział 15 Struktury sieci / 559

15.1 Podstawy / 559	15.6 Strategie projektowe / 581
15.2 Motyw / 561	15.7 Przykład działania sieci / 584
15.3 Topologia / 563	15.8 Podsumowanie / 587
15.4 Typy sieci / 569	Ćwiczenia / 587
15.5 Komunikacja / 573	Uwagi bibliograficzne / 589

Rozdział 16 Struktury systemów rozproszonych / 591

16.1 Sieciowe systemy operacyjne / 591	16.5 Zagadnienia projektowe / 605
16.2 Rozproszone systemy operacyjne / 594	16.6 Podsumowanie / 609
16.3 Usługi zdalne / 596	Ćwiczenia / 609
16.4 Odporność / 603	Uwagi bibliograficzne / 610

Rozdział 17 Rozproszone systemy plików / 613

17.1 Podstawy / 613	17.6 Przykłady systemów / 630
17.2 Nazewnictwo i przezroczystość / 615	17.7 Podsumowanie / 664
17.3 Zdalny dostęp do plików / 620	Ćwiczenia / 665
17.4 Obsługa doglądana i niedoglądana / 627	Uwagi bibliograficzne / 666
17.5 Zwielokrotnianie pliku / 629	

Rozdział 18 Koordynacja rozproszona / 667

18.1 Porządkowanie zdarzeń / 667	18.6 Algorytmy elekcji / 694
18.2 Wzajemne wykluczanie / 670	18.7 Osiąganie porozumienia / 697
18.3 Niepodzielność / 674	18.8 Podsumowanie / 700
18.4 Sterowanie współbieżnością / 679	Ćwiczenia / 701
18.5 Postępowanie z zakleszczeniami / 685	Uwagi bibliograficzne / 703

CZEŚĆ 6 ■ OCHRONA I BEZPIECZEŃSTWO

Rozdział 19 Ochrona / 707

19.1 Cele ochrony / 707	19.3 Macierz dostępów / 715
19.2 Domeny ochrony / 709	19.4 Implementacja macierzy dostępów / 720

19.5 Cofanie praw dostępu / 724	19.8 Podsumowanie / 734
19.6 Systemy działające na zasadzie uprawnień / 726	Ćwiczenia / 734
19.7 Ochrona na poziomie języka programowania / 729	Uwagi bibliograficzne / 736

Rozdział 20 Bezpieczeństwo / 737

20.1 Zagadnienie bezpieczeństwa / 737	20.8 Klasyfikacja poziomów bezpieczeństwa komputerowego / 757
20.2 Uwierzytelnianie / 739	20.9 Przykład modelu bezpieczeństwa – system Windows NT / 759
20.3 Hasła jednorazowe / 743	20.10 Podsumowanie / 762
20.4 Zagrożenia programowe / 744	Ćwiczenia / 762
20.5 Zagrożenia systemowe / 746	Uwagi bibliograficzne / 763
20.6 Nadzorowanie zagrożeń / 751	
20.7 Szyfrowanie / 754	

CZĘŚĆ 7 ■ PRZYKŁADY KONKRETNYCH SYSTEMÓW

Rozdział 21 System UNIX / 767

21.1 Historia / 767	21.7 System plików / 803
21.2 Podstawy projektu / 775	21.8 System wejścia-wyjścia / 814
21.3 Interfejs programisty / 777	21.9 Komunikacja międzyprocesowa / 819
21.4 Interfejs użytkownika / 788	21.10 Podsumowanie / 826
21.5 Zarządzanie procesami / 792	Ćwiczenia / 827
21.6 Zarządzanie pamięcią / 798	Uwagi bibliograficzne / 829

Rozdział 22 System Linux / 831

22.1 Historia / 831	22.8 Wejście i wyjście / 872
22.2 Podstawy projektu / 837	22.9 Komunikacja międzyprocesowa / 877
22.3 Moduły jądra / 841	22.10 Struktura sieci / 879
22.4 Zarządzanie procesami / 846	22.11 Bezpieczeństwo / 882
22.5 Planowanie / 851	22.12 Podsumowanie / 886
22.6 Zarządzanie pamięcią / 857	Ćwiczenia / 887
22.7 Systemy plików / 866	Uwagi bibliograficzne / 888

Rozdział 23 System Windows NT / 891

23.1 Historia / 892	23.6 Praca w sieci / 926
23.2 Podstawy projektu / 892	23.7 Interfejs programowy / 934
23.3 Elementy systemu / 894	23.8 Podsumowanie / 943
23.4 Podsystemy środowiskowe / 913	Ćwiczenia / 943
23.5 System plików / 918	Uwagi bibliograficzne / 944

Rozdział 24 Perspektywa historyczna / 945

24.1 Wczesne systemy operacyjne / 945	24.6 System CTSS / 957
24.2 System Atlas / 952	24.7 System MULTICS / 958
24.3 System XDS-940 / 954	24.8 System OS/360 / 959
24.4 System THE / 955	24.9 System Mach / 961
24.5 System RC 4000 / 956	24.10 Inne systemy / 963

Bibliografia / 965

Credits / 985

Skorowidz / 987

Część 1

PRZEGŁĄD

System operacyjny jest programem, który działa jako pośrednik między użytkownikiem komputera a sprzętem komputerowym. Zadaniem systemu operacyjnego jest tworzenie środowiska, w którym użytkownik może wykonywać programy w wygodny i wydajny sposób.

Prześledzimy rozwój systemów operacyjnych od pierwszych ręcznych systemów aż po obecne, wieloprogramowe systemy z podziałem czasu. Zrozumienie przyczyn rozwoju systemów operacyjnych pozwoli nam ocenić, co robi system operacyjny i w jaki sposób.

System operacyjny musi gwarantować bezbłędną pracę systemu komputerowego. Aby można było zapobiegać wpływowi programów użytkownika na działanie systemu, sprzęt komputerowy musi dostarczać systemowi pewnych mechanizmów gwarantujących jego odpowiednie zachowanie. Opisujemy podstawową architekturę komputera, która umożliwia napisanie poprawnego systemu operacyjnego.

System operacyjny oferuje programom i użytkownikom tych programów usługi ułatwiające zadanie programistyczne. Poszczególne usługi będą oczywiście różne w różnych systemach, niemniej jednak istnieją pewne klasy usług, które określmy i przeanalizujemy.

Rozdział 1

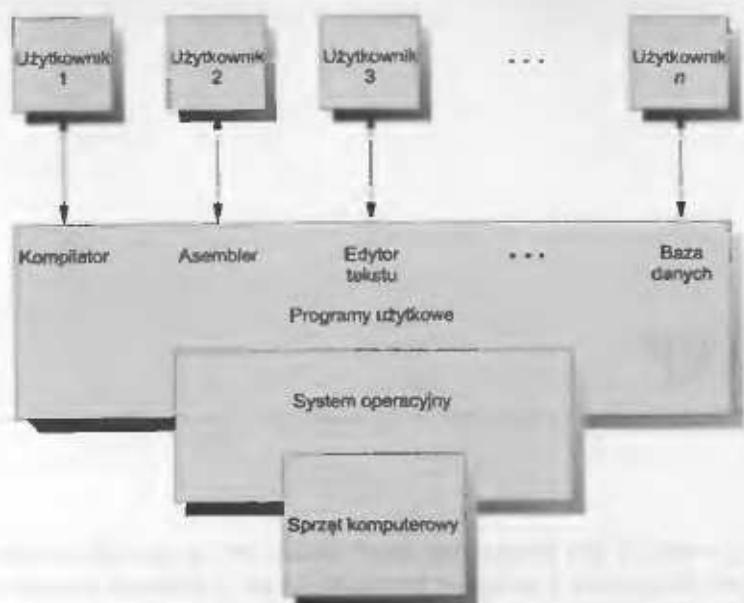
WSTĘP

System operacyjny jest programem, który działa jako pośrednik między użytkownikiem komputera a sprzętem komputerowym. Zadaniem systemu operacyjnego jest tworzenie środowiska, w którym użytkownik może wykonywać programy. Podstawowym celem systemu operacyjnego jest zatem spowodowanie, aby system komputerowy był *wygodny* w użyciu. Drugim celem jest *wydaajna* eksploatacja sprzętu komputerowego.

Aby zrozumieć, czym są systemy operacyjne, musimy najpierw dowiedzieć się, jak one powstawały. W tym rozdziale, a także w rozdz. 24. omawiamy rozwój systemów operacyjnych od pierwszych systemów bezpośrednich aż po obecne, wieloprogramowe systemy z podziałem czasu. Poznając kolejne stadia ich rozwoju, będziemy mogli zobaczyć ewolucję współtworzących je elementów, które powstawały jako naturalne rozwiązania problemów pojawiających się we wczesnych systemach komputerowych. Zrozumienie przyczyn, które wyznaczyły rozwój systemów operacyjnych, da nam pojęcie o tym, jakie zadania wykonuje system operacyjny i w jaki sposób.

1.1 ■ Co to jest system operacyjny?

System operacyjny (ang. *operating system*) jest ważną częścią prawie każdego systemu komputerowego. System komputerowy można z grubsza podzielić na cztery części: sprzęt, system operacyjny, programy użytkowe i użytkowników (rys. 1.1).



Rys. 1.1 Abstrakcyjne wyobrażenie elementów systemu komputerowego

Sprzęt (ang. *hardware*), czyli: procesor – zwany też jednostką centralną* (ang. *central processing unit* – CPU), pamięć i urządzenia wejścia-wyjścia, to podstawowe zasoby systemu komputerowego. Programy użytkowe (aplikacje) – kompilatory, systemy baz danych, gry komputerowe lub programy handlowe – określają sposoby użycia tych zasobów do rozwiązywania zadań stawianych przez użytkowników. Zazwyczaj istnieje wielu różnych użytkowników (ludzie, maszyny, inne komputery) zmagających się z rozwiązywaniem różnych zadań. Odpowiednio do rozmaitych potrzeb może istnieć wiele różnych programów użytkowych. System operacyjny nadzoruje i koordynuje posługiwanie się sprzętem przez różne programy użytkowe, które pracują na zlecenie różnych użytkowników.

System operacyjny jest podobny do rządu. W skład systemu komputerowego wchodzą: sprzęt, oprogramowanie i dane. System operacyjny dostarcza środków do właściwego użycia tych zasobów w działającym systemie komputerowym. Podobnie jak rząd, system operacyjny nie wykonuje sam żadnej użytecznej funkcji. Po prostu tworzy środowisko (ang. *environment*), w którym inne programy mogą wykonywać pożyteczne prace.

Mogemy uważać system operacyjny za dystrybutora zasobów (alokator zasobów; ang. *resource allocator*). System komputerowy ma wiele zasobów

*Obu terminów używamy zamiennie. – Przyp. tłum.

(sprzęt i oprogramowanie), które mogą być potrzebne do rozwiązywania zadania: czas procesora, obszar w pamięci operacyjnej lub w pamięci plików, urządzenia wejścia-wyjścia itd. System operacyjny pełni funkcję zarządcy tych dóbr i przydziela je poszczególnym programom i użytkownikom wówczas, gdy są one nieodzowne do wykonywania zadań. Ponieważ często może dochodzić do konfliktów przy zamawianiu zasobów, system operacyjny musi decydować o przydzielaniu zasobów poszczególnym zamawiającym, mając na względzie wydajne i harmonijne działanie całego systemu komputerowego.

Nieco inne spojrzenie na system operacyjny jest związane z zapotrzebowaniem na sterowanie różnymi urządzeniami wejścia-wyjścia i programami użytkownika. System operacyjny jest *programem sterującym* (ang. *control program*). Program sterujący nadzoruje działanie programów użytkownika, przeciwdziała błędom i zapobiega niewłaściwemu użyciu komputera. Zajmuje się zwłaszcza obsługiwaniem i kontrolowaniem pracy urządzeń wejścia-wyjścia.

Nie ma wszakże w pełni adekwatnej definicji systemu operacyjnego. Istnienie systemów operacyjnych jest uzasadnione tym, że umożliwiają one rozsądne rozwiązywanie problemu kreowania użytecznego systemu obliczeniowego. Podstawowym celem systemów komputerowych jest wykonywanie programów użytkownika i ułatwianie rozwiązywania stawianych przez użytkownika problemów. Do spełnienia tego celu konstruuje się sprzęt komputerowy. Ponieważ posługiwanie się samym sprzętem nie jest szczególnie wygodne, opracowuje się programy użytkowe. Rozmaite programy wymagają pewnych wspólnych operacji, takich jak sterowanie pracą urządzeń wejścia-wyjścia. Wspólne funkcje sterowania i przydzielania zasobów gromadzi się zatem w jednym fragmencie oprogramowania – systemie operacyjnym.

Nie ma również uniwersalnie akceptowanej definicji tego, co jest, a co nie jest częścią systemu operacyjnego. Przymuje się w uproszczeniu, że należy wziąć pod uwagę to wszystko, co dostawca wysyła w odpowiedzi na nasze zamówienie na „system operacyjny”. Jednakże w zależności od rodzaju systemu zapotrzebowanie na pamięć oraz oferowane właściwości bywają bardzo zróżnicowane. Istnieją systemy zajmujące mniej niż 1 MB pamięci (1 MB to milion bajtów*), a jednocześnie nie wyposażone nawet w pełnoekranowy edytor, podczas gdy inne wymagają setek megabajtów pamięci oraz zawierają takie udoskonalenia jak korektory pisowni i całe „systemy okien”. Częściej spotykamy definicję, że system operacyjny jest to ten program, który działa w komputerze nieustannie (nazywany zazwyczaj *jądrem*), podczas gdy wszystkie inne są programami użytkowymi. Ta druga definicja jest popularniejsza od pierwszej i przy niej – ogólnie biorąc – pozostaniemy.

* W punkcie 2.3.2 podano dokładniejszą definicję megabajta. – Przyp. tłum.

Prawdopodobnie łatwiej definiować systemy operacyjne określając, *co robią*, aniżeli czym *są*. Najważniejszym celem systemu operacyjnego jest wygoda użytkownika. Systemy operacyjne istnieją, ponieważ przyjmuję się, że łatwiej z nimi niż bez nich korzystać z komputerów. Widać to szczególnie wyraźnie wówczas, gdy przyjrzymy się systemom operacyjnym małych komputerów osobistych.

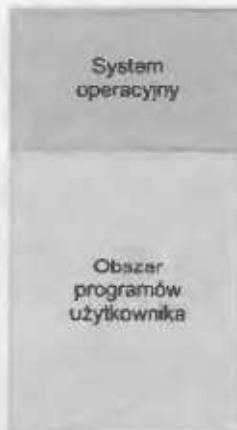
Celem drugorzędnym jest efektywne działanie systemu komputerowego. Ten cel jest szczególnie ważny w rozbudowanych, wielodostępnych systemach z podziałem czasu. Systemy tego rodzaju są zazwyczaj kosztowne, jest więc pożądane, aby były maksymalnie wydajne. Te dwa cele – wygoda i wydajność – są nieraz ze sobą sprzeczne. W przeszłości osiągnięcie wydajności było często przedkładane nad wygodę. Toteż teoria systemów operacyjnych skupia się przede wszystkim na optymalnym wykorzystaniu zasobów komputerowych.

Aby zobaczyć, czym są i co robią systemy operacyjne, prześledźmy ich rozwój na przestrzeni ostatnich 35 lat. Przyglądając się tej ewolucji, będziemy mogli wyodrębnić wspólne elementy systemów operacyjnych i zrozumieć, jak oraz dlaczego systemy te rozwinęły się właśnie tak, a nie inaczej. Bardziej szczegółowe omówienie perspektywy historycznej można znaleźć w rozdz. 24.

Systemy operacyjne i architektura komputerów wywarły na siebie wzajemnie znaczny wpływ. Aby ułatwić posługивание się sprzętem, zaczęto rozwijać systemy operacyjne. Wraz z postępami w projektowaniu i użytkowaniu systemów operacyjnych okazało się, że wprowadzenie zmian w sprzęcie wpłynęłoby na ich uproszczenie. Studując poniższy szkic historyczny, warto zwrócić uwagę, w jaki sposób problemy dotyczące systemów operacyjnych prowadziły do powstawania nowych rozwiązań sprzętowych.

1.2 ■ Proste systemy wsadowe

Pierwsze komputery były wielkimi (fizycznie) maszynami obsługiwanyymi za pośrednictwem konsoli. Popularnymi urządzeniami wejściowymi były czytniki kart i przewijaki taśm. Na wyjściu najczęściej można było spotkać drukarki wierszowe, przewijaki taśm i perforatory kart. Użytkownicy takich systemów nie współpracowali bezpośrednio z systemem komputerowym. Przeciwnie, użytkownik przygotowywał zadanie, które składało się z programu, danych i pewnych, charakteryzujących zadanie informacji sterujących (karty sterujące), po czym przedkładał to wszystko operatorowi komputera. Zadanie znajdowało się zazwyczaj na kartach perforowanych. W późniejszym czasie (po minutach, godzinach lub dniach) pojawiały się informacje wyjściowe zawierające wyniki działania programu, a nickedy obraz jego pamięci – jeśli działanie programu zakończyło się błędem.



Rys. 1.2 Wygląd pamięci operacyjnej prostego systemu wsadowego

Systemy operacyjne tych pierwszych komputerów były zupełnie proste. Podstawowym ich obowiązkiem było automatyczne przekazywanie sterowania od jednego zadania do następnego. System operacyjny rezydował na stałe w pamięci operacyjnej (rys. 1.2).

Aby przyspieszyć przetwarzanie, zadania o podobnych wymaganiach grupowano razem i wykonywano w komputerze w formie tzw. *wsadu* (ang. *batch*). Programiści zostawiali zatem programy operatorowi. Operator sortował je w grupy o podobnych wymaganiach i z chwilą, gdy komputer stawał się dostępny, przekazywał poszczególne pakiety zadań do wykonania. Informacje wyprowadzane przez każde z zadań rozsyiano odpowiednim autorom programów.

Wsadowy system operacyjny czyta zwykle cały strumień odrębnych zadań (np. za pomocą czytnika kart) – każde z odpowiednimi kartami sterującymi, które określają, co ma być w zadaniu zrobione. Po zakończeniu zadania jego wyniki są zazwyczaj drukowane (np. na drukarce wierszowej). Wyróżniającą cechą systemu wsadowego jest *brak* bezpośredniego nadzoru ze strony użytkownika podczas wykonywania zadania. Zadania są przygotowywane i przedkładane. Wyniki pojawiają się po jakimś czasie. Zwłoka między przedłożeniem zadania a jego zakończeniem, czyli *czas obiegu zadania** (ang. *turnaround time*), może wynikać z ilości obliczeń lub być spowodowana opóźnieniem rozpoczęcia zadania przez system.

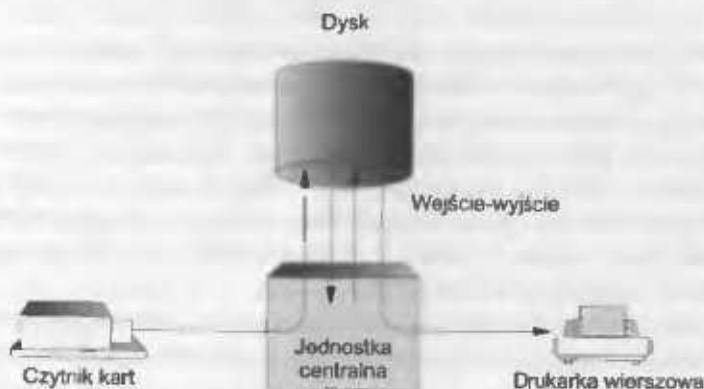
W takim środowisku wykonywania programów jednostka centralna często pozostawała bezczynna. Przyczyna tych przestojów wynikła z szybkości

* Inaczej: czas cyklu przetwarzania. – Przyp. tłum.

działania mechanicznych urządzeń wejścia-wyjścia, które z natury są powolniejsze od urządzeń elektronicznych. Nawet niezbyt szybka jednostka centralna pracowała w tempie mikrosekundowym, wykonując tysiące operacji na sekundę. Natomiast szybki czytnik kart mógł czytać 1200 kart na minutę (17 kart na sekundę). Różnica między szybkościami jednostki centralnej a urządzeń wejścia-wyjścia mogła być zatem większa niż trzy rzędy wielkości. Z czasem, rzecz jasna, na skutek postępu w technice urządzenia zewnętrzne zaczęły działać szybciej. Jednak szybkość procesorów też wzrosła i to nawet bardziej. Problem w związku z tym nie został rozwiązany, lecz nawet uległ pogłębiению.

Pomocne okazało się tutaj wprowadzenie technologii dyskowej. Zamiast czytać karty za pomocą czytnika wprost do pamięci, a następnie przetwarzac zadanie, karty z czytnika kart są czytane bezpośrednio na dysk. Rozmieszczenie obrazów kart jest zapisywane w tablicy przechowywanej przez system operacyjny. Podczas wykonywania zadania zamówienia na dane wejściowe z czytnika kart są realizowane przez czytanie z dysku. Podobnie, gdy zadanie zamówi wyprowadzenie wiersza na drukarkę, wówczas dany wiersz będzie skopiowany do bufora systemowego i zapisany na dysku. Po zakończeniu zadania wyniki są oczywiście drukowane. Tej metodzie przetwarzania nadano nazwę *spooling* (rys. 1.3), którą utworzono jako skrót określenia „*simultaneous peripheral operation on-line*” (jednoczesna, bezpośrednią pracę urządzeń). Spooling w istocie polega na tym, że używa się dysku jako olbrzymiego bufora do czytania z maksymalnym wyprzedzeniem z urządzeń wejściowych oraz do przechowywania plików wyjściowych do czasu, aż urządzenia wyjściowe będą w stanie je przyjąć.

Spooling jest także stosowany przy przetwarzaniu danych w instalacjach zdalnych. Jednostka centralna wysyła dane przez łącza komunikacyjne do



Rys. 1.3 Spooling

zdalnej drukarki (albo przyjmuje cały pakiet zadań ze zdalnego czytnika kart). Przetwarzanie zdalne odbywa się z własną szybkością, bez jakiegokolwiek interwencji ze strony jednostki centralnej. Procesor powinien być jedynie powiadomiony o zakończeniu przetwarzania zdalnego, aby mógł wystać następny pakiet danych.

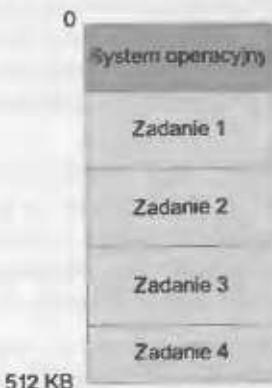
Spooling umożliwia nakładanie w czasie operacji wejścia-wyjścia jednego zadania na obliczenia innych zadań. Nawet w prostym systemie procedura *spooler* może czytać dane jednego zadania i równocześnie drukować wyniki innego. W tym samym czasie może być wykonywane jeszcze inne zadanie (lub zadania), które czyta swoje „karty” z dysku i również na dysku „drukuje” wiersze swoich wyników.

Spooling wywarł bezpośredni, dobry wpływ na zachowanie systemu. Kosztem pewnego obszaru pamięci na dysku i niewielu tablic procesor mógł wykonywać obliczenia jednego zadania równocześnie z operacjami wejścia-wyjścia dla innych zadań. Dzięki spoolingowi stało się możliwe utrzymywanie zarówno procesora, jak i urządzeń wejścia-wyjścia w znacznie większej aktywności.

1.3 ■ Wieloprogramowane systemy wsadowe

Ze spoolingiem jest związana bardzo ważna struktura danych – *pula zadań* (ang. *job pool*). Spooling powoduje, że pewna liczba zadań jest zawsze czytana na dysk, gdzie czeka gotowa do wykonania. Dzięki istnieniu puli zadań na dysku system operacyjny może tak wybierać następne zadania do wykonania, aby zwiększyć wykorzystanie jednostki centralnej. Przy zadańach nadchodzących wprost z kart lub nawet z taśmy magnetycznej nie ma możliwości wykonywania zadań w dowolnym porządku. Zadania muszą być wykonane po kolej w myśl zasad: pierwsze nadeszło – pierwsze zostanie obsłużone. Natomiast gdy kilka zadań znajdzie się na urządzeniu o dostępie bezpośrednim – jak dysk, wówczas staje się możliwe *planowanie zadań* (szerżegowanie zadań; ang. *job scheduling*). Planowanie zadań i przydziału procesora omawiamy szczegółowo w rozdz. 5. Tutaj podamy kilka istotnych aspektów tego zagadnienia.

Najważniejszym aspektem planowania zadań jest możliwość *wieloprogramowania*. Praca pośrednia (ang. *off-line operation*) oraz spooling umożliwiający nakładanie operacji wejścia-wyjścia mają ograniczenia. Jeden użytkownik na ogół nie zdola utrzymać cały czas w aktywności procesora lub urządzeń wejścia-wyjścia. Dzięki wieloprogramowaniu zwiększa się wykorzystanie procesora wskutek takiej organizacji zadań, aby procesor miał zawsze któreś z nich do wykonania.



Rys. 1.4 Wygląd pamięci w systemie wieloprogramowym

Idea jest następująca. W tym samym czasie system operacyjny przechowuje w pamięci kilka zadań (rys. 1.4). Ten zbiór zadań jest podzbiorem zgrupowanych w puli zadań (gdzie liczba zadań, które można jednocześnie przechowywać w pamięci operacyjnej, jest na ogół znacznie mniejsza niż liczba zadań, którą może pomieścić pula). System operacyjny pobiera któreś z zadań do pamięci i rozpoczyna jego wykonywanie. Prędzej czy później zadanie to może zacząć oczekiwania na jakąś usługę, na przykład na zamontowanie taśmy lub na zakończenie operacji wejścia-wyjścia. W systemie jednoprogramowym jednostka centralna musiałaby wówczas przejść w stan bezczynności. W systemie wieloprogramowym można po prostu przejść do wykonywania innego zadania itd. Po jakimś czasie pierwsze zadanie skończy oczekiwanie i otrzyma z powrotem dostęp do procesora. Dopóki są jakieś zadania do wykonania, dopóty jednostka centralna nie jest bezczynna.

Takie postępowanie jest typowe dla zwyczajnych sytuacji zyciowych. Adwokat nie prowadzi na ogół sprawy tylko jednego klienta. Przeciwnie – sprawy kilku klientów toczą się w tym samym czasie. Kiedy jeden pozew czeka na rozprawę lub sporządzenie maszynopisów, wtedy adwokat może pracować nad innym przypadkiem. Przy wystarczającej liczbie klientów adwokat nigdy się nie nudzi. (Bezczynni adwokaci mają skłonności do zostawiania politykami, toteż utrzymywanie adwokatów w ciągłym zatrudnieniu ma pewne społeczne znaczenie).

Wieloprogramowanie jest pierwszym przypadkiem, w którym system operacyjny musi decydować za użytkowników. Wieloprogramowane systemy operacyjne są więc dość skomplikowane. Wszystkie zadania wchodzące do systemu trafiają do puli zadań. Pula ta składa się ze wszystkich procesów pozostających w pamięci masowej i czekających na przydział pamięci opera-

cyjnej. Jeżeli kilka zadań jest gotowych do wprowadzenia do pamięci operacyjnej, lecz brakuje dla wszystkich miejsca, to system musi wybierać spośród nich. Podejmowanie takich decyzji jest *planowaniem zadań* (szeregowaniem zadań); omawiamy je w rozdz. 5. Wybrawszy któryś z zadań z puli, system wprowadza je do pamięci operacyjnej w celu wykonania. Przechowywanie wielu programów w pamięci w tym samym czasie wymaga jakiegoś zarządzania pamięcią – tym zajmujemy się w rozdz. 8 i 9. Ponadto, jeżeli kilka zadań jest gotowych do działania w tym samym czasie, to system musi wybrać któryś z nich. Tego rodzaju decyzje są *planowaniem przydziału procesora* (ang. *CPU scheduling*), które omawiamy w rozdz. 5. Ze współbieżnego wykonywania wielu zadań wynika też potrzeba ograniczania możliwości ich wzajemnego zaburzania we wszystkich stadiach pobytu w systemie operacyjnym: w czasie planowania procesów, przydzielania pamięci dyskowej i zarządzania pamięcią operacyjną. Rozważamy to w wielu miejscach w dalszej części tekstu.

1.4 ■ Systemy z podziałem czasu

Wieloprogramowane systemy wsadowe tworzą środowisko, w którym rozmaite zasoby systemowe (np. jednostka centralna, pamięć operacyjna, urządzenia zewnętrzne) są skutecznie użytkowane. Z punktu widzenia użytkownika system wsadowy jest jednak trochę kłopotliwy. Ponieważ użytkownik nie może ingerować w zadanie podczas jego wykonywania, musi przygotować karty sterujące na okoliczność wszystkich możliwych zdarzeń. W zadaniu wykonywanym krok po kroku następne kroki mogą zależeć od wcześniejszych wyników. Na przykład uruchomienie programu może zależeć od powodzenia fazy komplikacji. Trudno przewidzieć, co należy robić we wszystkich przypadkach.

Inną wadą jest konieczność statycznego testowania programów na podstawie ich migawkowych obrazów pamięci. Programista nie może na bieżąco zmieniać programu w celu zaobserwowania jego zachowań. Długi czas obiegu zadania wyklucza eksperymentowanie z programem. (I na odwrót – sytuacja taka może powodować zwiększenie dyscypliny przy pisaniu i testowaniu programu).

Podział czasu (inaczej *wielozadaniowość*, ang. *multitasking*) stanowi logiczne rozszerzenie wieloprogramowości. Procesor wykonuje na przemian wiele różnych zadań, przy czym przełączenia następują tak często, że użytkownicy mogą współdziałać z każdym programem podczas jego wykonania.

Interakcyjny lub *bezpośredni* (ang. *hands-on*) system komputerowy umożliwia bezpośredni dialog użytkownika z systemem. Użytkownik wydaje bezpo-

średnio instrukcje systemowi operacyjnemu lub programowi i otrzymuje natychmiastowe odpowiedzi. Za wejście służy zazwyczaj klawiatura, a jako wyjściowe – ekran (np. ekran monitora). Po wykonaniu jednego polecenia system szuka następnej „instrukcji sterującej” przekazywanej nie za pośrednictwem czytnika kart, lecz klawiatury użytkownika. Użytkownik wydaje polecenie, czeka na odpowiedź i o kolejnym poleceniu decyduje na podstawie wyników poprzedniego polecenia. Użytkownik może łatwo eksperymentować i natychmiast oglądać rezultaty. Większość systemów ma interakcyjne edytory tekstów do wprowadzania programów i interakcyjne programy uruchomieniowe, ułatwiające usuwanie błędów z programów.

Aby użytkownicy mogli wygodnie korzystać zarówno z danych, jak i z oprogramowania, powinien mieć bezpośredni dostęp do systemu plików (ang. *on-line file system*). Plik (ang. *file*) jest zestawem powiązanych informacji, zdefiniowanym przez jego twórcę. Z grubsza biorąc, w plikach pamięta się programy (zarówno w postaci źródłowej, jak i wynikowej) oraz dane. Pliki danych mogą zawierać liczby, teksty lub mieszane dane alfanumeryczne. Pliki mogą mieć układ swobodny, jak w plikach tekstowych, lub precyzyjnie określony format. Mówiąc ogólnie, plik jest ciągiem bitów, bajtów, wierszy lub rekordów, których znaczenie jest określone przez jego twórcę i użytkownika. System operacyjny urzeczywistnia abstrakcyjną koncepcję pliku, zarządzając takimi urządzeniami pamięci masowych jak taśmy i dyski. Zazwyczaj pliki są zorganizowane w logiczne niepodzielne grupy, czyli katalogi (ang. *directories*), ułatwiające ich odnajdywanie i wykonywanie na nich działań. Ponieważ do plików ma dostęp wielu użytkowników, jest pożądane, by istniał nadzór nad tym, kto i w jaki sposób z nich korzysta.

Systemy wsadowe są odpowiednie dla wielkich zadań, których wykonanie nie wymaga stałego bezpośredniego dozoru. Użytkownik może przedłożyć zadanie i przyjść później po wyniki; nie ma powodu, aby czekał na nie podczas wykonywania zadania. Zadania interakcyjne składają się z wielu krótkich działań, w których rezultaty kolejnych poleceń mogą być nieprzewidywalne. Czas odpowiedzi (ang. *response time*) powinien więc być krótki – co najwyżej rzędu sekund. Systemy interakcyjne mają zastosowanie tam, gdzie oczekuje się szybkich odpowiedzi.

Pierwsze komputery, przeznaczone dla jednego użytkownika, były systemami interakcyjnymi. Rozumiemy przez to, że cały system pozostawał w bezpośredniej dyspozycji programisty-operatora. Zapewniało to programistę wielką elastyczność i swobodę w testowaniu i rozbudowywaniu programu. Ale, jak już to zaobserwowaliśmy, taka organizacja powodowała spore marnotrawstwo czasu wtedy, kiedy procesor czekał na jakieś działania ze strony programisty-operatora. Ze względu na wysoką cenę ówczesnych komputerów przestoje jednostki centralnej były niepożądane. Aby ich uniknąć,

skonstruowano systemy wsadowe. Przyczyniło się to do usprawnienia użytkowania komputera z punktu widzenia właścicieli systemów komputerowych.

Wprowadzenie systemów z podziałem czasu (ang. *time-sharing systems*) umożliwiło interakcyjne użytkowanie systemu komputerowego po umiarkowanych kosztach. W systemie operacyjnym z podziałem czasu zastosowano planowanie przydziału procesora i wieloprogramowość, aby zapewnić każdemu użytkownikowi możliwość korzystania z małej porcji dzielnego czasu pracy komputera. Każdy użytkownik ma przynajmniej jeden oddzielny program w pamięci. Załadowany do pamięci operacyjnej i wykonywany w niej program przyjęto nazywać *procesem* (ang. *process*). Wykonywanie procesu trwa zwykle niedługo i albo się kończy, albo powoduje zapotrzebowanie na operację wejścia-wyjścia. Operacje wejścia-wyjścia mogą przebiegać w trybie konwersacyjnym, tzn. wyniki są wyświetlane użytkownikowi na ekranie, a polecenia i dane – wprowadzane z klawiatury. Ponieważ szybkość interakcyjnego wejścia-wyjścia zależy od człowieka, może ono zajmować sporo czasu. Na przykład wejście może być ograniczone przez tempo pisania na maszynie: pięć znaków na sekundę jest dla ludzi dużą szybkością, ale niezwykle małą dla komputera. Zamiast pozwalać procesorowi na bezczynność, system operacyjny w takich przypadkach angażuje go błyskawicznie do wykonywania programu innego użytkownika.

System operacyjny z podziałem czasu sprawia, że wielu użytkowników dzieli (ang. *share*) równocześnie jeden komputer. Ponieważ pojedyncze działania lub polecenia w systemie z podziałem czasu trwają krótko, każdemu użytkownikowi wystarcza mały przydział czasu jednostki centralnej. Dzięki błyskawicznym przełączeniom systemu od jednego użytkownika do drugiego, każdy z nich odnosi wrażenie, że dysponuje własnym komputerem, choć w rzeczywistości jeden komputer jest dzielony pomiędzy wielu użytkowników.

Ideę podziału czasu zaprezentowano już w 1960 r., lecz ze względu na trudności i koszty budowy systemy z podziałem czasu pojawiły się dopiero we wczesnych latach siedemdziesiątych. W miarę wzrostu popularności podziału czasu konstruktorzy systemów zaczęli łączyć systemy wsadowe z systemami z podziałem czasu. Wiele systemów komputerowych, które pierwotnie zaprojektowano z myślą o użytkowaniu w trybie wsadowym, zostało zmodyfikowanych w celu umożliwienia pracy z podziałem czasu. Na przykład system operacyjny OS/360 dla komputerów IBM, który był systemem wsadowym, poszerzono o możliwość korzystania z podziału czasu (ang. *Time Sharing Option – TSO*). W tym samym okresie systemy z podziałem czasu wzbogacano często o podsystemy wsadowe. Obecnie większość systemów umożliwia zarówno przetwarzanie wsadowe, jak i podział czasu, chociaż w ich podstawowych założeniach i sposobie użycia zwykle przeważa jeden z tych typów pracy.

Systemy operacyjne z podziałem czasu są jeszcze bardziej złożone niż wieloprogramowe systemy operacyjne. Tak jak w wieloprogramowości, w pamięci operacyjnej należy przechowywać jednocześnie wiele zadań, które potrzebują pewnych form zarządzania pamięcią i ochrony (rozdz. 8). Aby zagwarantować akceptowalny czas odpowiedzi, zadania z pamięci operacyjnej trzeba niekiedy odsyłać na dysk i wprowadzać do niej z powrotem. Dysk staje się zapłaszem dla pamięci głównej komputera. Popularną metodą osiągania tego celu jest *pamięć wirtualna* (ang. *virtual memory*), czyli technika umożliwiająca wykonywanie zadania nie mieszczącego się w całości w pamięci operacyjnej (rozdz. 9). Najbardziej widoczną zaletą takiego rozwiązania jest umożliwienie wykonania programów większych niż pamięć fizyczna. Ponadto powstaje tu abstrakcja pamięci głównej w postaci wielkiej, jednolitej tablicy, oddzielająca pamięć logiczną – oglądaną przez użytkownika – od pamięci fizycznej. Uwalnia to programistów od zajmowania się ograniczeniami pamięciowymi. Systemy z podziałem czasu muszą też dostarczać bezpośrednio dostępnego systemu plików (rozdz. 10 i 11). System plików rezyduje w zbiorze dysków, należy więc także zapewnić zarządzanie dyskami (rozdz. 13). Systemy z podziałem czasu muszą też umożliwiać działania współbieżne, a to wymaga przemyślanych metod przydziału procesora (rozdz. 5). Aby zagwarantować porządek wykonywanych działań, w systemie muszą istnieć mechanizmy synchronizowania zadań oraz komunikacji między nimi (rozdz. 6), system musi również zapewnić, że zadania nie będą się zakleszczać*, nieustannie wzajemnie na siebie czekając (rozdz. 7).

Wieloprogramowość i praca z podziałem czasu są podstawowymi zagadnieniami dotyczącymi nowoczesnych systemów operacyjnych i stanowią główny temat tej książki.

1.5 ■ Systemy operacyjne dla komputerów osobistych

Zmniejszanie się cen sprzętu spowodowało po raz wtóry możliwość zbudowania systemu komputerowego przeznaczonego dla indywidualnych użytkowników. Ten rodzaj systemów komputerowych zwykło się nazywać *komputerami osobistymi* (ang. *personal computers* – PC). Oczywiście, zmieniły się urządzenia wejścia-wyjścia: pulpity przełączników i czytniki kart zostały zastąpione klawiaturami, podobnymi do maszyn do pisania, i myszkami. Drukarki wierszowe i perforatory kart ustąpiły miejscu monitorom ekranowym i małym, szybkim drukarkom.

* Z uwagi na aktualną normę terminologiczną obowiązującą w WNT na określenie sytuacji, w których dochodzi do blokady udziałem wiele (co najmniej dwóch) procesów, będzie w tym przykładzie stosowany termin „zakleszczenie”. – Przyp. tłum.

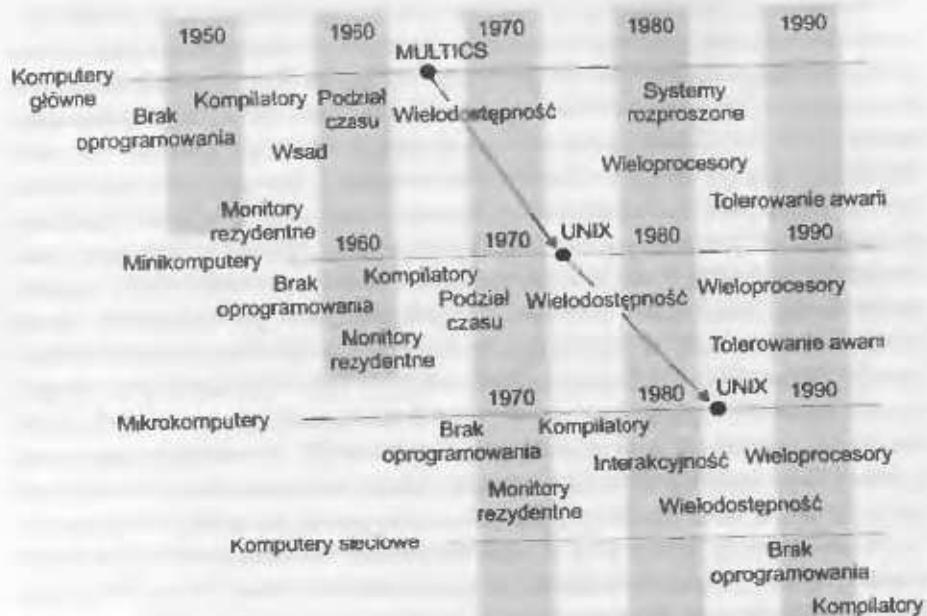
Komputery osobiste pojawiły się w latach siedemdziesiątych. Są to mikrokomputery, zdecydowanie mniejsze i tańsze od systemów komputerów głównych* (ang. *mainframe*). W pierwszym dziesięcioleciu rozwoju komputerów osobistych stosowane w nich jednostki centralne były pozbawione cech potrzebnych do ochrony systemu operacyjnego przed programami użytkowymi. Systemy operacyjne komputerów osobistych nie były więc ani wielostanowiskowe, ani wielozadaniowe. Jednakże cele tych systemów zmieniły się z upływem czasu – zamiast maksymalizowania wykorzystania procesora i urządzeń zewnętrznych położono w nich nacisk na maksimum wygody użytkowania i szybkości kontaktu z użytkownikiem. Do systemów takich zalicza się komputery PC pracujące pod nadzorem systemu Microsoft Windows i systemu Apple Macintosh. Pochodzący z firmy Microsoft system operacyjny MS-DOS został zastąpiony przez liczne atrakcje systemu Microsoft Windows, a firma IBM w miejsce systemu MS-DOS wprowadziła ulepszony system OS/2. System operacyjny Apple Macintosh przeniesiono na lepszy sprzęt, dzięki czemu na on obecnie nowe cechy, takie jak pamięć wirtualna.

W systemach operacyjnych mikrokomputerów skorzystano z różnych wzorów sprawdzonych podczas rozwoju systemów operacyjnych dla dużych komputerów. W mikrokomputerach od samego początku zaadaptowano technikę opracowaną dla większych systemów operacyjnych. Jednak taniość sprzętu mikrokomputerowego sprawia, że może on być użytkowany przez indywidualne osoby, a wykorzystanie procesora przestaje mieć doniosłe znaczenie. W związku z tym pewne rozwiązania uzyskane przy tworzeniu systemów operacyjnych dla dużych komputerów mogą być nieodpowiednie dla systemów mniejszych. Na przykład ochrona plików w komputerach osobistych może okazać się zbędna.

W związku z rozwojem tanich mikroprocesorów i pamięci postawiono tezę, że ich upowszechnienie spowoduje, iż systemy operacyjne (i kursy szkoleniowe na ich temat) staną się zbędne. Trudno uwierzyć w tę przepowiednię. Przeciwnie – obniżenie kosztu sprzętu pozwala realizować względnie wyszukane koncepcje (takie jak podział czasu bądź pamięć wirtualna) w większej niż dotąd liczbie systemów. Dzieje się więc tak, że spadek cen sprzętu komputerowego, na przykład mikroprocesorów, zwiększa nasze zapotrzebowanie na rozumienie zasad działania systemów operacyjnych.

Na przykład ochrona plików nie wydaje się konieczna w oddzielnych komputerach osobistych, ale komputery te są często dopisywane do innych komputerów za pomocą linii telefonicznych lub lokalnych sieci komputer-

* Mówienie o komputerach *mainframe* jako o „ciężkich komputerach” może obecnie odnosić się przede wszystkim do ich fizycznych rozmiarów, dlatego też stosujemy termin *komputer główny*, dziś z kolei często używany w gwarze „serwerem”. Przyp. tłum.



Rys. 1.5 Wędrówka cech i koncepcji systemów operacyjnych

wych. Jeśli zaś inne komputery i inni użytkownicy mogą mieć dostęp do plików w komputerze osobistym, to ochrona plików znów staje się niezbędną cechą systemu operacyjnego. Brak takiej ochrony ułatwił złośliwym programom niszczenie danych w systemach operacyjnych typu MS-DOS lub Macintosh. Programy takie mogą się same powielać i gwałtownie rozprzestrzeniać jako *robaki* lub *wirusy* (ang. *worms*, *viruses*), czyniąc spustoszenie w całych instytucjach lub nawet sieciach komputerowych o zasięgu światowym. Ochronę i bezpieczeństwo rozważamy w rozdz. 19 i 20.

Po przeanalizowaniu systemów operacyjnych dla dużych komputerów i dla mikrokomputerów okazało się, że w istocie te cechy, które były w swoim czasie dostępne tylko w komputerach głównych, zaadaptowano też w mikrokomputerach. Te same koncepcje okazują się odpowiednie dla rozmaitych klas komputerów: komputerów głównych, minikomputerów i mikrokomputerów (rys. 1.5).

Dobrym przykładem przenoszenia koncepcji systemów operacyjnych jest system operacyjny MULTICS. System ten opracowano w latach 1965-1970 w Massachusetts Institute of Technology (MIT) jako *narzędzie obliczeniowe*. Działał na dużym, złożonym komputerze głównym GE 645. Wiele pomysłów wprowadzonych do systemu MULTICS zastosowano następnie w firmie Bell

Laboratories (która była jednym z partnerów opracowujących MULTICS) przy projektowaniu systemu UNIX. System operacyjny UNIX powstał około 1970 r., pierwotnie dla minikomputera PDP-11⁷. Około 1980 r. rozwiązania rodem z systemu UNIX stały się bazą dla uniksopodobnych systemów operacyjnych przeznaczonych dla systemów mikrokomputerowych i pojawiły się w późniejszych systemach operacyjnych, takich jak Microsoft Windows NT, IBM OS/2 i Macintosh Operating System. W ten sposób pomysły zastosowane w wielkich systemach komputerowych przeniknęły z czasem do mikrokomputerów.

W tym samym czasie, gdy ogólnie przydatne cechy wielkich systemów operacyjnych poddawano odpowiednim ograniczeniom, aby lepiej dopasować je do komputerów osobistych, opracowywano silniejsze, szybsze i bardziej wyszukane konstrukcje sprzętowe. Osobista stacja robocza (ang. *workstation*), taka jak Sun, HP/Apollo lub IBM RS/6000, jest wielkim komputerem osobistym. Wiele uniwersytetów i przedsiębiorstw dysponuje znaczną liczbą stacji roboczych powiązanych ze sobą w lokalne sieci komputerowe. W miarę unowocześniania się sprzętu i oprogramowania komputerów PC zanika linia podziału między nimi a stacjami roboczymi.

1.6 ■ Systemy równolegle

Dzisiejsze systemy są w większości jednoprocesorowe, tzn. mają tylko jedną główną jednostkę centralną. Obserwuje się jednakże zainteresowanie systemami wieloprocesorowymi (ang. *multiprocessor systems*). W systemach tego rodzaju pewna liczba procesorów ściśle współpracuje ze sobą, dzieląc szynę komputera, zegar, a czasami pamięć i urządzenia zewnętrzne. Systemy takie nazywa się ściśle powiązanymi (ang. *tightly coupled*).

Istnieje kilka powodów uzasadniających budowanie takich systemów. Jednym z argumentów jest zwiększenie przepustowości (ang. *throughput*). Zwiększając liczbę procesorów, możemy oczekiwać, że większą ilość pracy da się wykonać w krótszym czasie. Jednak współczynnik przyspieszenia przy n procesorach nie wynosi n , lecz jest od n mniejszy. Kiedy kilka procesorów współpracuje przy wykonaniu jednego zadania, wtedy traci się pewną część czasu na utrzymywanie właściwego działania wszystkich części. Ten nakład w połączeniu z rywalizacją o zasoby dzielone powoduje zmniejszenie oczekiwanej zysku z zastosowania dodatkowych procesorów. Analogicznie, grupa współpracujących ze sobą n programistów nie powoduje n -krotnego wzrostu wydajności pracy.

⁷ Według innych źródeł (A. Tanenbaum) był to komputer PDP-7. – Przyp. tłum.

Czesczędności, jakie można uzyskać, stosując wieloprocesory w porównaniu z wykorzystaniem wielu pojedynczych systemów, wynikają także z możliwości wspólnego użytkowania przez nie urządzeń zewnętrznych, zamknięcia ich we wspólnych obudowach i zasilania ze wspólnego źródła. Jeśli kilka programów ma działać na tym samym zbiorze danych, to taniej jest zapamiętać dane na jednym dysku i pozwolić na korzystanie z nich wszystkim procesorom, aniżeli rozmieszczać liczne ich kopie na lokalnych dyskach wielu komputerów.

Innym argumentem na rzecz systemów wieloprocesorowych jest to, że zwiększą one nieczwodność. Umiejętnie rozdzielenie zadań między pewną liczbą procesorów powoduje, że awaria jednego procesora nie zatrzymuje systemu, tylko go spowalnia. Gdy mamy dziesięć procesorów i jeden ulegnie awarii, wówczas pozostałe procesory muszą przejąć i podzielić między siebie pracę uszkodzonego procesora. Dzięki temu cały system będzie pracował tylko o 10% wolniej i nie grozi mu całkowite załamanie. Zdolność kontynuowania usług na poziomie proporcjonalnym do ilości ocalonego sprzętu jest nazywana *łagodną degradacją* (ang. *graceful degradation*). Systemy przystosowane do łagodnej degradacji są również zwane systemami *tolerującymi awarie* (ang. *fault-tolerant*).

Kontynuowanie działania niezależnie od uszkodzeń wymaga mechanizmów ich wykrywania, diagnozowania i (w miarę możliwości) naprawy. Na przykład system Tandem stosuje zarówno sprzętowe, jak i programowe podwojenie funkcji, aby gwarantować nieprzerwane działanie mimo awarii. System składa się z dwóch identycznych procesorów, z których każdy ma lokalną pamięć. Procesory są połączone za pomocą szyny. Jeden z nich jest procesorem podstawowym, drugi jest procesorem zapasowym. Każdy proces ma dwie kopie: jedną w maszynie podstawowej, drugą w maszynie zapasowej. Podczas działania systemu, w ustalonych punktach kontrolnych stan informacji o każdym zadaniu (włącznie z obrazem pamięci) jest kopiowany z maszyny podstawowej do zapasowej. W przypadku wykrycia uszkodzenia uaktywnia się kopię zapasową. Podejmuje ona działanie od ostatniego punktu kontrolnego. Jest to, oczywiście, drogie rozwiązywanie ze względu na podwajanie sprzętu.

Obecnie w systemach wieloprocesorowych używa się najczęściej modelu *wieloprzetwarzania symetrycznego* (ang. *symmetric multiprocessing*), w którym na każdym procesorze działa identyczna kopia systemu operacyjnego. Kopie te komunikują się ze sobą w zależności od potrzeb. W niektórych systemach zastosowano *wieloprzetwarzanie asymetryczne* (ang. *asymmetric multiprocessing*) polegające na tym, że każdy procesor ma przypisane inne zadanie. Systemem takim zawiaduje procesor główny. Inne procesory albo czekają na instrukcje od procesora głównego, albo zajmują się swoimi uprzednio

określonymi zadaniami. Procesor główny planuje i przydziela prace procesorom podporządkowanym.

Przykładem systemu z wieloprzetwarzaniem symetrycznym jest wersja Encore systemu UNIX dla komputera Multimax. Komputer ten jest tak skonfigurowany, że umożliwia działanie wielkiej liczby procesorów, z których każdy pracuje pod nadzorem kopii systemu UNIX. Zaletą tego modelu jest to, iż równocześnie może pracować wiele procesów (N procesów na N egzemplarzach jednostki centralnej) bez pogarszania działania całego systemu. Należy jednak zadbać o takie wykonanie operacji wejścia-wyjścia, aby dane docierały do właściwych procesorów. Z powodu izolacji procesorów może się też zdarzać, że jedne procesory będą pozostawać bezczynne, podczas gdy inne będą przeciążone pracą, co prowadzi do nieefektywności. Aby tego uniknąć, procesory mogą korzystać z pewnych wspólnych struktur danych. Tego rodzaju system wieloprocesorowy pozwala na dynamiczny podział zadań i zasobów między różne procesory i może zmniejszyć różnice między poszczególnymi systemami. Jednakże system tego rodzaju musi być bardzo starannie zaprogramowany; przedstawimy to w rozdz. 6.

Wieloprzetwarzanie asymetryczne częściej występuje w bardzo wielkich systemach, w których czynnościami zużywającymi najwięcej czasu są po prostu operacje wejścia-wyjścia. W starszych systemach wsadowych małe procesory, zlokalizowane w pewnej odległości od głównej jednostki centralnej, służyły do obsługiwanego czytników kart i drukarek oraz do przekazywania zadań do i od komputera głównego. Stacje takie zwykle się nazywały *stanowiskami zdalnego wprowadzania zadań* (ang. *remote job entry* – RJE). W systemach z podziałem czasu operacje wejścia-wyjścia polegają głównie na przekazywaniu znaków między końcówkami konwersacyjnymi a komputerem. Gdyby procesor główny miał być angażowany do obsługi każdego znaku z każdego terminala, mógłby spędzać cały swój czas na przetwarzaniu znaków. W celu uniknięcia tej sytuacji większość systemów ma specjalny procesor czołowy (ang. *front-end*) zajmujący się wszystkimi transmisjami z końcówek. Na przykład duży system komputerowy IBM może używać minikomputera IBM Series/1 jako procesora czołowego. Procesor czołowy działa jak bufor między końcówką konwersacyjną a procesorem głównym, umożliwiając mu obsługę całych wierszy i bloków zamiast pojedynczych znaków. Systemy takie z powodu większej specjalizacji są, niestety, mniej niezawodne.

Ważne jest, aby dostrzegać, że różnica między przetwarzaniem symetrycznym a asymetrycznym może wynikać albo ze sprzętu, albo z oprogramowania. Rozróżnianie wielu procesorów może być wykonywane za pomocą specjalnego sprzętu. Można też napisać oprogramowanie, które będzie pozwalało na istnienie tylko jednego komputera głównego i wielu komputerów podporządkowanych. Na przykład wersja 4 systemu operacyjnego SunOS dla

komputerów Sun umożliwia wieloprzetwarzanie asymetryczne, natomiast wersja 5 tego systemu (Solaris 2) jest symetryczna.

Spadek cen mikroprocesorów i zwiększenie ich możliwości powodują przerzucanie wielu funkcji systemowych na podporządkowany systemowi operacyjnemu sprzęt mikroprocesorowy, czyli jego *zaplecze* (ang. *back-ends*). Na przykład można obecnie z łatwością dodać do systemu mikroprocesor wyposażony we własną pamięć i przeznaczony do zarządzania dyskami. Mikroprocesor taki może przyjmować polecenia od procesora głównego i realizować własną kolejkę dyskową i algorytm planowania. Rozwiążanie to uwalnia procesor główny od zajmowania się planowaniem operacji dyskowych. Komputery PC zawierają wmontowany w klawiaturę mikroprocesor zamieniający uderzenia klawiszy na kody gotowe do przesłania do procesora. Tego rodzaju zastosowania mikroprocesorów rozpowszechniły się do tego stopnia, że nie uważa się ich już za wieloprzetwarzanie.

1.7 ■ Systemy rozproszone

W tworzonych ostatnio systemach komputerowych daje się zauważać tendencja do rozdzielania obliczeń między wiele procesorów. W porównaniu ze ścisłe powiązanymi systemami, omówionymi w p. 1.6, procesory te nie dzielą pamięci ani zegara. Każdy procesor ma natomiast własną pamięć lokalną. Procesory komunikują się za pomocą różnych linii komunikacyjnych, na przykład szybkich szyn danych lub linii telefonicznych. Systemy takie nazywają się zazwyczaj *luźno powiązanymi* (ang. *loosely coupled*) lub *rozproszonymi* (ang. *distributed systems*).

Procesory w systemach rozproszonych mogą się różnić pod względem rozmiaru i przeznaczenia. Mogą wśród nich być małe mikroprocesory, stacje robocze, minikomputery i wielkie systemy komputerowe ogólnego przeznaczenia. Na określenie tych procesorów używa się różnych nazw, takich jak: *stanowiska* (ang. *sites*), *węzły* (ang. *nodes*), *komputery* itp. – zależnie od kontekstu, w którym się o nich mówi.

Systemy rozproszone budują się z wielu powodów; wymienimy tu kilka ważniejszych.

- **Podział zasobów:** Po połączeniu ze sobą różnych stanowisk (o różnych możliwościach) użytkownik jednego stanowiska może korzystać z zasobów dostępnych na innym. Na przykład użytkownik węzła A może korzystać z drukarki laserowej zainstalowanej w węźle B. Użytkownik węzła B może w tym samym czasie mieć dostęp do pliku znajdującego się w A. Mówiąc ogólnie, podział zasobów w systemie rozproszonym tworzy me-

chanizmy dzielnego dostępu do plików w węzłach zdalnych, przetwarzania informacji w rozproszonych bazach danych, drukowania plików w węzłach zdalnych, zdalnego użytkowania specjalizowanych urządzeń sprzętowych (np. odznaczających się wielką szybkością procesorów tablicowych) i wykonywania innych operacji.

- **Przyspieszanie obliczeń:** Jeśli pewne obliczenie da się rozłożyć na zbiór obliczeń cząstkowych, które można wykonywać współbieżnie, to system rozproszony umożliwia przydzielenie tych obliczeń do poszczególnych stanowisk i współbieżne ich wykonanie. Ponadto, jeżeli pewne stanowisko jest w danej chwili przeciążone zadaniami, to część z nich można przenieść do innego, mniej obciążonego stanowiska. Takie przemieszczanie zadań nazywa się *dzieleniem obciążenia* (ang. *load sharing*).
- **Niezawodność:** W przypadku awarii jednego stanowiska w systemie rozproszonym pozostałe mogą kontynuować pracę. Jeżeli system składa się z dużych, autonomicznych instalacji (tzn. komputerów ogólnego przeznaczenia), to awaria jednego z nich nie wpływa na działanie pozostałych. Natomiast gdy system składa się z małych maszyn, z których każda odpowiada za jakąś istotną funkcję (np. za wykonywanie operacji wejścia-wyjścia z kołeczek konwersacyjnych lub za system plików), wówczas z powodu jednego błędu może zostać wstrzymane działanie całego systemu. Ogólnie można powiedzieć, że istnienie w systemie wystarczającego zapasu (zarówno sprzętu, jak i danych) sprawia, że system może pracować nawet po uszkodzeniu pewnej liczby jego węzłów (stanowisk).
- **Komunikacja:** Istnieje wiele sytuacji, w których programy muszą wymieniać dane między sobą w ramach jednego systemu. Przykładem tego są systemy okien, w których często dzieli się dane lub wymienia je między terminalami. Wzajemne połączenie węzłów za pomocą komputerowej sieci komunikacyjnej umożliwia procesom w różnych węzłach wymianę informacji. Użytkownicy sieci mogą przesyłać pliki lub kontaktować się ze sobą za pomocą *poczty elektronicznej* (ang. *electronic mail*). Przesyłki pocztowe mogą być nadawane do użytkowników tego samego węzła lub do użytkowników innych węzłów.

W rozdziale 15 przedstawiamy systemy rozproszone i ogólną strukturę łączących je sieci. W rozdziale 16 omawiamy budowę systemów rozproszonych. W rozdziale 17 rozpatrujemy różne możliwe sposoby projektowania i realizacji rozproszonego systemu plików. Na koniec, w rozdziale 18 zajmujemy się koordynacją rozproszoną.

1.8 ■ Systemy czasu rzeczywistego

Jeszcze innym rodzajem specjalizowanego systemu operacyjnego jest system *czasu rzeczywistego* (ang. *real-time*). System czasu rzeczywistego jest stosowany tam, gdzie istnieją surowe wymagania na czas wykonania operacji lub przepływu danych, dlatego używa się go często jako sterownika w urządzeniu o ścisłe określonym celu. Czujniki dostarczają dane do komputera. Komputer musi analizować te dane i w zależności od sytuacji tak regulować działanie kontrolowanego obiektu, aby zmieniły się wskazania wejściowe czujników. Przykładami systemów czasu rzeczywistego są systemy nadzorowania eksperymentów naukowych, obrazowania badań medycznych, sterowania procesami przemysłowymi i niektóre systemy wizualizacji. Można tu podać również takie przykłady, jak elektroniczny sterownik wtrysku paliwa do silników samochodowych, sterowniki urządzeń gospodarstwa domowego, a także sterowniki stosowane w różnych rodzajach broni. System operacyjny czasu rzeczywistego ma ścisłe zdefiniowane, stałe ograniczenia czasowe. Przetwarzanie danych *musi* się zakończyć przed upływem określonego czasu, w przeciwnym razie system *nie spełnia* wymagań. Na przykład po instruowaniu robota, aby zatrzymał ruch ramienia już *po* zgnieceniu samochodu, który właśnie montował, mija się z celem. Porównajmy ten warunek z systemem podziału czasu, w którym jest pożądane (lecz niekonieczne) szybkie uzyskanie odpowiedzi, lub z systemem wsadowym, w którym może nie być żadnych ograniczeń czasowych.

Istnieją dwie odmiany systemów czasu rzeczywistego. *Rygorystyczny system czasu rzeczywistego* (ang. *hard real-time system*) gwarantuje terminowe wypełnianie krytycznych zadań. Osiągnięcie tego celu wymaga ograniczenia wszystkich opóźnień w systemie, poczynając od odzyskiwania przechowywanych danych, a kończąc na czasie zużywanym przez system na wypełnienie dowolnego zamówienia. Takie ograniczenia czasu wpływają na dobór środków, w które są wyposażane rygorystyczne systemy czasu rzeczywistego. Wszelkiego rodzaju pamięć pomocnicza jest na ogół bardzo mała albo nie występuje wcale. Wszystkie dane są przechowywane w pamięci o krótkim czasie dostępu lub w pamięci, z której można je tylko pobierać (ang. *read-only memory* – ROM). Pamięć ROM jest nieulotna, tzn. zachowuje zawartość również po wyłączeniu dopływu prądu elektrycznego; większość innych rodzajów pamięci jest nietrwała. Systemy te nie mają również większości cech nowoczesnych systemów operacyjnych, które oddalają użytkownika od sprzętu, zwiększając niepewność odnośnie do ilości czasu zużywanego przez operacje. Na przykład prawie nie spotyka się w systemach czasu rzeczywistego pamięci wirtualnej (omawianej w rozdz. 9). Dlatego rygorystyczne systemy czasu rzeczywistego pozostają w konflikcie z działaniem

systemów z podziałem czasu i nie wolno ich ze sobą mieszać. Żaden z istniejących, uniwersalnych systemów operacyjnych nie umożliwia działania w czasie rzeczywistym, dlatego nie będziemy się dalej zajmować systemami tego rodzaju w tej książce.

Mniej wymagającym rodzajem systemu czasu rzeczywistego jest *lagodny system czasu rzeczywistego* (ang. *soft real-time system*), w którym krytyczne zadanie do obsługi w czasie rzeczywistym otrzymuje pierwszeństwo przed innymi zadaniami i zachowuje je aż do swojego zakończenia. Podobnie jak w rygorystycznym systemie czasu rzeczywistego opóźnienia muszą być ograniczone – zadanie czasu rzeczywistego nie może w nieskończoność czekać na usługi jądra. Lagodne traktowanie wymagań dotyczących czasu rzeczywistego umożliwia godzenie ich z systemami innych rodzajów. Jednak użyteczność lagodnych systemów czasu rzeczywistego jest bardziej ograniczona niż systemów rygorystycznych. Ponieważ nie zapewniają one nieprzekraczalnych terminów, zastosowanie ich w przemyśle i robotyce jest ryzykowne. Niemniej jednak istnieje kilka dziedzin, w których są one przydatne: są to m.in. techniki multimedialne, kreowanie sztucznej rzeczywistości oraz zaawansowane projekty badawcze w rodzaju eksploracji podmorskich lub wypraw planetarnych. Takie przedsięwzięcia wymagają systemów operacyjnych o rozbudowanych właściwościach, których nie mogą zapewnić rygorystyczne systemy czasu rzeczywistego. Rosnący krąg zastosowań, o łagodnych wymaganiach na przetwarzanie w czasie rzeczywistym powoduje, że większość współczesnych systemów operacyjnych, łącznie z przeważającą częścią wersji systemu UNIX, czyni zadość tym wymaganiom.

W rozdziale 5 omawiamy aspekty planowania niezbędne do spełnienia w systemie operacyjnym o łagodnych wymaganiach przetwarzania w czasie rzeczywistym. Rozdział 9 zawiera opis projektu zarządzania pamięcią w warunkach czasu rzeczywistego. W zakończeniu, w rozdziale 23 przedstawiamy komponenty przetwarzania w czasie rzeczywistym zrealizowane w systemie Windows NT.

1.9 ■ Podsumowanie

Dwa cele wpłynęły na rozwój systemów operacyjnych, który dokonał się w ciągu minionych 40 lat. Po pierwsze, systemy operacyjne służyły do takiego zaplanowania procesu obliczeniowego, które pozwoliłoby uzyskać efektywne działanie systemu komputerowego. Po drugie, systemy operacyjne tworzą wygodne środowisko do opracowywania i wykonywania programów.

Początkowo komputery były obsługiwane za pośrednictwem konsoli operatora. Oprogramowanie takie, jak asemblerы, programy ładujące i łączące oraz

kompilatory, poprawiło wygodę programowania, niemniej jednak wymagało ono sporo czasu na instalowanie. Aby zmniejszyć czas instalowania zadań, wynajęto operatorów, a podobne zadania łączono w grupy zwane wsadami.

Systemy wsadowe pozwoliły na automatyczne wykonywanie ciągu zadań za pomocą rezydującego w pamięci systemu operacyjnego i w znacznym stopniu zwiększyły ogólne wykorzystanie komputera. Komputer nie musiał już czekać na działania człowieka. Jednak stopień użytkowania jednostki centralnej był wciąż niski z powodu dysproporcji między jej szybkością a małą szybkością urządzeń wejścia-wyjścia. Pośrednie korzystanie z pracy powolnych urządzeń pozwoliło na współpracę z jedną jednostką centralną wielu systemów przenoszenia danych z czytnika na taśmę i z taśmy na drukarkę. Spooling umożliwił wykonywanie przez jednostkę centralną w tym samym czasie operacji wejściowych jednego zadania oraz obliczeń i operacji wyjściowych innych zadań.

Aby polepszyć ogólną wydajność systemu, zrealizowano koncepcję wieloprogramowości. Wieloprogramowość wymaga jednoczesnego przechowywania kilku zadań w pamięci operacyjnej oraz naprzemiennego przełączania jednostki centralnej od jednego zadania do drugiego w celu zwiększenia jej wykorzystania i obniżenia łącznego czasu wykonywania zadań.

Wieloprogramowość, opracowana w celu poprawy działania systemu, pozwoliła również na zrealizowanie podziału czasu. W systemach z podziałem czasu wielu użytkowników (od jednego do kilkuset) posługuje się komputerem w sposób interakcyjny, w tym samym czasie.

Mikrokomputery osobiste są znacznie mniejsze i tańsze od głównych systemów komputerowych. W systemach operacyjnych tych komputerów w wielu przypadkach skorzystano z doświadczeń uzyskanych w toku rozwoju systemów operacyjnych dużych komputerów. Jednak z powodu użytkowania komputerów osobistych przez indywidualne osoby wykorzystanie ich procesorów przestało być zagadnieniem pierwszoplanowym. Dlatego też pewne rozwiązania przyjęte w systemach operacyjnych komputerów głównych nie są odpowiednie dla mniejszych systemów.

W systemach równoległych pewna liczba procesorów pozostaje ze sobą w bliskim kontakcie. Procesory korzystają ze wspólnej szyny, a niekiedy również dzielą wspólną pamięć i urządzenia zewnętrzne. Systemy tego rodzaju mogą zapewniać większą przepustowość i lepszą niezawodność.

System rozproszony jest zbiorem procesorów nie dzielących pamięci ani zegara. Zamiast tego każdy procesor ma własną pamięć lokalną, a procesory utrzymują ze sobą łączność za pomocą różnorodnych linii komunikacyjnych – na przykład za pomocą szybkich szyn lub łączy telefonicznych. System rozproszony umożliwia użytkownikowi dostęp do różnych zasobów znajdujących się w odległych komputerach.

Rygorystyczny system czasu rzeczywistego znajduje często zastosowanie jako sterownik urządzenia o specjalnym przeznaczeniu. W systemie takim istnieją dobrze zdefiniowane i stałe ograniczenia czasowe. Przetwarzanie musi się mieścić w zdefiniowanych ramach czasowych, w przeciwnym razie system nie spełni wymagań. Łagodne systemy czasu rzeczywistego mają mniej napięte ograniczenia czasowe i nie zapewniają planowania w terminach nieprzekraczalnych.

Przedstawiliśmy logikę rozwoju systemów operacyjnych, która wyrażała się przez dodawanie do sprzętu jednostki centralnej udogodnień warunkujących uzyskanie funkcjonalności nowoczesnych systemów operacyjnych. Otworzona trend można obserwować dzisiaj w ewolucji komputerów osobistych, których niedrogi sprzęt jest ciągle ulepszany, przez co zwiększa się ich możliwości.

■ Ćwiczenia

- 1.1 Jakie są trzy główne cele systemu operacyjnego?
- 1.2 Wymień cztery kroki niezbędne do wykonania programu na maszynie pozostającej całkowicie pod bezpośrednim nadzorem jej użytkownika.
- 1.3 Skrajną postacią spoolingu, nazywaną przemieszczeniem (ang. *staging*), jest przesyłanie na dysk całej zawartości taśmy magnetycznej przed jej użyciem. Omów główną zaletę takiego postępowania.
- 1.4 W środowisku wieloprogramowym i wielodostępnym pewna liczba użytkowników wspólnie korzysta z usług systemu. Może to powodować powstawanie różnorodnych problemów związanych z bezpieczeństwem systemu.
 - (a) Wymień dwa takie problemy.
 - (b) Czy w maszynie z podziałem czasu możemy zagwarantować taki sam poziom bezpieczeństwa jak w maszynie dla indywidualnego użytkownika? (Odpowiedź uzasadnij).
- 1.5 Co jest główną zaletą wieloprogramowości?
- 1.6 Czym przede wszystkim różnią się od siebie systemy operacyjne komputerów głównych i komputerów osobistych?
- 1.7 Zdefiniuj najistotniejsze cechy następujących typów systemów operacyjnych:
 - (a) systemu wsadowego;
 - (b) systemu interakcyjnego;

- (c) systemu z podziałem czasu;
 - (d) systemu czasu rzeczywistego;
 - (e) systemu rozproszonego.
- 1.8 Podkreślamy znaczenie systemu operacyjnego jako środka efektywnego wykorzystania sprzętu komputerowego. Kiedy można odstąpić od tej zasady i pozwolić systemowi operacyjnemu na „marmowanie” zasobów? Jak uzasadnić, że system taki nie jest w istocie marnotrawiący?
- 1.9 W jakich warunkach byłoby lepiej dla użytkownika, żeby korzystał z usług systemu z podziałem czasu, a nie z komputera osobistego lub jednostanowiskowej stacji roboczej?
- 1.10 Opisz różnice między przetwarzaniem symetrycznym i asymetrycznym. Wymień trzy zalety i jedną wadę systemów wieloprocesorowych?
- 1.11 Skąd bierze się zapotrzebowanie na systemy rozproszone?
- 1.12 Jaką największą trudność musi pokonać osoba pisząca system operacyjny przeznaczony do pracy w warunkach czasu rzeczywistego?

Uwagi bibliograficzne

Omówienie historii ewolucji sprzętu komputerowego i oprogramowania systemowego zawierają artykuły Rosena [356], Denninga [99] i Weizera [439].

Systemy autonomiczne (przetwarzanie satelitarne) stosowano w systemie IBM FORTRAN Monitor od późnych lat pięćdziesiątych do połowy lat sześćdziesiątych. Spooling zastosowano po raz pierwszy dla komputera Atlas w Manchester University [213]. Spooling był również stosowany w systemie operacyjnym EXEC II dla komputera Univac [265].

Obecnie spooling jest standardową cechą większości systemów, choć we wczesnych latach sześćdziesiątych, gdy wprowadzono system OS/360 dla rodziny komputerów 360 firmy IBM, spooling nie był integralną częścią tego systemu. Wprowadzono go natomiast jako specjalną cechę uzupełniającą w systemie pracującym w centrum obliczeniowym National Aeronautics and Space Administration (NASA) w Houston. Dlatego system ten jest znany pod nazwą HASP (Houston Automatic Spooling Priority).

Systemy z podziałem czasu po raz pierwszy zaproponował Strachey w pracy [409]. Najwcześniejszymi systemami z podziałem czasu były: CTSS, czyli Compatible Time-sharing System opracowany w Massachusetts Institute of Technology (MIT) i opisany w pracy Corbato i in. [86], oraz system SDC

Q-32 zbudowany przez firmę System Development Corporation, przedstawiony w referatach Schwartza i in. [384] oraz Schwartza i Weissmana [383]. Do innych wczesnych, lecz bardziej skomplikowanych systemów należą: system MULTICS, tj. MULTIplexed Information and Computing Services, opracowany w MIT (Corbató i Vyssotsky [85]), system XDS-940 zrealizowany w University of California w Berkeley (Lichtenberger i Pirtle [255]) i system IBM TSS/360 (Lett i Konigsford [249]).

Tabak omawia w książce [413] systemy operacyjne dla sprzętu wieloprocesorowego. Systemy operacyjne dla sieci komputerowych przedstawiają w swoich artykułach: Forsdick i in. [138] i Donnelley [115]. Przegląd rozproszonych systemów operacyjnych podają Tanenbaum i van Renesse [417]. Systemy operacyjne czasu rzeczywistego omawiają Stankowic i Ramamrithan w pracy [403]. Specjalne wydanie miesięcznika *Operating System Review*, poświęcone systemom czasu rzeczywistego, ukazało się pod redakcją Zhao [449].

System MS-DOS oraz komputery PC opisali Norton [311] oraz Norton i Wilton [312]. Przegląd sprzętu i oprogramowania komputera Apple Macintosh zawiera dokumentacja firmowa Apple [13]. Omówienie systemu operacyjnego OS/2 można znaleźć w opisie firmowym Microsoft [289]. Więcej informacji na temat systemu OS/2 zawierają prace Letwina [250] oraz Deitla i Kogana [97]. Custer omawia budowę systemu Microsoft Windows NT w książce [90].

Istnieje spora liczba aktualnych, ogólnych książek o systemach operacyjnych, między innymi takich autorów, jak: Comer [79], Mackawa i in [268], Milenkovic [293], Bic i Shaw [37], Finkel [135], Krakowiak [223], Pinkert i Wear [331], Deitel [96], Stallings [401] i Tanenbaum [416]. Wartościowe bibliografie zawierają artykuły Metznera [284] i Brumfielda [58].



Rozdział 2

STRUKTURY SYSTEMÓW KOMPUTEROWYCH

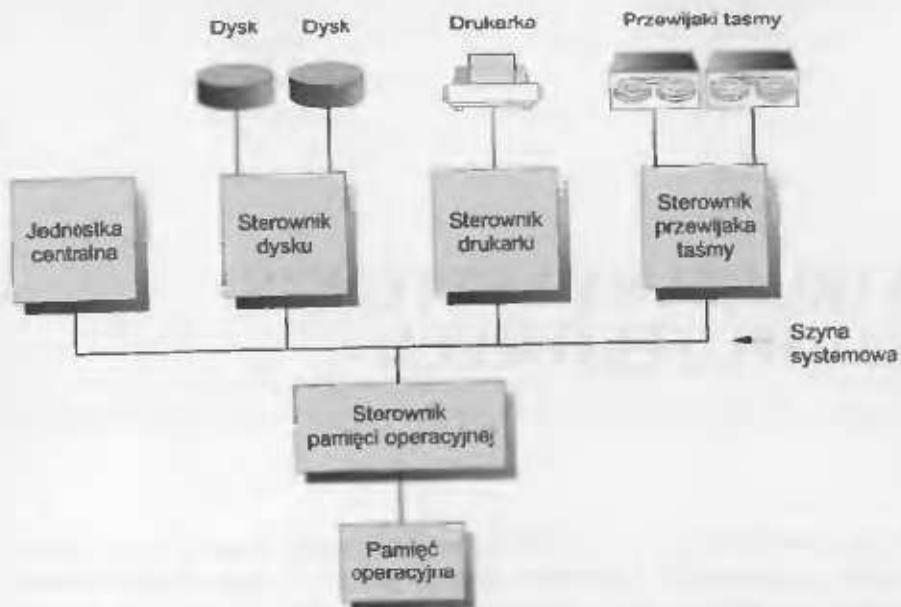
Zanim przejdziemy do szczegółowej analizy działań systemowych, musimy zdobyć ogólną wiedzę o strukturze systemu komputerowego. W tym rozdziale omawiamy odrębne części takiego systemu. Skupimy się głównie na architekturze systemu komputerowego, toteż osoby zaznajomione z tą tematyką mogą ten rozdział przejrzeć pobieżnie lub pominięć. Ponieważ system operacyjny jest niezwykle blisko powiązany z mechanizmami wejścia-wyjścia komputera, najpierw zajmiemy się wejściem-wyjściem. W następnych punktach omówimy strukturę magazynowania danych.

System operacyjny musi również zapewniać poprawne działanie systemu komputerowego. Aby programy użytkowe nie mogły zdezorganizować pracy systemu, sprzęt powinien mieć odpowiednie mechanizmy gwarantujące właściwe zachowanie się całości. W dalszej części tego rozdziału opisujemy podstawową architekturę komputera, umożliwiającą napisanie sprawnie działającego systemu operacyjnego.

2.1 ■ Działanie systemu komputerowego

Nowoczesny, uniwersalny system komputerowy składa się z jednostki centralnej* (ang. *central processor unit* – CPU) i pewnej liczby sprzętowych

* Obok terminu *jednostka centralna* stosujemy w tej książce jego domyślny synonim – *procesor*. Domyślność przy używaniu terminu „procesor” dotyczy jego centralnej funkcji, ponieważ współczesne systemy komputerowe (z kategorii tzw. jednoprocessorowych) zawierają często – oprócz głównego – także inne procesory, o znaczeniu pomocniczym. – Przyp. tłum.

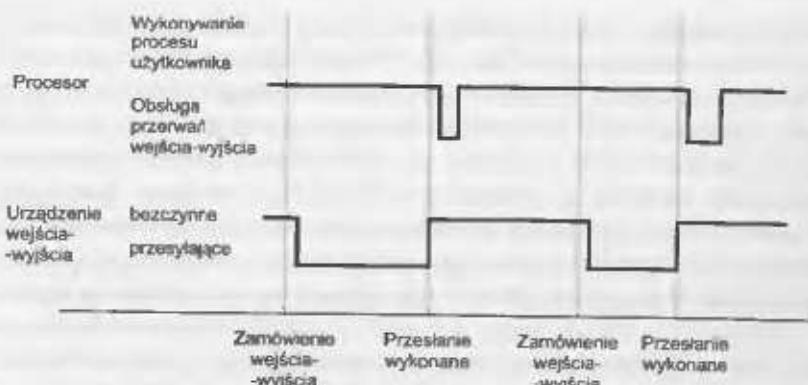


Rys. 2.1 Współczesny system komputerowy

sterowników urządzeń (ang. *device controller*) połączonych wspólną szyną umożliwiającą kontakt ze wspólną pamięcią (rys. 2.1). Każdy sterownik urządzenia odpowiada za określony typ urządzenia (np. za napędy dysków, urządzenia dźwiękowe i wyświetlacze obrazu). Jednostka centralna i sterowniki urządzeń mogą działać współbieżnie, rywalizując o cykle pamięci. Sterownik pamięci ma za zadanie zapewnić uporządkowany, synchroniczny dostęp do wspólnej pamięci.

Aby komputer mógł rozpoczęć pracę, na przykład gdy zostaje podłączony do zasilania lub gdy wznowia się jego działanie, musi w nim nastąpić wykonanie jakiegoś wstępnego programu. Ów wstępny program, nazywany też *programem rozruchowym* (ang. *bootstrap program*), jest zazwyczaj prosty. Określa on stan początkowy wszelkich elementów systemu, poczynając od rejestrów jednostki centralnej, poprzez sterowniki urządzeń, aż po zawartość pamięci. Program rozruchowy musi wiedzieć, jak załadować system operacyjny i rozpocząć jego działanie. Aby wywiązać się z tego zadania, musi on zlokalizować i wprowadzić do pamięci jądro systemu operacyjnego. System operacyjny rozpoczyna wówczas wykonanie swojego pierwszego procesu,

* Tj. magistralą danych; dalej stosujemy termin *szyna* jako równoważny i krótszy.
– Przyp. tłum.



Rys. 2.2 Wykres czasowy przerwań procesu wykonującego operację wyjścia

w rodzaju procesu *init*^{*}, i zaczyna czekać na wystąpienie jakiegoś zdarzenia. Wystąpienie zdarzenia jest na ogół sygnalizowane za pomocą *przerwania* (ang. *interrupt*) pochodzącego od sprzętu lub od oprogramowania. Sprzęt może powodować przerwanie w dowolnej chwili, wysyłając sygnał do jednostki centralnej zwykle za pomocą szyny systemowej. Oprogramowanie może spowodować przerwanie wskutek wykonania specjalnej operacji nazywanej *wywołaniem systemowym* (ang. *system call*), a niekiedy – *wywołaniem monitora*^{**} (ang. *monitor call*).

Istnieje wiele różnych rodzajów zdarzeń mogących powodować przerwanie. Są to na przykład: zakończenie operacji wejścia-wyjścia, dzielenie przez zero, niedozwolony dostęp do pamięci lub zapotrzebowanie na pewną usługę systemu. Każdemu takiemu przerwaniu odpowiada procedura zajmująca się jego obsługą.

Procesor po otrzymaniu sygnału przerwania wstrzymuje aktualnie wykonywaną pracę i natychmiast przechodzi do ustalonego miejsca w pamięci. Miejsce to zawiera na ogół adres startowy procedury obsługującej dane przerwanie. Następuje wykonanie procedury obsługi przerwania, po której zakończeniu jednostka centralna wznowia przerwane obliczenia. Przebieg czasowy tych operacji widać na rys. 2.2.

Przerwania są ważnym elementem architektury komputera. Poszczególne rodzaje komputerów mają indywidualne mechanizmy przerwań, niemniej jednak kilka ich funkcji jest wspólnych. Przerwanie musi przekazywać stereo-

* Tak nazywa się proces rozpoczęty działanie systemu operacyjnego UNIX
– Przyp. tłum.

** Oprócz terminu *wywołanie systemowe* z powodzeniem można używać określeń *funkcja systemowa* lub *odezwanie do systemu* – zależnie od kontekstu. – Przyp. tłum.

wanie do procedury obsługi przerwania. Prosty sposób spowodowania tego polega na wywołaniu ogólnej procedury sprawdzającej informacje opisujące przerwanie, która na tej podstawie wywoła konkretną procedurę obsługi przerwania. Jednak przerwania muszą być obsługiwane szybko, więc przy założeniu, że liczba możliwych przerwań jest zadana z góry, można zamiast takiego postępowania posłużyć się tablicą wskaźników do procedur obsługujących przerwania. Procedura obsługi przerwania jest wówczas wywoływana za pośrednictwem tej tablicy, bez potrzeby korzystania z pośredniczącej procedury. Tablica takich wskaźników jest z reguły przechowywana w dolnej części pamięci (pierwszych 100 komórek lub tp.). Ta tablica, zwana *wektorem przerwań* (ang. *interrupt vector*), jest indeksowana jednoznaczny numerem urządzenia, w który jest zaopatrywane żądanie przerwania, dzięki czemu otrzymuje się właściwy adres procedury obsługującej przerwanie zgłoszone przez dane urządzenie. Nawet tak różne systemy operacyjne, jak MS-DOS i UNLX, kierują przerwania do obsługi w opisany sposób.

W architekturze przerwań trzeba również uwzględniać przechowywanie adresu przerwanego rozkazu. W wielu starych rozwiązaniach ten adres był po prostu przechowywany w ustalonej komorce lub w komórce indeksowanej numerem urządzenia. W nowszych konstrukcjach adres powrotny jest przechowywany na stoisie systemowym. Jeśli procedura obsługi przerwania chce zmienić stan procesora, na przykład przez zmianę wartości rejestrów, to musi jawnie przechować stan bieżący, a przy końcu swojego działa musi go odtworzyć. Po obsłużeniu przerwania następuje pobranie do licznika rozkazów zapamiętanego adresu powrotnego i wznowienie przerwanych obliczeń, tak jakby przerwania nie było.

Zwykle podczas obsługi jednego przerwania inne przerwania są *wyłączone* (ang. *disabled*), więc każde nowe przerwanie jest opóźniane do czasu, aż system upora się z bieżącym przerwaniem i przerwania zostaną *włączone* (ang. *enabled*). Jeśli nie byłoby wyłączenia przerwań, to przetworzenie drugiego przerwania – przy niedokończonej obsłudze pierwszego – mogłoby zniszczyć (przez ponowne zapisanie) dane pierwszego przerwania i spowodować jego utratę (ang. *lost interrupt*). Doskonalsze architektury przerwań zezwalają na obsługę nowego przerwania przed zakończeniem obsługi innego. Zazwyczaj korzysta się w tym celu ze schematu priorytetów, w którym poszczególnym typom zadań nadaje się priorytety według ich względnej ważności, a związane z przerwaniami informacje są pamiętane w osobnym miejscu dla każdego priorytetu. Przerwanie o wyższym priorytecie będzie obsłużone nawet wtedy, gdy jest aktywne jakieś przerwanie o niższym priorytecie, natomiast przerwania tego samego lub niższego poziomu będą *maskowane* (ang. *masked interrupts*), tj. selektywnie wyłączone, co zapobiega utracie przerwań lub występowaniu przerwań niechcianych.

Nowoczesne systemy operacyjne są sterowane przerwaniami (ang. *interrupt driven*). Jeżeli nie ma procesów do wykonania, żadne urządzenia wejścia-wyjścia nie wymagają obsługi i nikt z użytkowników nie oczekuje odpowiedzi, to system operacyjny spokojnie czeka na jakieś zdarzenie. Zdarzenia są prawie zawsze sygnalizowane za pomocą przerwań lub tzw. pułapek. *Pułapka* (ang. *trap*), czyli wyjątek, jest rodzajem przerwania generowanym przez oprogramowanie, a powodowanym albo przez błąd (np. dzielenie przez zero lub próba niewłaściwego dostępu do pamięci), albo przez specjalne zamówienie pochodzące z programu użytkownika, które wymaga obsłużenia przez system operacyjny.

System operacyjny jest sterowany zdarzeniami, co znajduje odzwierciedlenie w jego ogólnej strukturze. Po wykryciu przerwania (lub pułapki) sprzęt przekazuje sterowanie do systemu operacyjnego. System operacyjny w pierwszej kolejności przechowuje bieżący stan jednostki centralnej, zapamiętując zawartość rejestrów i licznika rozkazów. Następnie ustala rodzaj powstałego przerwania. Może to wymagać *odpytywanie* (ang. *polling*), tj. badania stanu wszystkich urządzeń wejścia-wyjścia w celu wykrycia tego, które potrzebuje obsługi, albo może stanowić naturalny wynik zadziałania wektorowego systemu przerwań. Kazdemu rodzajowi przerwania odpowiadają w systemie operacyjnym oddzielne segmenty kodu, określające działania, które należy podjąć w związku z przerwaniem.

2.2 ■ Struktura wejścia-wyjścia

Jak powiedzieliśmy w p. 2.1, uniwersalny system komputerowy składa się z jednostki centralnej i pewnej liczby sprzętowych sterowników urządzeń połączonych za pomocą wspólnej szyny. Każdy ze sterowników urządzeń odpowiada za określony typ urządzenia. Do niektórych rodzajów sterowników można dołączyć więcej niż jedno urządzenie. Na przykład do sterownika *SCSI* (ang. *Small Computer System Interface* – interfejs małego systemu komputerowego), występującego w wielu małych i średniej wielkości komputerach, można dołączyć siedem lub więcej urządzeń. Sprzętowy sterownik urządzenia zarządza pewną ilością lokalnej pamięci buforowej i zbiorem specjalizowanych rejestrów. Sterownik taki odpowiada za przemieszczanie danych między urządzeniami zewnętrznymi, nad którymi sprawuje nadzór, a swoją lokalną pamięcią buforową. Wielkość lokalnego bufora w sterownikach urządzeń zależy od ich rodzaju i rodzaju nadzorowanego urządzenia. Na przykład wielkość bufora sterownika dysku jest taka sama jak wielkość najmniejszej adresowalnej porcji dysku, nazywanej *sektorem* (ang. *sector*), wynoszącej na ogół 512 B – albo jest jej wielokrotnością.

2.2.1 Przerwania wejścia-wyjścia

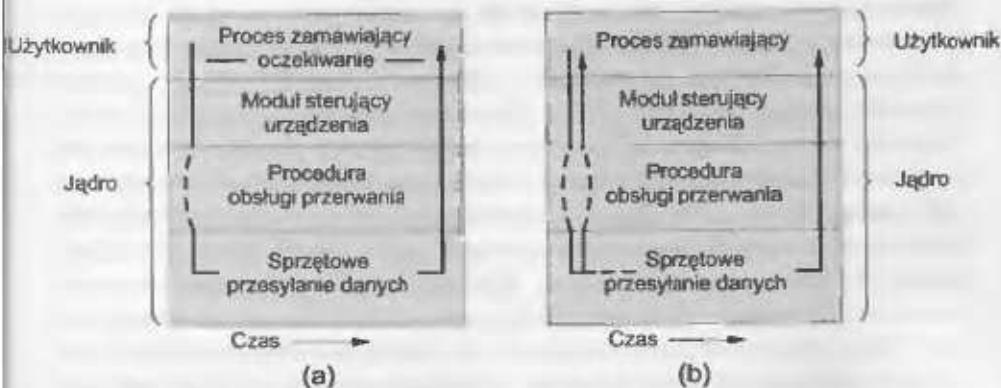
Aby rozpocząć operację wejścia-wyjścia, jednostka centralna określa zawartość odpowiednich rejestrów w sterowniku urządzenia. Sterownik sprawdza dane w tych rejestrach, żeby określić rodzaj mającego nastąpić działania. Jeśli sterownik wykryje na przykład zamówienie czytania, to rozpoczęcie przesyłania danych z urządzenia do swojego lokalnego bufora. Po przesłaniu danych sterownik urządzenia informuje jednostkę centralną, że skończył operację. Aby przekazać tę wiadomość, sterownik powoduje przerwanie.

Opisana sytuacja występuje z reguły jako wynik zamawiania przez proces użytkownika operacji wejścia-wyjścia. Po rozpoczęciu operacji wejścia-wyjścia są możliwe dwa scenariusze zdarzeń. W najprostszym przypadku operacja przesyłania danych rozpoczyna się, kończy, po czym sterowanie wraca do procesu użytkownika. Ten przypadek nazywa się *synchronicznym wejściem-wyjściem* (ang. *synchronous I/O*). Druga możliwość – nazywana *asynchronicznym wejściem-wyjściem* (ang. *asynchronous I/O*) – polega na oddaniu sterowania do programu użytkownika bez czekania na zakończenie operacji. Operacja wejścia-wyjścia może być wtedy kontynuowana wraz z innymi działaniami systemu (rys. 2.3).

Czekanie na zakończenie transmisji może się odbyć na jeden z dwóch sposobów. Niektóre komputery mają specjalny rozkaz *wait* (czekaj), który powoduje beczynność procesora aż do chwili wystąpienia następnego przerwania. Maszyny nie mające takiego rozkazu mogą wykonywać pętlę czekania:

Loop: jmp Loop

Ta zwięzła pętla jest po prostu powtarzana tak długo, aż nadaje sygnał przerwania powodujący przekazanie sterowania do innej części systemu ope-



Rys. 2.3 Dwa sposoby obsługi wejścia-wyjścia: (a) synchroniczny; (b) asynchroniczny

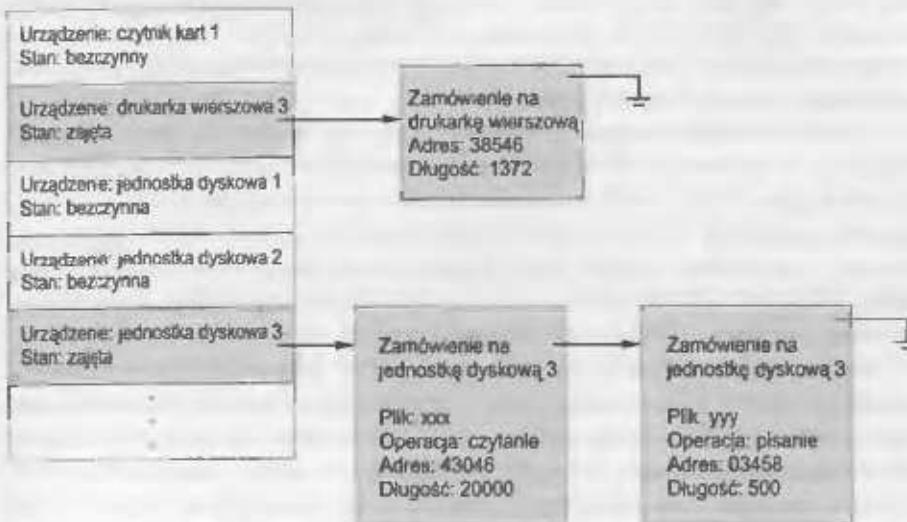
racyjnego. W pętli tego rodzaju może powstać konieczność odpytywania urządzeń wejścia-wyjścia, które nie powodują przerwań, lecz określają po prostu znacznik w jednym z ich własnych rejestrów i oczekują, że system operacyjny zauważą zmianę jego wartości.

Jeżeli jednostka centralna zawsze czeka na koniec operacji wejścia-wyjścia, to w danej chwili tylko jedno zamówienie wejścia-wyjścia pozostaje nie obsłużone. Gdy zatem występuje przerwanie z wejścia-wyjścia, wtedy system operacyjny jest dokładnie poinformowany o tym, które urządzenie wysłało przerwanie. Jednak takie podejście wyklucza równoczesną pracę kilku urządzeń, jak również wyklucza możliwość zachodzenia na siebie w czasie pozytycznych obliczeń i operacji wejścia-wyjścia.

Lepszym rozwiązaniem jest zapoczątkowanie transmisji i kontynuowanie innych działań systemu operacyjnego lub programu użytkownika. Potrzebne jest wówczas wywołanie systemowe, czyli zamówienie odnoszące się do systemu operacyjnego, które w razie potrzeby pozwoliłoby programowi użytkownika zaczekać na zakończenie operacji wejścia-wyjścia. Jeżeli żaden z programów użytkownika nie będzie gotowy do działania, a system operacyjny nie będzie też miał nic innego do roboty, to znów okaże się potrzebny rozkaz czekania lub pętla beczynności – tak jak poprzednio. Musimy również umieć odnotowywać wiele zamówień na operacje wejścia-wyjścia w tym samym czasie. W tym celu system operacyjny posługuje się tablicą, której elementy odnoszą się do poszczególnych urządzeń. Jest to *tablica stanów urządzeń* (ang. *device status table*). Każdy element tej tablicy (rys. 2.4) określa typ urządzenia, jego adres i stan (odłączone, beczynne, zajęte). Jeżeli urządzenie jest zajęte z powodu przyjęcia zamówienia, to odpowiadający mu element tablicy zawiera rodzaj zamówienia i inne parametry. Ponieważ inne procesy mogą składać zamówienia do tego samego urządzenia, system operacyjny będzie utrzymywał dla każdego urządzenia kolejkę, tj. listę, oczekujących zamówień.

Urządzenie wejścia-wyjścia wysyła przerwanie, jeśli wymaga obsługi. Po wystąpieniu przerwania system operacyjny określa najpierw, które urządzenie spowodowało przerwanie. Następnie pobiera z tablicy urządzeń informacje o stanie danego urządzenia i zmienia je, odnotowując wystąpienie przerwania. Zakończenie wykonywania operacji jest przez większość urządzeń wejścia-wyjścia również sygnowane przerwaniem. Jeśli są jakieś następne zamówienia oczekujące na dane urządzenie, to system operacyjny rozpoczyna ich realizację.

Na koniec procedura obsługi przerwania urządzenia wejścia-wyjścia zwraca sterowanie. Jeśli na zakończenie jej działania czekał jakiś program (co zostało odnotowane w tablicy stanów urządzeń), to można oddać mu sterowanie. W przeciwnym razie następuje powrót do tego, co było robione przed



Rys. 2.4 Tablica stanów urządzeń

przerwaniem: do wykonywania programu użytkownika (program rozpoczął operację wejścia-wyjścia, operacja ta się zakończyła, a program nie zaczął jeszcze na nią czekać) albo do pętli czekania (program zapoczątkował dwie lub więcej operacji wejścia-wyjścia i czeka na zakończenie jednej z nich, lecz to przerwanie pochodziło od jakiejś innej). W systemie z podziałem czasu system operacyjny mógłby podjąć wykonywanie innego procesu gotowego do działania.

W niektórych urządzeniach wejściowych mogą występować rozwiązania różniące się od tutaj zaprezentowanego. Wiele systemów interakcyjnych umożliwia użytkownikom pisanie na terminalach z wyprzedzeniem, tzn. wprowadzenie danych zanim zostaną one zapotrzebowane. Występowanie przerwań może wtedy sygnalizować nadchodzenie znaków z terminalu, choć blok kontrolny urządzenia będzie wykazywał brak zamówienia na wejście z danego urządzenia ze strony jakiegokolwiek programu. Jeśli pisanie z wyprzedzeniem ma być dozwolone, to należy zastosować bufor na przechowywanie przekazywanych z wyprzedzeniem znaków, w którym będą one mogły pozostać do czasu, aż któryś z programów zechce z nich skorzystać. W ogólnym przypadku bufor może się okazać potrzebny dla każdego terminalu wejściowego.

Główną zaletą asynchronicznego wejścia-wyjścia jest większa wydajność systemu. W czasie wykonywania operacji wejścia-wyjścia jednostka centralna systemu może być użyta do przetwarzania lub rozpoczęcania operacji wej-

scia-wyjścia odnoszących się do innych urządzeń. Ponieważ operacje wejścia-wyjścia mogą być powolne w porównaniu z szybkością procesora, system wykorzystuje go w znacznie lepszym stopniu. W punkcie 2.2.2 zapoznamy się z innym mechanizmem umożliwiającym poprawę działania systemu.

2.2.2 Struktura DMA

Rozważmy prosty moduł obsługi* terminalu. Gdy ma być przeczytany jeden wiersz danych, wówczas pierwszy napisany znak będzie przesłany do komputera. Po jego nadaniu urządzenie transmisji asynchronicznej (lub port szeregowy), do którego jest przyłączony terminal, wysyla sygnał przerwania do procesora. W chwili nadania od terminalu sygnału przerwania procesor będzie zapewne wykonywać jakiś rozkaz. (Jeśli procesor jest w środku cyklu wykonywania rozkazu, to przerwanie z reguły jest wstrzymywane do czasu zakończenia danego rozkazu). Następuje wtedy zapamiętanie adresu rozkazu, przy którym wystąpiło przerwanie, i przekazanie sterowania do procedury obsługi danego urządzenia.

Procedura obsługi przerwania zapamiętuje bieżące zawartości wszelkich rejestrów procesora, którymi będzie się posługiwać. Sprawdza, czy w związku z poprzednią operacją wejścia nie wystąpiły jakiekolwiek błędy. Potem pobiera znak od urządzenia i zapamiętuje go w buforze. Procedura przerwania musi także uaktualnić wskaźnik i zmienne licznikowe bufora, aby następny znak z wejścia mógł być zapamiętyany w następnej komórce bufora. Z kolei procedura przerwania ustawia znaczniki w pamięci, wskazując innym częściami systemu operacyjnego, że otrzymano nowe dane wejściowe. Inne części systemu odpowiadają za przetwarzanie danych w buforze i przekazywanie znaków do programu, który żąda danych wejściowych (por. p. 2.5). Następnie procedura obsługi przerwania odtwarza poprzednią zawartość wszystkich używanych przez nią rejestrów i oddaje sterowanie do przerwanego rozkazu.

Jeśli znaki są pisane na terminalu o wydajności 9600 bodów, to mogą one być akceptowane i przesyłane w przybliżeniu co 1 ms (1000 µs). Dobrze napisanej procedurze obsługi przerwania mogą wystarczyć 2 µs na wprowadzenie znaku do bufora. Z każdego zatem tysiąca µs procesorowi pozostaje 998 µs na obliczenia i obsługę innych przerwań. Na skutek tej dysproporcji przerwania asynchronicznego wejścia-wyjścia otrzymują zwykle niski priorytet, dzięki czemu przerwania ważniejsze mogą być obsługiwane w pierwszej kolejności, a nawet powodować zaniechanie obsługi bieżącego, mniej ważnego przerwania. Natomiast szybkie urządzenia, takie jak taśmy, dyski

* Nazwy *moduł obsługi* lub *moduł sterujący* będą dalej odpowiednikami angielskiego terminu *driver*. — Przyp. tłum.

albo sieci komunikacyjne, mogą przesyłać informacje z szybkością zbliżoną do szybkości pamięci operacyjnej. Mogłoby się zatem zdarzyć, że procesor potrzebuje 2 µs na obsługę każdego przerwania, a nadchodzą one (na przykład) co 4 µs. Nie zostaje więc wiele czasu na wykonywanie procesu.

Problem ten rozwiązuje się, umożliwiając szybkim urządzeniom wejścia-wyjścia *bezpośredni dostęp do pamięci operacyjnej* (ang. *direct memory access* – DMA). Po ustawieniu buforów, wskaźników i liczników sterownik danego urządzenia przesyła bezpośrednio cały blok danych między własnym buforem a pamięcią – bez interwencji procesora. Przerwanie wypada wówczas jeden raz na cały blok danych, a nie po przesłaniu każdego znaku (lub słowa), jak to się dzieje w przypadku powolnych urządzeń zewnętrznych.

Zasadnicze działanie jednostki centralnej pozostaje niezmienione. Program użytkownika lub sam system operacyjny może zażądać przesłania danych. System operacyjny wybiera bufor (pusty wejściowy lub pełny wyjściowy) z kolejki buforów do przesłania. (W zależności od typu urządzenia bufor ma zazwyczaj od 128 do 4096 bajtów). Następnie część systemu operacyjnego, zwana modelem obsługi urządzenia (ang. *device driver*), ustawia w rejestrach sterownika DMA odpowiednie adresy źródła i miejsca przeznaczenia oraz długość transmisji. Z kolei sterownik DMA zostaje poinstruowany, że należy zainicjować operację wejścia-wyjścia. Gdy sterownik DMA jest zajęty przesyaniem danych, jednostka centralna może wykonywać inne zadania. Ponieważ pamięć operacyjna może zazwyczaj przesyłać tylko jedno słowo w danej chwili, więc sterownik DMA „kradnie” cykle pamięci jednostce centralnej. Ta kradzież cykli może spowalniać działanie jednostki centralnej w trakcie przesyłania DMA. Po zakończeniu przesyłania sterownik DMA wysyła jednostce centralnej przerwanie.

2.3 ■ Struktura pamięci

Programy muszą znajdować się w pamięci operacyjnej, aby mogły być wykonywane. *Pamięć operacyjna* (ang. *main memory*) jest jedynym wielkim obszarem pamięci dostępnym dla procesora bezpośrednio. Tworzy ona tablicę słów lub bajtów, których liczba waha się od setek tysięcy do setek milionów. Każde słowo ma własny adres. Współpraca z pamięcią operacyjną odbywa się za pomocą ciągu rozkazów **load** (pobierz) lub **store** (przechowaj) odnoszących się do określonych adresów. Rozkaz **load** powoduje pobranie słowa z pamięci operacyjnej do wewnętrznego rejestru jednostki centralnej^{*}, nato-

* Lub inaczej: *pamięć główna*. – Przyp. tłum.

** Nie oznaczając, oczywiście, oryginalnego słowa z pamięci. – Przyp. tłum.

miast rozkaz **store** powodując umieszczenie zawartości rejestru procesora w pamięci operacyjnej. Oprócz jawnego pobrania i przechowania jednostka centralna automatycznie pobiera z pamięci operacyjnej rozkazy do wykonania.

Typowy cykl rozkazowy w systemie o architekturze von Neumanna zaczyna się od pobrania rozkazu z pamięci i przesłania go do *rejestru rozkazów* (ang. *instruction register*). Rozkaz jest następnie dekodowany i realizowany (może spowodować pobranie argumentów z pamięci i umieszczenie ich w innym rejestrze wewnętrznym). Po wykonaniu rozkazu na argumentach jego wynik można z powrotem przechować w pamięci. Zauważmy, że jednostka pamięci „widzi” tylko strumień adresów pamięci. Nie jest jej znany sposób, w jaki one powstały (licznik rozkazów, indeksowanie, modyfikacje pośrednie, adresy literalne itp.) ani czemu służą (rozkazy lub dane). Z uwagi na to możemy zaniedbać sposób generowania adresu pamięci przez program. Interesujemy się tylko ciągiem adresów pamięci wytwarzanych przez wykonywany program.

W idealnych warunkach moglibyśmy sobie życzyć, aby program i dane stale pozostawały w pamięci operacyjnej. Nie jest to możliwe z dwu powodów:

1. Pamięć operacyjna jest zazwyczaj za mała, aby przechowywać na stałe wszystkie potrzebne programy i dane.
2. Pamięć operacyjna jest tzw. *pamięcią ulotną* (nietrwałą; ang. *volatile storage*). Traci ona swoją zawartość po odłączeniu zasilania.

Wobec tego większość systemów komputerowych jest wyposażona w *pamięć pomocniczą* (ang. *secondary storage*), która rozszerza pamięć operacyjną. Od pamięci pomocniczej wymaga się przede wszystkim, aby mogła trwale przechowywać duże ilości danych.

Najpopularniejszym urządzeniem pamięci pomocniczej jest *dysk magnetyczny* umożliwiający zapamiętywanie zarówno programów, jak i danych. Większość programów (przeglądarki WWW, kompilatory, procesory tekstu, arkusze kalkulacyjne itd.) przechowuje się na dysku, zanim nie nastąpi ich umieszczenie w pamięci operacyjnej. Wiele programów używa potem dysku zarówno jako źródła, jak i miejsca przeznaczenia przetwarzanych przez siebie informacji. Dlatego też właściwe zarządzanie pamięcią dyskową na zasadnicze znaczenie w systemie komputerowym, co omówimy w rozdz. 13.

W szerszym jednak rozumieniu przedstawiona przez nas struktura pamięci, składająca się z rejestrów, pamięci operacyjnej i dysków magnetycznych, jest tylko jednym z wielu możliwych systemów pamięci. Istnieją także pamięci podrzędne, pamięci na płytach kompaktowych (CD-ROM), taśmy magnetyczne itd. Każdy system pamięci spełnia podstawowe funkcje przechowywania danych do czasu, gdy zostaną one z niego odzyskane. To, co przede wszyst-

kim różni te systemy pamięci, to szybkość działania, koszt, rozmiar i ulotność (danych). W punktach od 2.3.1 do 2.3.3 opisujemy pamięć operacyjną, dyski magnetyczne i taśmy magnetyczne, gdyż ilustrują one ogólne właściwości wszystkich, komercyjnie ważnych urządzeń pamięci. W rozdziałach 13 i 14 omówimy specyficzne cechy wielu konkretnych urządzeń, takich jak dyski elastyczne, dyski twarde, pamięci CD-ROM oraz urządzenia DVD.

2.3.1 Pamięć operacyjna

Pamięć operacyjna oraz rejesty wbudowane w procesor są jedynymi rodzajami pamięci dostępnej dla jednostki centralnej bezpośrednio. (Zauważmy, że istnieją rozkazy, których argumentami są adresy pamięci operacyjnej, lecz nie ma rozkazów posługujących się adresami dyskowymi). Z tego powodu każdy wykonywany rozkaz i wszystkie używane przez niego dane muszą znajdować się w jednym z tych urządzeń pamięci o dostępie bezpośredniem. Jeżeli danych nie ma w pamięci operacyjnej, to należy je do niej sprowadzić, zanim jednostka centralna zacznie je przetwarzać.

W przypadku urządzeń wejścia-wyjścia każdy sterownik wejścia-wyjścia zawiera, jak wspomnialiśmy w p. 2.1, rejesty do przechowywania rozkazów i przesyłanych danych. Specjalne operacje wejścia-wyjścia umożliwiają przesyłanie danych między tymi rejestrami a pamięcią systemu. Aby ułatwić dostęp do urządzeń wejścia-wyjścia, w wielu architekturach komputerów stosuje się *wejście-wyjście odwzorowywane w pamięci* (ang. *memory-mapped I/O*). W tym przypadku pewna część adresów pamięci operacyjnej zostaje wydzielona i odwzorowana na rejesty urządzeń. Operacje czytania i zapisywania miejsc określonych przez te adresy powodują przesyłanie danych z lub do rejestrów urządzeń. Metoda ta jest odpowiednia dla urządzeń o krótkich czasach reakcji, takich jak wideosterowniki. W komputerze IBM PC każde miejsce ekranu jest odwzorowane w komórce pamięci. Wyświetlanie tekstu na ekranie jest niemal tak samo łatwe, jak jego wpisywanie do odpowiednich miejsc odwzorowanych w pamięci.

Odwzorowywanie w pamięci wejścia-wyjścia jest także wygodne dla innych urządzeń, takich jak porty szeregowe i równoległe, stosowane do podłączania do komputerów modemów i drukarek. Za pomocą tego rodzaju urządzeń jednostka centralna przesyła dane, czytając i zapisując niewielką liczbę rejestrów urządzeń zwanych *portami wejścia-wyjścia* (ang. *I/O ports*). Aby wysłać długą ciąg bajtów przez odwzorowany w pamięci port szeregowy, procesor wpisuje jeden bajt danych do rejestru danych, a następnie ustawia bit w rejestrze kontrolnym na wartość sygnalizującą, że bajt jest dostępny. Urządzenie pobiera ten bajt danych i zeruje bit w rejestrze kontrolnym, sygnalizując, że jest gotowe na przyjęcie kolejnego bajta. Procesor może wówczas

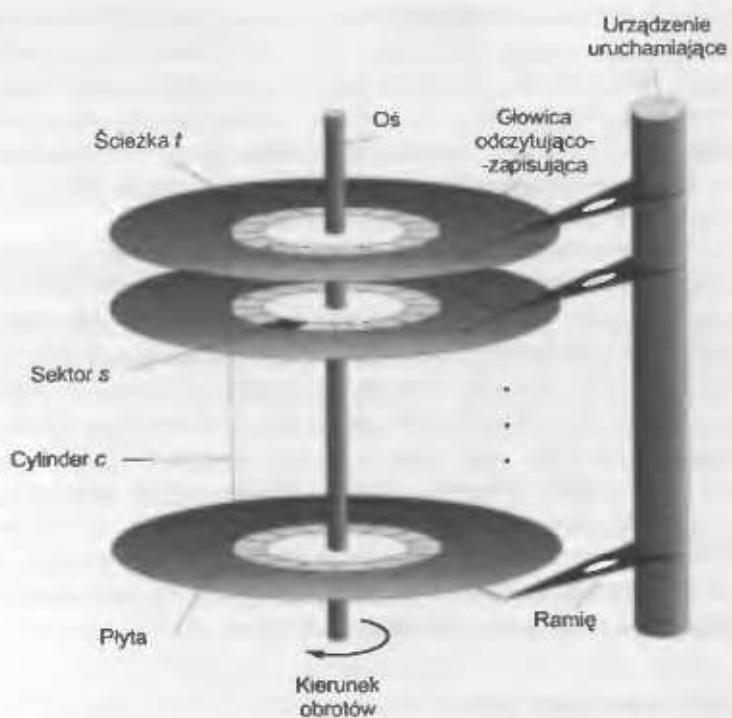
przesłać następny bajt. Jeżeli procesor stosuje odpytywanie do obserwowania bitu kontrolnego, wciąż wykonując pętlę sprawdzania, czy urządzenie jest gotowe, to taką metodę działania nazywa się *programowanym wejściem-wyjściem* (ang. *programmed I/O* – PIO). Jeżeli procesor nie odpytuje portu kontrolnego, otrzymując w zamian przerwanie, gdy urządzenie stanie się gotowe na przyjęcie następnego bajta, to o takim przesyłaniu danych mówi się, że jest *sterowane przerwaniami* (ang. *interrupt driven*).

Rejestry wbudowane w jednostkę centralną są na ogół dostępne w jednym cyklu jej zegara. Większość procesorów może dekodować rozkazy i wykonywać proste działania na zawartości rejestrów z szybkością jednej lub więcej operacji na jeden impuls zegara. Nie można tego powiedzieć o pamięci operacyjnej, do której dostęp odbywa się za pośrednictwem transakcji z szyną pamięci. Dostęp do pamięci może zajmować wiele cykli, a wtedy procesor zazwyczaj musi utykać (ang. *stall*), gdyż brakuje mu danych do zakończenia rozkazu, który właśnie wykonuje. Jest to sytuacja nie do przyjęcia, zważwszy na częstotliwość kontaktów z pamięcią. Ratunkiem jest wstawienie między jednostkę centralną a pamięć operacyjną jakiejś szybkiej pamięci. Bufor pamięci stosowany do niwelowania różnic w szybkości nazywa się *pamięcią podręczną* (ang. *cache*), co omawiamy w p. 2.4.1.

2.3.2 Dyski magnetyczne

Dyski magnetyczne stanowią zdecydowaną większość pamięci pomocniczych współczesnych systemów komputerowych. Zasada działania dysków jest stosunkowo prosta (rys. 2.5). Każda płyta (ang. *platter*) dysku ma kształt kolistego, jak płyta kompaktowa. Średnice popularnych płyt wahają się w przedziale od 1,8 do 5,25 cala. Obie powierzchnie płyty są pokryte materiałem magnetycznym, podobnym do stosowanego na taśmach magnetycznych. Informacje przechowuje się przez odpowiednie namagnesowanie warstwy magnetycznej.

Głowice odczytająco-zapisujące unoszą się tuż nad powierzchnią każdej płyty. Są one przymocowane do *ramienia dysku* (ang. *disk arm*), które przymieszcza je wszystkie jednocześnie. Powierzchnia płyty jest logicznie podzielona na koliste ścieżki (ang. *tracks*), które z kolei dzielą się na sektory (ang. *sectors*). Zbiór ścieżek przy danym położeniu ramienia tworzy cylinder (ang. *cylinder*). Na dysku mogą być tysiące koncentrycznych cylinderów, a każda ścieżka może zawierać setki sektorów. Pojemność pamięci popularnych napędów dysków mierzy się w gigabajtach. (Kilobajt to 1024 bajty, megabajt ma 1024^2 bajtów, a gigabajt oznacza 1024^3 bajtów, lecz producenci dysków często zaokrąglają te liczby, mówiąc że megabajt wynosi 1 milion bajtów, a gigabajt to miliard bajtów).



Rys. 2.5 Mechanizm dysku z ruchomymi głowicami

Podeczas pracy dysk wiruje z dużą prędkością, wprawiany w ruch przez silnik jego napędu. Prędkość obrotowa większości napędów wynosi od 60 do 150 obrotów na sekundę. Szybkość dysku jest określana przez dwa czynniki. *Tempo przesyłania* (ang. *transfer rate*) oznacza szybkość, z jaką dane przepływają między napędem dysku a komputerem. Na czas ustalania położenia głowicy (ang. *positioning time*), niekiedy nazywany czasem losowego dostępu (ang. *random access time*) składają się: czas przesuwania głowicy do odpowiedniego cylindra, nazywany czasem wyszukiwania (ang. *seek time*), oraz czas, w którym potrzebny sektor, obracając się, przejdzie pod głowicą. Czas ten jest nazywany opóźnieniem obrotowym (ang. *rotational latency*). Typowe dyski mogą przesyłać kilka megabajtów danych na sekundę, a ich czasy wyszukiwania i opóźnienia obrotowe wynoszą kilka milisekund.

Ponieważ głowica dysku unosi się na niezwykle cieniutkiej poduszce powietrznej (mierzonej w mikronach), istnieje niebezpieczeństwo zetknięcia się jej z powierzchnią dysku. Chociaż płyty dysku są powleczone warstwą ochronną, czasami głowica uszkadza powierzchnię magnetyczną. Wypadek

taki jest nazywany *awarią głowicy* (ang. *head crash*). Awaria głowicy zazwyczaj jest nieusuwalna i powoduje konieczność wymiany całego dysku.

Dysk może być *wymienny* (ang. *removable*), co umożliwia zamontowywanie różnych dysków według potrzeby. Wymienne dyski magnetyczne to na ogół jedna płyta trzymana w plastikowym pudełku, aby uchronić ją przed uszkodzeniem po wyjęciu z napędu dysku. *Dyski elastyczne* (ang. *floppy disks*) są niedrogimi wymiennymi dyskami, mającymi miękką, plastikową obudowę, zawierającą giętką płytę. Głowica dysku elastycznego z reguły spoczywa na jego powierzchni, dlatego jego napęd jest zaprojektowany na wolniejsze obroty niż napęd dysku twardego, aby zmniejszyć ścieranie powierzchni dyskowej. Typowa pojemność pamięci dysku elastycznego wynosi zaledwie ok. 1 MB. Są również w użyciu dyski wymienne działające niczym zwykłe dyski twarde i mające pojemności mierzone w gigabajtach.

Napęd dysku jest podłączony do komputera za pomocą wiązki przewodów nazywanych *szyną wejścia-wyjścia* (ang. *I/O bus*). Jest kilka rodzajów szyn, w tym EIDE i SCSI. Przesyłanie danych szyną odbywa się pod nadzorem specjalnych, elektronicznych procesorów, nazywanych *sterownikami* (ang. *controllers*). *Sterownik macierzysty* (ang. *host controller*) to sterownik po stronie szyny przylegającej do komputera. *Sterownik dysku* (ang. *disk controller*) jest wbudowany w każdy napęd dyskowy. Aby wykonać dyskową operację wejścia-wyjścia, komputer umieszcza rozkaz w sterowniku macierzystym, na ogół za pomocą portów wejścia-wyjścia odwzorowanych w pamięci, jak opisaliśmy w p. 2.3.1. Sterownik macierzysty wysyła następnie polecenie w formie komunikatu do sterownika dysku, a ten uruchamia napęd dysku w celu wykonania polecenia. Sterowniki dysków zazwyczaj mają wbudowaną pamięć podręczną. Przesyłanie danych w sterowniku dysku odbywa się między pamięcią podręczną a powierzchnią dyskową, natomiast przesyłanie danych po stronie komputera przebiega szybko, bo z szybkościami elektronicznymi i zachodzi między pamięcią podręczną a sterownikiem macierzystym.

2.3.3 Taśmy magnetyczne

Taśma magnetyczna (ang. *magnetic tape*) jako nośnik pamięci pomocniczej była używana od dawna. Choć jest ona względnie trwała i można na niej przechowywać wielkie ilości danych, ma długi czas dostępu w porównaniu z pamięcią operacyjną. Dostęp losowy do taśmy magnetycznej jest tysiące razy wolniejszy niż dostęp losowy do dysku magnetycznego, toteż taśmy nie

* W użyciu jest też termin *magistrala wejścia-wyjścia*. – Przyp. tłum.

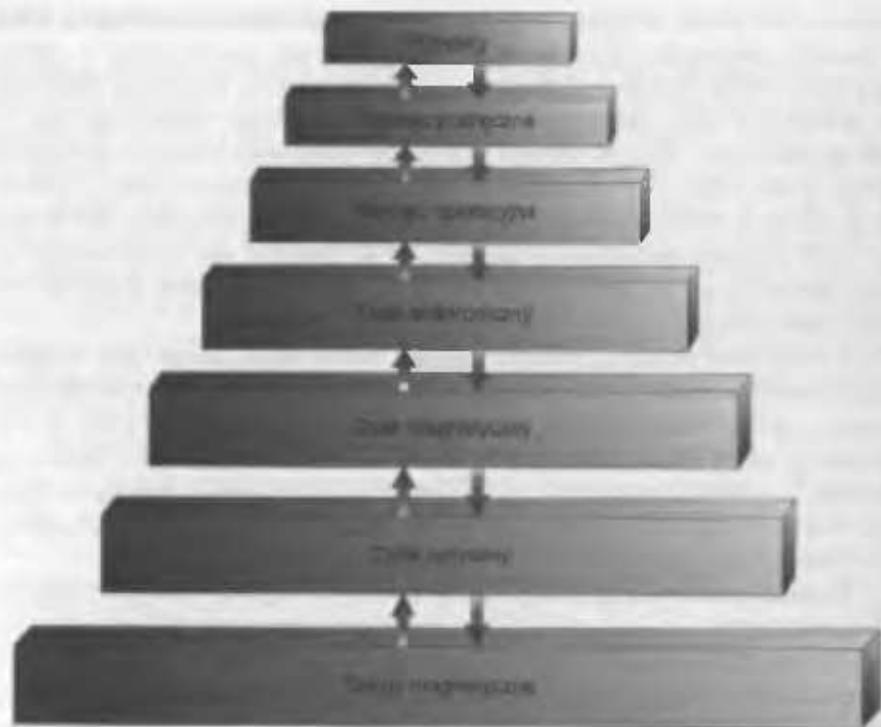
są wygodne w roli pamięci pomocniczej. Taśm uzywa się głównie do przechowywania informacji rzadko używanych oraz jako środka do transportu informacji z jednego systemu do drugiego.

Taśma znajduje się na szpuli i przewija się w jedną bądź w drugą stronę pod głowicą odczytująco-zapisującą. Przesunięcie taśmy do właściwego miejsca może zajmować minuty, lecz z chwilą jego odnalezienia napędy taśmowe mogą zapisywać dane z szybkościami porównywalnymi do szybkości napędów dysków. Pojemności taśmy znacznie się różnią między sobą i zależą od poszczególnych rodzajów napędu taśmy. Niektóre taśmy mogą przechowywać 20 razy więcej danych niż wielkie napędy dysków. Taśmy dzieli się według szerokości: są taśmy 4-, 8- i 19-milimetrowe oraz taśmy o szerokości 1/4 i 1/2 cala.

2.4 ■ Hierarchia pamięci

Rozmaite rodzaje pamięci w systemie komputerowym można zorganizować w hierarchię (rys. 2.6) zależnie od ich szybkości i kosztów. Na najwyższych poziomach pamięci są drogie, za to szybkie. W miarę przemieszczania się w dół hierarchii, ogólnie biorąc maleje cena jednego bitu, natomiast wydłuża się czas dostępu. Ten kompromis jest zrozumiały: gdyby dany system pamięci był zarówno szybszy, jak i tańszy od innych, przy takich samych innych właściwościach, to nie byłoby sensu używać pamięci wolniejszej i droższej. W rzeczywistości wiele wczesnych urządzeń pamięci, wliczając w to taśmę papierową i pamięci rdzeniowe, trafiło do muzeów z chwilą, gdy taśma magnetyczna i pamięci półprzewodnikowe stały się szybsze i tańsze.

Oprócz szybkości i kosztu różnych systemów pamięci uwzględnia się także jej ulotność. Pamięć ulotna traci zawartość po odłączeniu od niej zasilania. W razie braku drogich, rezerwowych źródeł zasilania (baterii lub generatorów) w celu bezpiecznego przechowywania dane należy zapisywać w pamięciach nieulotnych. W hierarchii pokazanej na rys. 2.6 systemy pamięci leżące powyżej różnych typów dysków są ulotne, a te, które znajdują się poniżej pamięci operacyjnej, są nieulotne. Projektując pełny system pamięci, należy równoważyć wszystkie te czynniki. Drogiej pamięci używa się tylko w niezbędnych ilościach, natomiast pamięci taniej i nieulotnej dostarcza się w ilościach możliwie dużych. W celu niwelowania różnic w wydajności, tam, gdzie występują długie czasy dostępu lub dysproporcje w szybkości przesyłania między dwoma składowymi, można instalować pamięci podrzczne.



Rys. 2.6 Hierarchia pamięci

2.4.1 Przechowywanie podręczne

Stosowanie pamięci podręcznej (ang. *caching*) jest ważną zasadą przy projektowaniu systemów komputerowych. W normalnych warunkach informacje są przechowywane w jakimś systemie pamięci (np. w pamięci operacyjnej). Przed ich użyciem są kopiowane do szybszego systemu pamięci – tj. do pamięci podręcznej – na okres przejściowy. Gdy jest potrzebny jakiś fragment informacji, wtedy sprawdza się najpierw, czy nie ma go w pamięci podręcznej. Jeśli jest, informacje pobiera się wprost z pamięci podręcznej; jeśli zaś nie, to korzysta się z informacji w głównym systemie pamięci, umieszczając ich kopię w pamięci podręcznej przy założeniu, że istnieje duże prawdopodobieństwo, że będą one znów potrzebne.

Rozszerzając ten punkt widzenia na wewnętrzne, programowalne rejesty w rodzaju rejestrów indeksowych, możemy spojrzeć na nie jak na szybką pamięć podręczną pamięci operacyjnej. Programista (lub kompilator) implementuje przydział rejestrów i algorytmy zastępowania ich zawartości, decy-

dując o tym, która informacja ma być przechowywana w rejestrach, a która w pamięci operacyjnej. Istnieją też pamięci podręczne zrealizowane w całości sprzętowo. Na przykład większość systemów ma pamięć podręczną rozkazów do przechowywania następnego rozkazu, co do którego przewiduje się, że będzie wykonany. Bez tej pamięci podręcznej jednostka centralna musiałaby czekać przez kilka cykli na pobranie rozkazu z pamięci operacyjnej. Z podobnych przyczyn większość systemów ma w hierarchii pamięci jedną lub więcej szybkich pamięci podręcznych danych. W tej książce nie interesujemy się tymi sprzętowymi pamięciami podręcznymi, gdyż są one poza kontrolą systemu operacyjnego.

Zarządzanie pamięcią podręczną (ang. *cache management*) jest ważnym zagadnieniem projektowym ze względu na ograniczone rozmiary tych pamięci. Staranny dobór wielkości pamięci podręcznej i polityki zastępowania w niej informacji może spowodować, że 80 do 99% wszystkich dostępów będzie się odnosić do pamięci podręcznej, co w dużym stopniu usprawni działanie systemu. Rozmaite algorytmy zastępowania informacji w programowo nadzorowanych pamięciach podręcznych są omówione w rozdz. 9.

Pamięć operacyjną (główną) można uważać za szybką pamięć podręczną dla pamięci pomocniczej, gdyż dane z pamięci pomocniczej muszą być przed użyciem kopiowane do pamięci operacyjnej, a dane przeznaczone do przeniesienia do pamięci pomocniczej w celu bezpiecznego przechowywania muszą wpierw znajdować się w pamięci operacyjnej. Dane systemu plików mogą występować na kilku poziomach w hierarchii pamięci. Na najwyższym poziomie system operacyjny może utrzymywać pamięć podręczną danych systemu plików w pamięci operacyjnej. Do bardzo szybkiego, ulotnego pamiętania można również stosować elektroniczne RAM-dyski, udostępniane za pomocą interfejsu systemu plików. Duża ilość pamięci pomocniczej znajduje się na dyskach magnetycznych. Pamięć dysków magnetycznych jest z kolei często składowana na taśmach lub dyskach wymiennych w celu ochrony przed utratą danych w przypadku awarii dysku twardego. Niektóre systemy automatycznie archiwizują stare pliki danych z pamięci pomocniczej w pamięci trzeciorzędnej, takiej jak roboty kasetowe (ang. *jukeboxes*), w celu obniżenia kosztów ich magazynowania (zob. p. 14.2.3).

Przemieszczanie informacji między poziomami hierarchii pamięci może być jawne lub niejawne – zależnie od konstrukcji sprzętu i nadzoru ze strony oprogramowania systemu operacyjnego. Na przykład przesyłanie danych z pamięci podręcznej do jednostki centralnej i rejestrów jest zwykle funkcją sprzętową, nie wymagającą żadnej interwencji ze strony systemu operacyjnego. Z kolei przesyłanie danych z dysku do pamięci operacyjnej jest zazwyczaj nadzorowane przez system operacyjny.

2.4.2 Zgodność i spójność

W hierarchicznej strukturze pamięci te same dane mogą występować na różnych jej poziomach. Rozważmy na przykład liczbę całkowitą A umieszczoną w pliku B, która ma być zwiększaona o 1. Założymy, że plik B rezyduje na dysku magnetycznym. Operację zwiększania poprzedza wykonanie operacji wejścia-wyjścia mającej na celu skopiowanie bloku dyskowego z liczbą A do pamięci operacyjnej. Po tej z kolei operacji może nastąpić przekopiowanie A do pamięci podręcznej, a stamtąd – do wewnętrznego rejestrów. Tak więc kopia liczby A pojawia się w kilku miejscach. Z chwilą wykonania operacji zwiększania w rejestrze wewnętrznym wartość A będzie różna w różnych systemach pamięci. Stanie się ona taka sama dopiero po przekopiowaniu jej z powrotem na dysk magnetyczny.

W środowisku, w którym w danym czasie jest wykonywany tylko jeden proces, sytuacja taka nie powoduje żadnych trudności, ponieważ dostęp do liczby całkowitej A będzie dotyczyć zawsze jej kopii na najwyższym poziomie hierarchii. Jednak w środowisku wielozadaniowym, w którym procesor jest przełączany tam i z powrotem między różnymi procesami, należy przedsięwziąć skrajne środki ostrożności, aby zapewnić, że w przypadku gdy kilka procesów będzie chciało sięgnąć po A, wówczas każdy z nich otrzyma jej najnowszą wartość.

Sytuacja staje się bardziej skomplikowana w środowisku wieloprocesorowym, gdzie – oprócz utrzymywaniaewnętrznych rejestrów – jednostka centralna zawiera również lokalną pamięć podręczną. W takim środowisku kopia zmiennej A może istnieć jednocześnie w wielu pamięciach podręcznych. Ponieważ różne jednostki centralne mogą działać jednocześnie, musimy więc zapewnić, że uaktualnienie wartości A w jednej z pamięci podręcznych znajdzie natychmiast odbicie we wszystkich innych pamięciach podręcznych, które również przechowują zmienną A. Problem ten zwie się *zgodnością pamięci podręcznej* (ang. *cache coherency*) i zazwyczaj jest rozwiązywany sprzętowo (jego obsługa odbywa się poniżej poziomu systemu operacyjnego).

W środowisku rozproszonym sytuacja komplikuje się w jeszcze większym stopniu. W takim środowisku w różnych, przestrzennie odległych od siebie komputerach może być przechowywanych wiele kopii (replik) tego samego pliku. Ze względu na to, że różne repliki mogą być czytane i aktualizowane współbieżnie, należy zapewnić, że aktualizacja kopii w jednym miejscu pociągnie za sobą możliwie jak najszybsze uaktualnienie wszystkich innych kopii. Istnieje wiele różnych sposobów osiągania tego stanu, co przedstawimy w rozdz. 17.

2.5 ■ Ochrona sprzętowa

Wczesne systemy komputerowe były systemami jednostanowiskowymi, w których programista był zarazem operatorem. Programiści obsługujący komputer za pomocą konsoli sprawowali nad nim pełny nadzór. Z chwilą powstania systemów operacyjnych nadzór przekazano systemowi operacyjnemu. Począjąc od rezydencnego monitora, system operacyjny zaczął wykonywać wiele funkcji, zwłaszcza dotyczących wejścia-wyjścia, za które uprzednio był odpowiedzialny programista.

Ponadto, aby polepszyć wykorzystanie systemu, system operacyjny począł dzielić (ang. *share*) zasoby systemowe między pewną liczbę programów jednocześnie. Przy użyciu spoolingu można było wykonywać jeden program, a jednocześnie radzić sobie z przesyaniem danych do innych procesów; dysk przechowywał jednocześnie dane dla wielu procesów. Wieloprogramowość spowodowała konieczność koegzystencji wielu programów w pamięci w tym samym czasie.

Ów podział spowodował zarówno poprawę użytkowania, jak i zwiększenie liczby problemów. Gdy system działał bez podziału, wówczas błąd w programie mógł powodować trudności tylko w jednym programie, który właśnie był wykonywany. W sytuacji podziału zasobów komputera na szkodliwe skutki błędu w jednym programie mogły być narażonych wiele procesów.

Rozważmy na przykład prosty wsadowy system operacyjny, który jedynie automatycznie porządkuje zadania (p. 1.2). Założymy, że program ugrzązł w pętli czytania kart wejściowych. Program taki przeczytałby wszystkie swoje dane i dopóki coś by go nie zatrzymało, dopóty kontynuowałby czytanie kart należących do następnych zadań itd. Spowodowałoby to zaburzenie pracy wielu zadań.

W systemach wieloprogramowych mogłyby się zdarzać znacznie trudniejsze do wykrycia błędy, gdyby jakiś „rozbrykany” program pozmieniał dane innego programu lub nawet program samego monitora rezydencznego. Zarówno system MS-DOS, jak i Macintosh OS dopuszcza występowanie błędów tego rodzaju.

Bez ochrony przed tego rodzaju błędami komputer musi wykonywać w danej chwili tylko jeden proces albo wszystkie wyniki należy uznać za podejrzane. Dobrze zaprojektowany system operacyjny musi gwarantować, że niepoprawny (lub „złośliwy”) program nie będzie mógł zakłócić działania innych programów.

Wiele błędów programowania jest wykrywanych przez sprzęt. Tymi błędami zajmuje się na ogół system operacyjny. Gdy program użytkownika dopuści się jakiegoś uchybienia, na przykład próbuje wykonać niedozwolony rozkaz lub sięgnąć po komórkę pamięci nie należącą do jego przestrzeni adresowej,

sowej, wpadnie wówczas w pułapkę zastawioną przez sprzęt, co oznacza przejście do systemu operacyjnego. Tak jak przerwanie, pułapka powoduje przejście do systemu operacyjnego za pomocą wektora przerwań. Za każdym razem, gdy wystąpi błąd w programie, system operacyjny wymusza nienormalne zakończenie programu. Zdarzenie takie jest obsługiwane za pomocą tego samego kodu co żądanie nienormalnego zakończenia programu pochodzące od użytkownika. Pojawia się odpowiedni komunikat o błędzie, po czym następuje składowanie pamięci programu. Obraz pamięci programu jest zazwyczaj zapisywany w pliku, użytkownik może go więc przeanalizować i, po ewentualnej poprawce, spróbować uruchomić program od nowa.

2.5.1 Dualny tryb operacji

Aby zapewnić poprawną pracę, musimy chronić system operacyjny i wszystkie inne programy oraz ich dane przed każdym niewłaściwie działającym programem. Ochroną muszą być objęte wszystkie wspólnie wykorzystywane zasoby. Ta metoda postępowania polega na zaopatrzeniu sprzętu w środki pozwalające na rozróżnianie różmaitych trybów jego pracy. Potrzebujemy rozróżniania co najmniej dwu oddzielnych trybów pracy: *trybu użytkownika* (ang. *user mode*) i *trybu monitora* (ang. *monitor mode*), nazywanego także *trybem nadzorcy* (ang. *supervisor mode*), *trybem systemu* (ang. *system mode*) lub *trybem uprzywilejowanym* (ang. *privileged mode*). W sprzęcie komputerowym istnieje bit, zwany *bitem trybu* (ang. *mode bit*), którego stan wskazuje bieżący tryb pracy: monitor (0) albo użytkownik (1). Za pomocą bitu trybu można odróżnić działania wykonywane na zamówienie systemu operacyjnego od działań wykonywanych na zamówienie użytkownika. Jak zobaczymy, takie ulepszenie architektury sprzętu jest użyteczne ze względu na wiele innych aspektów działania systemu.

W czasie rozruchu systemu sprzęt rozpoczyna działanie w trybie monitora. Następuje załadowanie systemu operacyjnego, który uruchamia procesy użytkowe w trybie użytkownika. Za każdym razem po wystąpieniu pułapki lub przerwania sprzęt zmienia tryb pracy z trybu użytkownika na tryb monitora (tzn. zmienia wartość bitu trybu na 0). Tym samym, ilekroć system operacyjny przejmie sterowanie komputerem, tylekroć jest on w trybie monitora. Przed przejściem do programu użytkownika system zawsze przełącza tryb pracy na tryb użytkownika (ustawiając bit trybu na 1).

Dualny tryb działania komputera dostarcza środków do ochrony systemu operacyjnego przed nieodpowiedzialnymi użytkownikami, a także do chro- nienia nieodpowiedzialnych użytkowników wzajemnie przed sobą. Ochrona ta jest uzupełniana za pomocą oznaczenia potencjalnie niebezpiecznych rozkazów kodu maszynowego jako *rozkazów uprzywilejowanych* (ang. *privileged*

instructions). Sprzęt pozwala wykonywać rozkazy uprzywilejowane tylko w trybie monitora. Próba wykonania rozkazu uprzywilejowanego w trybie użytkownika nie zakończy się wykonaniem go przez sprzęt. Przeciwnie – rozkaz zostanie przez sprzęt potraktowany jako niedopuszczalny i spowoduje awaryjne przejście do systemu operacyjnego.

Brak sprzętowych środków do organizacji dualnego trybu pracy może powodować poważne następstwa w systemie operacyjnym. Na przykład system operacyjny MS-DOS napisano dla procesora Intel 8088, który nie ma bitu trybu, a więc i dwu trybów pracy. Niepoprawny przebieg wykonania programu użytkownika może spowodować zniszczenie systemu operacyjnego przez zapisanie jego kodu danymi. Jeśli zaś wiele programów pisały równocześnie na jednym urządzeniu wyjściowym, to mogłyby powstać bezsensowne wyniki. W późniejszym czasie ulepszone wersje jednostek centralnych Intel, takie jak 80486, zostały wyposażone w dualny tryb operacji. W rezultacie w nowszych systemach operacyjnych, takich jak Microsoft Windows NT i IBM OS/2, skorzystano z tej właściwości, dzięki czemu są one objęte lepszą ochroną.

2.5.2 Ochrona wejścia-wyjścia

Program użytkownika może zakłócić normalne działanie systemu, wydając niedozwolony rozkaz wejścia-wyjścia, docierając do komórek pamięci w obrębie samego systemu operacyjnego lub nie zwalniając procesora. Możemy zastosować różne mechanizmy nie dopuszczające do powstawania tego rodzaju zakłóceń w systemie.

Aby ustrzec użytkownika przed wykonywaniem niedozwolonych operacji wejścia-wyjścia, przyjęto, że wszystkie rozkazy wejścia-wyjścia są uprzywilejowane. Użytkownicy nie mogą wobec tego używać bezpośrednio tych rozkazów, lecz muszą to robić za pośrednictwem systemu operacyjnego. Aby uzyskać pełną ochronę wejścia-wyjścia, należy mieć pewność, że program użytkownika nigdy nie przejmie kontroli nad komputerem w trybie pracy monitora. Gdyby mu się to udało, ochrona zostałaby naruszona.

Rozważmy komputer, który pracuje w trybie użytkownika. Będzie on przechodzić w tryb monitora przy każdym wystąpieniu przerwania lub pułapki, wykonując skok pod adres określony w wektorze przerwań. Założymy, że program użytkownika umieściłby nowy adres w wektorze przerwań. Ten nowy adres mógłby zastąpić poprzedni adres i wskazać miejsce w programie użytkownika. Wówczas po wystąpieniu odpowiedniego przerwania sprzęt przełączyłby komputer w tryb monitora i przekazał sterowanie według (zmienionego) wektora przerwań do programu użytkownika! Program użytkownika przejąłby kontrolę nad komputerem w trybie monitora.

2.5.3 Ochrona pamięci

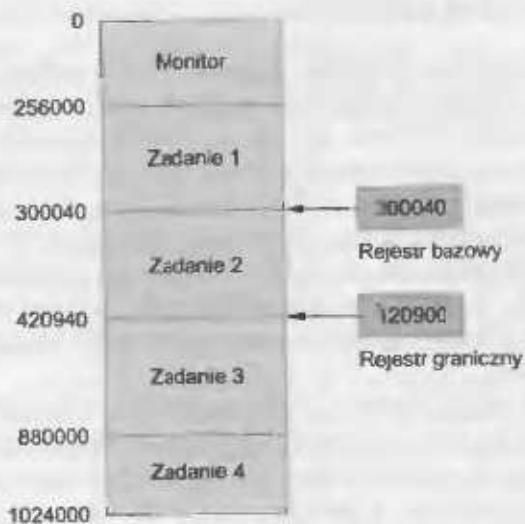
Aby zapewnić poprawne działanie, musimy chronić wektor przerwań przed zmianami, które mogłyby wprowadzić program użytkownika. Ponadto przed modyfikacjami należy również chronić systemowe procedury obsługi przerwań. W przeciwnym razie program użytkownika mógłby rozkazy w procedurze obsługi przerwania zastąpić skokami do własnego obszaru, przechwytyując sterowanie od procedury obsługi przerwania, pracującej w trybie monitora. Jeśli nawet użytkownik nie uzyskałby bezprawnej możliwości sterowania pracą komputera, to zmiany w procedurach obsługi przerwań zakłóciłyby prawdopodobnie właściwe działanie systemu komputerowego, przebieg spoolingu i buforowania.

Widzimy, że należy zapewnić ochronę pamięci przynajmniej w odniesieniu do wektora przerwań i systemowych procedur obsługi przerwań. Na ogół jednak zależy nam na ochronie całego systemu operacyjnego przed wpływami programów użytkowników, a ponadto – na wzajemnej ochronie programów użytkowników. Tę ochronę musi zapewniać sprzęt. Można ją zrealizować kilkoma sposobami, co zobaczymy w rozdziale 8. Teraz naszkicujemy tylko jedną z możliwych implementacji.

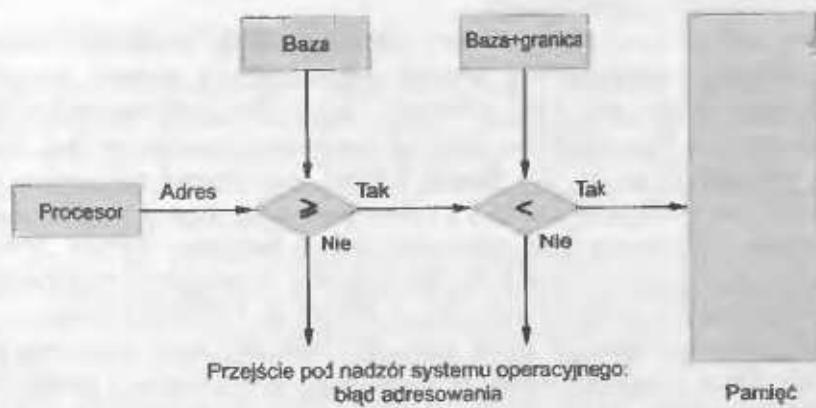
Aby oddzielić od siebie obszary pamięci każdego programu, musimy mieć możliwość rozstrzygania o zakresie dopuszczalnych adresów programu i chronienia pamięci poza tymi adresami. Tego rodzaju ochronę można uzyskać za pomocą dwóch rejestrów, zwanych *bazowym* i *granicznym* (ang. *base*, *limit*), pokazanych na rys. 2.7. Rejestr bazowy przechowuje najmniejszy dopuszczalny adres fizyczny pamięci, a rejestr graniczny zawiera rozmiar obszaru pamięci. Jeśli na przykład zawartość rejestru bazowego wynosi 300040, a rejestru granicznego – 120900, to jako poprawne w programie mogą wystąpić odniesienia do wszystkich adresów od 300040 do 420940 włącznie.

Ochronę taką sprawuje sprzęt jednostki centralnej przez porównywanie *każdego* adresu wygenerowanego w trybie pracy użytkownika z zawartością opisanych rejestrów. Jakiekolwiek usiłowanie programu pracującego w trybie użytkownika uzyskania dostępu do pamięci monitora lub programu innego użytkownika kończy się przejęciem do monitora, który traktuje taki zamiar jako poważny błąd (rys. 2.8). W takim schemacie program użytkownika jest chroniony przed (przypadkowym lub zamierzonym) zmodyfikowaniem kodu lub struktur danych systemu operacyjnego lub innych użytkowników.

Zawartości rejestrów – bazowego i granicznego – mogą być określone przez system operacyjny przy użyciu specjalnych, uprzywilejowanych rozkazów. Ponieważ rozkazy uprzywilejowane można wykonać tylko w trybie monitora, a jednocześnie tylko system operacyjny może pracować w tym trybie, więc jedynie system operacyjny może załadować rejestr bazowy i graniczny.



Rys. 2.7 Rejestr bazowy i rejestr graniczny definiują logiczną przestrzeń adresową



Rys. 2.8 Sprzętowa ochrona adresów z rejestrami – bazowym i granicznym

Schemat taki umożliwia monitorowi zmianie wartości tych rejestrów i nie pozwala zmieniać ich stanu przez programy użytkownika.

System operacyjny, działając w trybie monitora, ma nieograniczony dostęp zarówno do swojej pamięci, jak i do pamięci użytkowników. Dzięki temu system operacyjny może ładować programy użytkowników do przeznaczonych dla nich obszarów pamięci, w razie wystąpienia błędów może dokonywać składowania tych obszarów, ma dostęp do parametrów funkcji systemowych, które może modyfikować itd.

2.5.4 Ochrona jednostki centralnej

Ważnym elementem ochrony systemu przed zakłóceniami jest zapewnienie, że system operacyjny utrzymuje stałą kontrolę. Musimy zapobiec temu, żeby program użytkownika nie wpadł w nieskończoną pętlę, gdyż grozi to odebraniem sterowania systemowi operacyjnemu na zawsze. Osiąga się to przez zastosowanie czasomierza. *Czasomierz* (ang. *timer*) można ustawić tak, aby generował w komputerze przerwanie po wyznaczonym okresie. Okres ten może być stały (np. 1/60 s) lub zmienny (np. od 1 ms do 1 s, z przyrostami co 1 ms). Odmierzanie zmiennych okresów implementuje się za pomocą zegara stałookresowego i licznika. System operacyjny ustawia licznik. Przy każdym tyknięciu zegara następuje zmniejszenie licznika. Z chwilą wyzerowania licznika powstaje przerwanie. Dziesięcibitowy licznik z jednomilisekundowym zegarem umożliwia przerwania w odstępach od 1 do 1024 ms, z przyrostem co 1 ms.

Przed oddaniem sterowania do programu użytkownika system operacyjny dopilnowuje ustawienia czasomierza na przerwanie. Kiedy czasomierz powoduje przerwanie, wtedy sterowanie wraca automatycznie do systemu operacyjnego, który może uznać to przerwanie za poważny błąd lub zdecydować o przyznaniu programowi większej ilości czasu. Rozkazy modyfikujące działanie czasomierza są oczywiście zastrzeżone na użytk monitora.

W ten sposób czasomierz może być użyły do zapobiegania zbyt długiemu działaniu programu użytkownika. Proste postępowanie polega na zapamiętaniu w liczniku, ile czasu przydziela się programowi na wykonanie. Na przykład program z siedmiominutowym przydziałem czasu może mieć licznik zainicjowany na 420. Co sekundę czasomierz generuje przerwanie, a licznik zmniejsza się o 1. Dopóki licznik jest dodatni, dopóty sterowanie powraca do programu użytkownika. Gdy licznik stanie się ujemny, wówczas system operacyjny zakończy program z powodu przekroczenia przydzielonego limitu czasu.

Powszechniejsze zastosowanie czasomierza występuje w realizacji podziału czasu. W najprostszym przypadku czasomierz może być nastawiony na przerywanie co każde N ms, gdzie N jest *kwantem czasu* (ang. *time slice*) przydzielanym każdemu użytkownikowi na działanie, zanim następny użytkownik nie przejmie nadzoru nad procesorem. System operacyjny jest wywoływany po upływie każdego kwantu czasu w celu wykonania rozmaitych prac administracyjnych, jak dodanie wartości N do rekordu określającego (w celach rozliczeniowych) ilość czasu, którą program użytkownika zużył do tej pory. System operacyjny odświeża również stany rejestrów, zmiennych wewnętrznych i buforów oraz zmienia kilka innych parametrów, przy-

gotowując następnemu programowi pole do działania. (Procedura ta nosi nazwę *przełączania kontekstu* (ang. *context switch*); jest ona omówiona w rozdz. 4). Po zmianie kontekstu następny program kontynuuje swoją pracę od miejsca, w którym został przerwany (po wyczerpaniu ostatniego przydzielonego kwantu czasu).

Innym zastosowaniem czasomierza jest obliczanie bieżącego czasu. Przerwania czasomierza sygnalizują upłynięcie pewnej jednostki czasu, co pozwala systemowi operacyjnemu na obliczanie bieżącego czasu w odniesieniu do pewnej wartości początkowej. Jeśli przerwania następują co 1 s i zdarzyło się ich 1427, odkąd powiedzieliśmy, że jest pierwsza po południu, to można obliczyć, że obecnie jest godzina 13:23:47. Niektóre komputery wyznaczają bieżący czas w opisany sposób. Wskazywanie dokładnego czasu wymaga jednak starannych obliczeń, ponieważ czas przetwarzania przerwań (i czas wyłączeniowych przerwań) powoduje opóźnianie takiego programowego zegara. Większość komputerów ma oddzielnego, sprzętowego zegara czasu rzeczywistego, na który system operacyjny nie ma wpływu.

2.6 ■ Ogólna architektura systemu

Zapotrzebowanie na polepszanie wykorzystania systemu komputerowego doprowadziło do rozwoju wieloprogramowości i podziału czasu, w których te warunkach zasoby systemu komputerowego są dzielone między wiele różnych programów i procesów. Wspólne użytkowanie zasobów wpłynęło bezpośrednio na zmiany podstawowej architektury komputera, umożliwiające systemowi operacyjnemu nadzorowanie pracy systemu komputerowego, a zwłaszcza urządzeń wejścia-wyjścia. Nadzorowanie jest nieodzowne do tego, aby działanie komputera było ciągłe, spójne i poprawne.

Do sprawowania nadzoru konstruktory zastosowali dualny tryb pracy komputera (tryb użytkownika i tryb monitora). Schemat ten opiera się na koncepcji rozkazów uprzywilejowanych, których wykonywanie jest możliwe tylko w trybie monitora. Rozkazy wejścia-wyjścia, rozkazy zmieniające zawartość rejestrów zarządzania pamięcią lub rejestrów czasomierza są uprzywilejowane.

Jak można oczekiwać, do uprzywilejowanych zalicza się także kilka innych rozkazów. Uprzywilejowany jest na przykład rozkaz **halt** (zatrzymaj) – program użytkownika nigdy nie powinien zatrzymać komputera. Rozkazy włączania i wyłączania systemu przerwań są również uprzywilejowane, ponieważ właściwe działanie czasomierza i operacji wejścia-wyjścia zależy od możliwości poprawnego reagowania na przerwania. Uprzywilejowany jest rozkaz przejścia z trybu użytkownika do trybu monitora. Również jakiekol-

wiek zmienianie bitu trybu pracy wymaga w wielu maszynach uprzywilejowania.

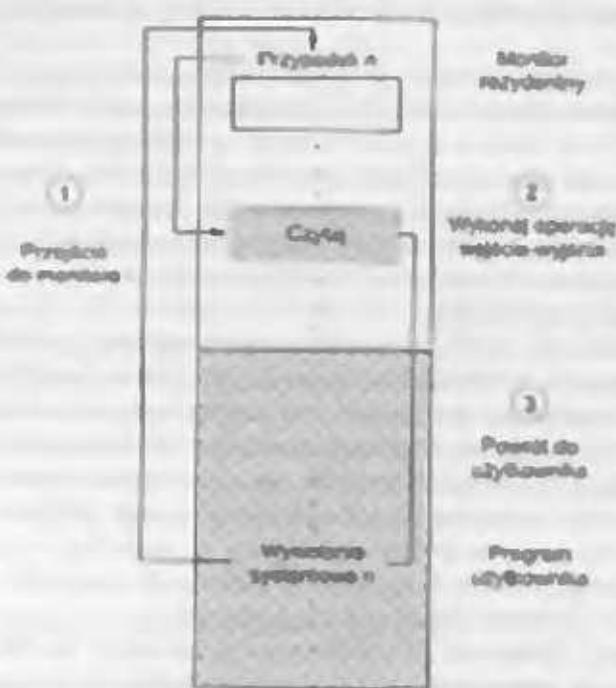
Rozkazy wejścia-wyjścia – jako uprzywilejowane – mogą być wykonywane tylko przez system operacyjny. Jak zatem program użytkownika może wykonać operację wejścia-wyjścia? Czyńiąc te rozkazy uprzywilejowanymi, odgradzamy programy użytkownika od wykonywania jakichkolwiek operacji wejścia-wyjścia – zarówno niedozwolonych, jak i dozwolonych. Problem ten rozwiązuje się w ten sposób, że skoro tylko monitor może wykonywać operacje wejścia-wyjścia, to użytkownik musi *poprosić* monitor, aby wykonał on taką operację w jego imieniu.

Prośba taka nosi nazwę *wywołania systemowego* (ang. *system call*), a bywa też nazywana *wywołaniem monitora* lub *wywołaniem funkcji systemu operacyjnego*^{*}. Wywołanie systemowe jest rozpoczynane na wiele sposobów, w zależności od właściwości danego procesora. We wszystkich odmianach jest to metoda, za pomocą której proces zamawia działanie systemu operacyjnego. Wywołanie systemowe zwykle przyjmuje postać przejścia do określonej komórki w wektorze przerwań. Przejście to może być wykonywane za pomocą ogólnego rozkazu trap, choć w niektórych systemach (np. rodzinny MIPS R2000) występuje specjalny rozkaz *syscall*.

Wywołanie systemowe jest traktowane przez sprzęt tak jak przerwanie programowe. Za pośrednictwem wektora przerwań sterowanie jest przekazywane do odpowiedniej procedury obsługi w systemie operacyjnym, a bit trybu zostaje przełączony w tryb monitora. Procedura obsługi wywołania systemowego jest częścią systemu operacyjnego. Monitor sprawdza rozkaz przerywający, aby określić, które wywołanie systemu miało miejsce. Rodzaj usługi, na którą użytkownik zgłasza zapotrzebowanie, jest określony przez parametr wywołania systemowego. Dodatkowe informacje potrzebne w związku z zamówieniem na usługę mogą być przekazane za pomocą rejestrów lub pamięci (za pomocą umieszczonego w rejestrach wskaźników do komórek pamięci). Monitor wykonuje zamówienie i przekazuje sterowanie do rozkazu, który następuje po wywołaniu systemowym.

Tak więc, aby wykonać operację wejścia-wyjścia, program użytkownika odwołuje się do systemu, powodując że system operacyjny wykona operację wejścia-wyjścia na jego życzenie (rys. 2.9). System operacyjny, pracujący w trybie monitora, sprawdza poprawność zamówienia i – jeśli jest ono dopuszczalne – wykonuje odpowiednią operację wejścia-wyjścia. Następnie system operacyjny przekazuje sterowanie do programu użytkownika.

* Dopuszczamy też jeszcze jeden synonim: *odwołanie do systemu*. — Przyp. tłum.



Rys. 2.9 Użycie rezydencji do celów wykonywania operacji wejścia-wyjścia

2.7 ■ Podsumowanie

Systemy wieloprogramowe oraz systemy «podziałem czasu» poprawią wydajność przez udogodnienie wykonywania w tym samym czasie na tej samej maszynie jednostki centralnej i operacji wejścia-wyjścia. Ta sama jednoznaczność działań wymaga, aby przesyłanie danych odbyły się z jednostką centralną a urządzeniem wejścia-wyjścia odbywało się za zasadnicze odpytowanie portu wejścia-wyjścia albo kontaktowanie się z nim przez odręgą przewozami bądź za pomocą rezydencji danych w trybie DMA.

Aby komputer mógł wykonywać programy, powinny się one znajdować w pamięci operacyjnej. Pamięć operacyjna (czyli główną) jest jedynym wielkim obiektem pamięci, do którego procesor ma dostęp bezpośredni. Pamięć operacyjna jest publiczna, tzn. wielkość wynosi od kilkuset tysięcy do kilkuset milionów słów lub bajtów. Każde słowo ma własny adres. Pamięć operacyjna jest pamięcią adresową, tzn. traci swoją zawartą po wyładowaniu zasilania lub w razie awarii. Wielkość systemów komputerowych jest zaopatrzona w pamięć permanentną, będącą rozszerzeniem pamięci operacyjnej. 1 M. pamięci

przeciwko której jest przede wszystkim trwalego przechowywanie wielu ilości danych. Niestandardizującym urządzeniem pamięci przewoźnej jest dysk magnetyczny, na którym można przechowywać zarówno programy, jak i dane. Dysk magnetyczny jest reprezentantem pamięci permanentnej (trwałej), a także umożliwia dostępu swobodnego. Tańce magnetycznych szuka się głównie do składowania, czyli przechowywania informacji: radiu i telewizji, oraz jako źródła przekazywanej informacji z jednego systemu do drugiego.

Dział rózmańdzenie systemów przetwarzania informacji w systemie komputerowym może być zorganizowane w hierarchię, niesione do ich szybkości i osoby. Przykłady wyższych poziomów są drogi, lecz wydajne. W mierze przechodzenia w dół hierarchii koszt przypadający na litr ma ogromne, natomiast z reguły zmniejsza się też czas dostępu.

System operacyjny musi gwarantować poprawne działanie systemu komputerowego. W celu zapobiedzenia niekorzystnego wpływu programów użytkownika system ma dwa tryby pracy użytkownika i monitora. Niektóre rozkazy (np. rozkaz wiersza-wyjścia albo rozkaz zatrzymania pracy komputera) są uprzywilejowane, tzn. mogą być wykonane tylko w trybie monitora. Pamięć, w której rezyduje system operacyjny, musi też być chroniona przed wprowadzaniem do niej danych przez użytkownika. Zasłonowanie ekranu pozwala zapobiegać wykorzystaniu rozkazów w niesłuchawnych sytuacjach. Te właściwości (dwa tryby pracy, rozkazy uprzywilejowane, ochrona pamięci, przerwanie od czasu do czasu) są podstawowymi elementami konstrukcyjnymi komputera przeznaczonego dla systemu operacyjnego do osiągnięcia poprawnego działania. W rozdziale 3 omówimy szczegółowo szczegółowe zagadnienia dotyczące trybów systemu operacyjnego.

■ Cwiczenia

- 2.1 Pobieranie z wyprzedzeniem (ang. prefetching) jest metodą umożliwiającą zasłodzenie na siebie operacji wejścia-wyjścia zadanego z reguły odświeżania. Pomyśl jest taki: Po zakończeniu operacji czytania, gdy dane nie zamierają przystąpić do działań na danych, urządzenie wejściowe otrzymuje poleceńce natychmiastowego rozpoczęcia następnej operacji czytania. Wywczai jednostka centralna i urządzenie wejściowe ta jednocześnie zajęte pracę. Przy udzieleniu zatroszczenia, czasem zadanie stanie się głośne do połkania następującej jednostki danych, urządzenie wejściowe zakończy jej czytanie. Procesu może wtedy podjąć przekształcanie nowego przekształconego danych, podczas gdy urządzenie wejściowe otrzymałyby kolejnych danych. Wtedyby pomyśl można zastanawiać na wyjściu. W tym przypadku zadanie tworzy dane, które są

umieszczane w buforze do chwili, gdy urządzenie wyjściowe będzie mogło je przyjąć.

Porównaj schemat pobierania z wyprzedzeniem ze schematem spoolingu, w którym działania jednostki centralnej wykonywane na rzecz jednego zadania przebiegają w tym samym czasie co obliczenia i wyprowadzanie wyników innych zadań.

- 2.2 W jaki sposób różnienie między trybem monitora a trybem użytkownika wpływa na elementarną ochronę (bezpieczeństwo) systemu?
- 2.3 Jakie są różnice między pułapką a przerwaniem? Jakie zastosowania znajduje każde z nich?
- 2.4 Do jakiego rodzaju działań przydaje się tryb DMA? Wyjaśnij swoją odpowiedź?
- 2.5 Które z poniższych rozkazów powinny być uprzywilejowane:
 - (a) określ wartość czasomierza;
 - (b) odczytaj stan zegara;
 - (c) zeruj pamięć;
 - (d) wyłącz wykrywanie przerwań;
 - (e) przełącz z trybu użytkownika w tryb monitora.
- 2.6 Niektóre systemy komputerowe nie dysponują sprzętowym trybem operacji uprzywilejowanych. Zastanów się, czy jest możliwe skonstruowanie dla takich komputerów bezpiecznego systemu operacyjnego? Podaj zarówno argumenty przemawiające na rzecz możliwości takiego przedsięwzięcia, jak i udowadniające, że jest to niemożliwe.
- 2.7 We wczesnych modelach komputerów system operacyjny był chroniony przez pozostawanie w obszarze pamięci, w którym nie były dozwolone żadne zmiany: ani ze strony użytkownika, ani ze strony systemu. Opisz dwie trudności, które – Twoim zdaniem – mogły powstawać przy takim rozwiązaniu?
- 2.8 Ochrona systemu operacyjnego jest ważna dla zagwarantowania po-prawnego działania systemu komputerowego. Zyski z takiej ochrony przemawiają za dualnym trybem pracy oraz sprzętową ochroną pamięci i procesora. Jednak jest także ważne, aby wyrikanające stąd ograniczenia były dla użytkownika jak najmniej uciążliwe.

Poniższa lista zawiera operacje, które zazwyczaj podlegają ochronie. Określ *minimalny* zbiór operacji, które muszą być chronione:

- (a) przejście do trybu użytkownika;
 - (b) przejście do trybu monitora;
 - (c) czytanie z pamięci monitora;
 - (d) zapisywanie w pamięci monitora;
 - (e) pobieranie rozkazów z pamięci monitora;
 - (f) włączanie czasomierza;
 - (g) wyłączenie czasomierza.
- 2.9 Gdzie znajdują zastosowanie pamięci podręczne? Jakie problemy pozwalają one rozwiązać? Jakich problemów są przyczyną? Gdyby można było wykonać pamięć podręczną tak dużą jak urządzenie, dla którego pełni funkcje pamięci podręcznej (np. pamięć podręczną tak dużą jak dysk), to dlaczego by tego nie zrobić, pozbywając się takiego urządzenia?
- 2.10 Opracowanie systemu operacyjnego odpornego na działanie błędnych bądź złośliwych programów wymaga pewnych środków sprzętowych. Wymień trzy elementy sprzętu pomocne przy opracowywaniu systemu operacyjnego i opisz, w jaki sposób mogłyby one być razem użyte do jego ochrony.

Uwagi bibliograficzne

Szczegółowe omówienie budowy urządzeń zewnętrznych, takich jak kanały i DMA, zawiera praca Baera [21]. Przegląd systemów wejścia-wyjścia i szyn oraz ogólnej architektury systemów podają Hennessy i Patterson [169]. Tanenbaum [415] opisuje architekturę mikrokomputerów, rozpoczynając od szczegółów poziomu sprzętowego.

Ogólne problemy wieloprzetwarzania opisują Jones i Schwarz [201]. Sprzęt wieloprocesorowy został opisany przez Satyanarayananana [373]. Działanie systemów wieloprocesorowych omówili: Maples [270], Sanguinetti [370], Agrawal i in. [3], a także Bhuyan i in. [36]. Przegląd architektur komputerów równoległych jest przedstawiony w pracy Duncana [122].

Omówienie technologii dysków magnetycznych opracowali Freedman [141] oraz Harker i in. [164]. Dyski optyczne opisali: Kenville [206], Fujitani [143], O'Leary i Kitts [313], Gait [144] oraz Olsen i Kenly [316]. Omówienie dysków elastycznych oferują Pechura i Schoeffler [328] oraz Sarisky [372].

Pamięci podręczne, w tym pamięć asocjacyjną, opisał i przeanalizował Smith [395]. Artykuł ten zawiera też szeroką bibliografię tematu. Hennessy i Patterson [169] omawiają sprzętowe aspekty urządzeń TLB, pamięci podręcznych i jednostek zarządzania pamięcią (MMU).

Ogólny opis technologii pamięci masowych zaproponowali: Chi [72] i Hoagland [171].

Rozdział 3

STRUKTURY SYSTEMÓW OPERACYJNYCH

System operacyjny tworzy środowisko, w którym są wykonywane programy. Pod względem organizacji wewnętrznej, systemy operacyjne znacznie się różnią. Zaprojektowanie nowego systemu operacyjnego stanowi poważne przedsięwzięcie. Jest istotne, aby przed rozpoczęciem projektowania cele systemu zostały dobrze określone. Od wymaganego typu systemu zależy wybór niezbędnych algorytmów i strategii.

Istnieje kilka dogodnych sposobów patrzenia na system operacyjny. Jednym z nich jest przegląd świadczonych przez system usług. Inny polega na zapoznaniu się z interfejsem, który system udostępnia użytkownikom i programistom. Trzeci sposób wymaga rozłożenia systemu na części i zbadania ich wzajemnych połączeń. W niniejszym rozdziale analizujemy wszystkie te trzy aspekty, aby pokazać systemy operacyjne z punktu widzenia ich użytkowników, programistów i projektantów. Rozważamy rodzaje usług dostarczanych przez system operacyjny, sposoby ich realizacji oraz różnorodne metody projektowania takich systemów.

3.1 ■ Składowe systemu

System tak wielki i złożony jak system operacyjny można utworzyć tylko wówczas, gdy podzieli się go na mniejsze części. Każda z takich części powinna być dobrze określonym fragmentem systemu ze starannie zdefiniowanym wejściem, wyjściem i działaniem. Jest oczywiste, że nie wszystkie systemy mają taką samą strukturę. Jednak wiele współczesnych systemów operacyjnych zawiera wspólne części składowe opisane w p. 3.1.1-3.1.8.

3.1.1 Zarządzanie procesami

Program nie robi nic dopóki, dopóki jego instrukcje nie są wykonywane przez jednostkę centralną. Za *proces* można uważać program, który jest wykonywany. Definicja ta zostanie poszerzona; zajmiemy się tym później. W myśl tego – zadanie wsadowe jest procesem. Program użytkownika wykonywany w systemie z podziałem czasu też jest procesem. Procesem jest również zadanie systemowe, takie jak buforowanie wyjścia na drukarkę. Na razie można przyjąć, że procesem jest zadanie lub program pracujący w systemie z podziałem czasu, choć w istocie pojęcie procesu jest o wiele ogólniejsze. W rozdziale 4 przekonamy się, że istnieją funkcje systemowe, które pozwalają procesom tworzyć wykonywane z nimi współbieżnie podprocesy.

Aby wypełnić swoje zadanie, proces musi korzystać z pewnych zasobów, takich jak czas jednostki centralnej, pamięć, pliki i urządzenia wejścia-wyjścia. Zasoby te proces otrzymuje w chwili jego utworzenia albo są one przydzielane procesowi podczas jego działania. Jako uzupełnieniem rozmaitych zasobów fizycznych i logicznych, które proces dostaje w chwili powstania, może on również otrzymać pewne dane początkowe (wejściowe). Rozważmy na przykład proces, który ma za zadanie wyświetlić na ekranie terminalu stan pliku. Proces taki otrzyma na wejściu nazwę pliku, wykona odpowiednie instrukcje i funkcje systemowe w celu uzyskania wymaganych informacji i wyświetli plik na terminalu. Po zakończeniu procesu system operacyjny odzyska wszelkie zasoby nadające się do powtórnego użytku.

Podkreślamy, że program sam w sobie nie jest procesem. Program jest elementem *parawanym*, tak jak zawartość pliku przechowywanego na dysku. Natomiast proces jest jednostką *aktywną*, której licznik rozkazów określa następną instrukcję do wykonania. Wykonanie procesu musi przebiegać w sposób sekwencyjny. Jednostka centralna wykonuje instrukcje procesu jedna po drugiej, aż do jego zakończenia. Ponadto w dowolnej chwili na zamówienie procesu jest wykonywana co najwyżej jedna instrukcja. Chociaż z jednym i tym samym programem mogą być związane dwa procesy, zawsze będzie się je rozważać jako dwa oddzielne ciągi wykonywanych instrukcji. Jest czymś zupełnie naturalnym, że jeden wykonywany program rodzi wiele procesów.

Proces jest jednostką pracy w systemie. W takim ujęciu system składa się ze zbioru procesów, z których część to procesy systemu operacyjnego (te, które wykonują kod systemu), pozostałe zaś są procesami użytkowymi (wykonującymi kod określony przez użytkownika). Wszystkie procesy potencjalnie mogą być wykonywane współbieżnie, na zasadzie multipleksowania między nimi jednostki centralnej.

W odniesieniu do zarządzania procesami system operacyjny odpowiada za następujące czynności:

- tworzenie i usuwanie zarówno procesów użytkowych, jak systemowych;
- wstrzymywanie i wznowianie procesów;
- dostarczanie mechanizmów synchronizacji procesów;
- dostarczanie mechanizmów komunikacji procesów;
- dostarczanie mechanizmów obsługi zakleszczeń.

Metody zarządzania procesami są szczegółowo omówione w rozdz. 4-7.

3.1.2 Zarządzanie pamięcią operacyjną

Dowiedzieliśmy się już w rozdz. 1, że pamięć operacyjna odgrywa główną rolę w działaniu współczesnego systemu komputerowego. Pamięć ta^{*} jest wielką tablicą słów lub bajtów, których liczba waha się w granicach od setek tysięcy do setek milionów. Każde słowo lub bajt ma przypisany adres. Pamięć stanowi magazyn szybko dostępnych danych eksploatowanych wspólnie przez jednostkę centralną i urządzenia wejścia-wyjścia. Podczas wykonywania cyklu pobierania rozkazów procesor czyta je z pamięci operacyjnej; także podczas wykonywania cyklu pobierania danych procesor czyta i zapisuje dane do tej pamięci. Operacje wejścia-wyjścia dokonywane metodą DMA również używają pamięci operacyjnej do odczytywania i zapisywania danych. Pamięć operacyjna jest w zasadzie jedyną pamięcią, którą jednostka centralna może adresować bezpośrednio. W przypadku innych rodzajów pamięci, na przykład pamięci dyskowej, aby przetwarzać znajdujące się w nich dane, procesor musi najpierw sprowadzić je do pamięci operacyjnej za pomocą odpowiednich operacji wejścia-wyjścia. Podobnie, aby jednostka centralna mogła wykonywać rozkazy, muszą one znajdować się w tej pamięci.

Aby program mógł być wykonany, musi być zaadresowany za pomocą adresów bezwzględnych oraz załadowany do pamięci. Podczas wykonywania programu rozkazy i dane są pobierane z pamięci za pomocą generowania tych właśnie adresów bezwzględnych. Kiedy program zakończy działanie, wtedy jego miejsce w pamięci jest oznaczane jako wolne, co umożliwia załadowanie i wykonanie następnego programu.

Jeśli chcemy uzyskać zarówno lepsze wykorzystanie jednostki centralnej, jak i szybszą reakcję komputera na polecenia jego użytkowników, to musimy przechowywać kilka programów w pamięci operacyjnej. Jest wiele różnych sposobów zarządzania pamięcią. Odzwierciedlają one różne podejścia do tego

* Zgodnie z oryginałem termin *pamięć* będzie zawsze oznaczać pamięć operacyjną.
– Przyp. tłum.

zagadnienia, przy czym efektywność poszczególnych algorytmów zależy od konkretnej sytuacji. Wybór sposobu zarządzania pamięcią zależy od wielu czynników, a zwłaszcza od rozwiązań sprzętowych zastosowanych w danym systemie. Każdy algorytm wymaga swoistego wspomagania sprzętowego.

W odniesieniu do zarządzania pamięcią system operacyjny odpowiada za:

- utrzymywanie ewidencji aktualnie zajętych części pamięci wraz z informacją, w czym są władani;
- decydowanie o tym, które procesy mają być załadowane do zwolnionych obszarów pamięci;
- przydzielanie i zwalnianie obszarów pamięci stosownie do potrzeb.

Metody zarządzania pamięcią są szczegółowo omówione w rozdz. 8 i 9.

3.1.3 Zarządzanie plikami

Zarządzanie plikami jest jedną z najbardziej widocznych części składowych systemu operacyjnego. Komputery mogą przechowywać informację na nośnikach fizycznych kilku różnych typów. Najbardziej rozpowszechnione są taśmy i dyski magnetyczne oraz dyski optyczne. Każdy z tych nośników ma charakterystyczne parametry techniczne i swoją organizację fizyczną. Kontrolę nad każdym z nośników sprawuje urządzenie, takie jak sterownik dysku lub sterownik taśmy, mające specyficzne cechy. Są to: szybkość działania, pojemność, szybkość przesyłania danych oraz metoda dostępu (dostęp sekwencyjny lub swobodny).

Dla wygody użytkowania systemu komputerowego system operacyjny tworzy jednolity, logiczny obraz magazynowanej informacji. System operacyjny definiuje *pliki*, czyli jednostki logiczne przechowywanej informacji, niezależne od fizycznych właściwości używanych urządzeń pamięci. System operacyjny odwzorowuje pliki na fizyczne nośniki informacji i umożliwia do nich dostęp za pomocą urządzeń pamięci.

Plik jest zbiorem powiązanych ze sobą informacji zdefiniowanych przez jego twórcę. W plikach zazwyczaj przechowuje się programy (zarówno w postaci źródłowej, jak i wynikowej) oraz dane. Pliki danych mogą być liczbowe, tekstowe lub alfanumeryczne. Format plików może być swobodny, jak w przypadku plików tekstowych, lub ścisłe określony. Plik zawiera ciąg bitów, bajtów, wierszy lub rekordów, których znaczenie zależy od ich twórców. Pojęcie pliku jest bardzo ogólne.

System operacyjny realizuje abstrakcyjny model plików przez zarządzanie nośnikami pamięci masowych, takimi jak taśmy lub dyski, oraz nadzoru-

jacymi je urządzeniami. Pliki są zazwyczaj zorganizowane w katalogi, co ułatwia ich użytkowanie. Poza tym, jeśli wielu użytkowników ma dostęp do plików, to może być pożądane sprawowanie pieczy nad tym, kto i w jaki sposób korzysta z tego dostępu.

W odniesieniu do zarządzania plikami system operacyjny odpowiada za:

- tworzenie i usuwanie plików;
- tworzenie i usuwanie katalogów;
- dostarczanie elementarnych operacji do manipulowania plikami i katalogami;
- odwzorowywanie plików na obszary pamięci pomocniczej;
- składowanie plików na trwałych nośnikach pamięci (na których informacja nie zanika).

Metody zarządzania plikami są omówione w rozdz. 10 i 11.

3.1.4 Zarządzanie systemem wejścia-wyjścia

Jednym z celów systemu operacyjnego jest ukrywanie przed użytkownikiem szczegółów dotyczących specyfiki urządzeń sprzętowych. Na przykład w systemie UNIX osobliwości urządzeń wejścia-wyjścia są ukrywane przed większością samego systemu operacyjnego przez tzw. *podsystem wejścia-wyjścia*. Podsystem ten składa się z:

- części zarządzającej pamięcią, wliczając w to: buforowanie, pamięć podręczną i spooling;
- ogólnego interfejsu do modułów sterujących urządzeń;
- modułów sterujących (programów obsługi) poszczególnych urządzeń sprzętowych.

Osobliwości konkretnego urządzenia zna tylko odpowiadający mu moduł sterujący.

Omawialiśmy już w rozdz. 2 pewne szczegóły konstruowania wydajnych podsystemów wejścia-wyjścia za pomocą programów obsługi przewrań oraz modułów sterujących urządzeń. W rozdziale 12 opisujemy, w jaki sposób podsystem wejścia-wyjścia kontaktuje się z innymi częściami systemu, zarządzania urządzeniami, przesyła dane i wykrywa zakończenia operacji wejścia-wyjścia.

3.1.5 Zarządzanie pamięcią pomocniczą

Podstawowym zadaniem systemu komputerowego jest wykonywanie programów. Podczas wykonywania programy oraz używane przez nie dane muszą znajdować się w pamięci operacyjnej. Ponieważ jednak pamięć operacyjna jest za mała, aby pomieścić wszystkie dane i programy, a zawarte w niej dane giną po odcięciu zasilania, system komputerowy musi mieć pamięć pomocniczą, będącą zapleczem dla pamięci operacyjnej. Większość współczesnych systemów komputerowych posługuje się pamięcią dyskową jako podstawowym środkiem magazynowania zarówno danych, jak i programów. Większość programów – w tym kompilatory, asemblery, procedury sortujące, edytory i programy formatujące – do czasu załadowania do pamięci jest przechowywana na dysku oraz używa dysków jako źródeł i miejsc przeznaczenia przetwarzanych przez nie danych. Dlatego jest niezwykle ważne, aby zarządzanie pamięcią dyskową w systemie komputerowym odbywało się w sposób właściwy.

W odniesieniu do zarządzania dyskami system operacyjny odpowiada za:

- zarządzanie obszarami wolnymi;
- przydzielanie pamięci;
- planowanie przydziału obszarów pamięci dyskowej.

Ponieważ pamięć pomocnicza jest często używana, musi działać wydajnie. Ogólna szybkość działania komputera może zależeć od podsystemu dyskowego i stosowanych w nim algorytmów. Metody zarządzania pamięcią pomocniczą są szczegółowo omówione w rozdz. 13.

3.1.6 Praca sieciowa

System rozproszony jest zbiorem procesorów, które nie dzielą pamięci, urządzeń zewnętrznych ani zegara. W zamian każdy procesor ma własną, lokalną pamięć i komunikuje się z innymi procesorami za pomocą różnorodnych linii komunikacyjnych, takich jak szybkie szyny danych lub linie telefoniczne. Procesory w systemie rozproszonym są zróżnicowane pod względem wielkości i funkcji. Mogą być wśród nich małe mikroprocesory, stacje robocze, minikomputery oraz wielkie systemy komputerowe ogólnego przeznaczenia.

Procesory w systemie rozproszonym są połączone za pomocą sieci komunikacyjnej, która może być skonfigurowana na kilka różnych sposobów. Sieć może mieć połączenia pełne lub częściowe. Konstruując sieć komunikacyjną, trzeba uwzględniać trasy, strategie połączeń, a także problemy współprzednia i bezpieczeństwa.

System rozproszony gromadzi fizycznie oddzielne i być może różnorodne systemy w jeden spójny system, umożliwiając użytkownikowi dostęp do różnych utrzymywanych w nim zasobów. Dostęp do zasobów dzielonych pozwala na przyspieszanie obliczeń, zwiększenie osiągalności danych i podnoszenie niezawodności. W systemach operacyjnych dostęp sieciowy jest na ogół uogólniany pod postacią dostępu do plików, przy czym szczegóły komunikacji sieciowej są zawarte w programie obsługi urządzenia sprzągającego z siecią.

Sieci komputerowe i systemy rozproszone są omówione w rozdz. 15-18

3.1.7 System ochrony

Jeżeli system komputerowy ma wielu użytkowników i umożliwia współbieżne wykonywanie wielu procesów, to poszczególne procesy należy chronić przed wzajemnym oddziaływaniem. Muszą istnieć mechanizmy gwarantujące, że pliki, segmenty pamięci, procesor i inne zasoby będą użytkowane tylko przez te procesy, które zostały przez system operacyjny odpowiednio uprawnione.

Na przykład sprzęt adresujący pamięć gwarantuje, że proces będzie działał tylko w obrębie swojej przestrzeni adresowej. Czasomierz zapewnia, że żaden proces nie przejmie na stałe kontroli nad jednostką centralną. Nie zezwala się również użytkownikom na bezpośredni dostęp do rejestrów kontrolnych urządzeń, co chroni niezakłóconą pracę rozmaitych urządzeń zewnętrznych.

Ochrona jest mechanizmem nadzorowania dostępu programów, procesów lub użytkowników do zasobów zdefiniowanych przez system komputerowy. Mechanizm ten musi zawierać sposoby określania, co i jakie ma podlegać ochronie, jak również środki do wymuszenia zaprowadzonych ustaleń.

Z pomocą działań ochronnych polegających na poszukiwaniu błędów ukrytych w interfejsach między składowymi podsystemami można poprawić niezawodność systemu. Wczesne wykrywanie błędów w interfejsach może często zapobiec zanieczyszczeniu zdrowego podsystemu przez podsystem uszkodzony. Zasoby, które nie są chronione, nie mogą obronić się przed użyciem (lub nadużyciem) przez nieupoważnionego lub niekompetentnego użytkownika. System ochrony dostarcza środków do rozróżniania między prawomocnym i nieprawomocnym użyciem; omówimy to w rozdz. 19.

3.1.8 System interpretacji poleceń

Jednym z najważniejszych programów w systemie operacyjnym jest interpreter poleceń będący interfejsem między użytkownikiem a systemem operacyjnym. Niektóre systemy operacyjne zawierają interpreter poleceń w swoim jądrze. W innych systemach operacyjnych, takich jak MS-DOS i UNIX, interpreter poleceń jest specjalnym programem, wykonywanym przy rozpoczętaniu zadania lub wtedy, gdy użytkownik rejestruje się w systemie (z podziałem czasu).

Wiele poleceń jest przekazywanych do systemu operacyjnego za pomocą *instrukcji sterujących* (ang. *control statements*). W chwili rozpoczęcia nowego zadania w systemie wsadowym lub zarejestrowania się użytkownika w systemie konwersacyjnym, automatycznie zaczyna pracować program interpretujący instrukcje sterujące. Program ten bywa niekiedy nazywany *interpretatorem kart sterujących* lub *interpretatorem wiersza poleceń*, a często jest znany pod nazwą *powłoki* (ang. *shell*). Jego zadanie jest proste – ma pobrać następną instrukcję i ją wykonać.

Systemy operacyjne często różnią się warstwą powłoki. Przyjazny interpreter sprawia, że system jest milszy dla niektórych użytkowników. Rodzaj przyjaznego dla użytkownika interfejsu tworzą systemy Macintosh i Microsoft Windows, pozwalające korzystać z okien i menu za pomocą myszki. Manewrując myszką, można ustawać jej kursor na wyświetlanym na ekranie obrazkach (*ikonach*). Zależnie od położenia kurSORA myszki naciśnięcie odpowiedniego przycisku myszki może spowodować wywołanie programu, wybranie pliku lub katalogu (zwanego tu skoroszytem; ang. *folder*) lub rozwinięcie menu zawierającego polecenia. Inni użytkownicy wolą wyposażone w bogatsze możliwości, bardziej złożone i trudniejsze do opanowania powłoki. W niektórych z tych powłok polecenia pisane na klawiaturze są wyświetlane na ekranie monitora lub drukowane. Do zasygnalizowania, że polecenie jest kompletne i gotowe do wykonania, służy klawisz **Enter** (lub **Return**). Tak działają powłoki systemów MS-DOS i UNIX.

Polecenia rozpoznawane przez interpreter dotyczą: tworzenia procesów i zarządzania nimi, obsługi wejścia-wyjścia, administrowania pamięcią pomocniczą i operacyjną, dostępu do plików, ochrony i pracy sieciowej.

3.2 ■ Usługi systemu operacyjnego

System operacyjny tworzy środowisko, w którym są wykonywane programy. Dostarcza on pewnych usług zarówno programom, jak i użytkownikom tych programów. Poszczególne usługi różnią się między sobą – rzecz jasna – w różnych systemach operacyjnych, można jednak wyodrębnić spośród nich pewne wspólne klasy. Usługi te wprowadzono dla wygody programisty, aby ułatwić jego pracę nad programem.

- **Wykonanie programu:** System powinien móc załadować program do pamięci i rozpoczęć jego wykonywanie. Program powinien móc zakończyć swoją pracę w sposób normalny lub z przyczyn wyjątkowych (sygnalizując błąd).

- **Operacje wejścia-wyjścia:** Wykonywany program może potrzebować operacji wejścia-wyjścia odnoszących się do pliku lub jakiegoś urządzenia. Poszczególne urządzenia mogą wymagać specyficznych funkcji (jak zwijanie taśmy na przewijaku lub wyczyszczenie ekranu monitora). Ze względu na wydajność i ochronę użytkownicy zazwyczaj nie mogą bezpośrednio nadzorować operacji wejścia-wyjścia, środki do realizacji tych czynności musi więc oferować system operacyjny.
- **Manipulowanie systemem plików:** System plików ma znaczenie szczególne. Nie ulega wątpliwości, że programy muszą zapisywać i odczytywać pliki. Jest im również potrzebna możliwość tworzenia i usuwania plików przy użyciu ich nazw.
- **Komunikacja:** Istnieje wiele sytuacji, w których procesy wymagają wzajemnego kontaktu i wymiany informacji. Są dwie podstawowe metody organizowania takiej komunikacji. Pierwszą stosuje się w przypadku procesów działających w tym samym komputerze; drugą – przy komunikowaniu się procesów wykonywanych w różnych systemach komputerowych, powiązanych ze sobą za pomocą sieci komputerowej. Komunikacja może przebiegać za pomocą *pamięci dzielonej* lub przy użyciu techniki *przekazywania komunikatów*, w której pakiety z informacją są przemieszczane między procesami za pośrednictwem systemu operacyjnego.
- **Wykrywanie błędów:** System operacyjny powinien być nieustannie powiadamiany o występowaniu błędów. Błędy mogą się pojawiać w działaniu jednostki centralnej i pamięci (sprzętowa wada pamięci lub awaria zasilania), w urządzeniach wejścia-wyjścia (np. błąd parzystości na taśmie, awaria połączenia w sieci albo brak papieru w drukarce) lub w programie użytkownika (np. nadmiar arytmetyczny, próba sięgnięcia poza obszar pamięci programu lub przekroczenia przydzielonego czasu procesora). Na wszystkie rodzaje błędów system operacyjny powinien odpowiednio reagować, gwarantując poprawność i spójność obliczeń.

Istnieje jeszcze dodatkowy zbiór funkcji systemu operacyjnego, które nie są przeznaczone do pomagania użytkownikowi, lecz do optymalizacji działania samego systemu. Systemy pracujące dla wielu użytkowników mogą zyskiwać na efektywności dzięki podziałowi zasobów komputera pomiędzy użytkowników.

- **Przydzielanie zasobów:** Jeżeli wielu użytkowników i wiele zadań pracuje w tym samym czasie, to każdemu z nich muszą być przydzielane

zasoby. System operacyjny zarządza różnego rodzaju zasobami. Przydzielanie niektórych z nich (jak cykli procesora, pamięci operacyjnej oraz pamięci plików) wymaga odrębnego kodu, podczas gdy inne (jak urządzenia wejścia-wyjścia) mogą mieć znacznie ogólniejszy kod zamawiania i zwalniania. Na przykład do określenia najlepszego wykorzystania jednostki centralnej służą systemowi operacyjnemu procedury planowania przydziału jednostki centralnej w zależności od szybkości procesora, zadań czekających na wykonanie, liczby dostępnych rejestrów i innych czynników. Mogą też istnieć procedury przydzielania przewijaka taśmy dla zadania. Procedura tego rodzaju odnajduje wolny przewijak i odnotowuje w wewnętrznej tablicy, że przydzielono go nowemu użytkownikowi. Do wymazania tej informacji w tablicy przydziałów wywołuje się inną procedurę. Takie procedury mogą być również wywoływane do przydzielania ploterów, modemów i innych urządzeń zewnętrznych.

- **Rozliczanie:** Przechowywanie danych o tym, którzy użytkownicy i w jakim stopniu korzystają z poszczególnych zasobów komputera, jest kolejnym zadaniem systemu operacyjnego. Przechowywanie takich rekordów może służyć do rozliczania (aby można było wystawiać użytkownikom rachunki) lub po prostu do gromadzenia informacji w celach statystycznych. Statystyka użytkowania może stanowić cenne narzędzie dla badaczy chcących dokonywać rekonfiguracji systemu w celu polepszania jego usług obliczeniowych.
- **Ochrona:** Właściciele informacji przechowywanej w systemie skupiającym wielu użytkowników mogą chcieć kontrolować jej wykorzystanie. Gdy kilka oddzielnych procesów jest wykonywanych współbieżnie, wówczas żaden proces nie powinien zaburzać pracy innych procesów lub samego systemu operacyjnego. Do zadań ochrony należy gwarantowanie nadzoru nad wszystkimi dostępami do zasobów systemu. Nienajważniejsze jest *zabezpieczenie* systemu przed niepożdanymi czynnikami zewnętrznymi. Zabezpieczenia tego rodzaju polegają przede wszystkim na tym, że każdy użytkownik, aby uzyskać dostęp do zasobów, musi uwierzytelnić w systemie swoją tożsamość, na ogół za pomocą hasła. Dalszym rozszerzeniem zabezpieczeń jest obrona zewnętrznie zlokalizowanych urządzeń wejścia-wyjścia, w tym modemów i adapterów sieciowych, przed usiłowaniami niedozwolonego dostępu i rejestrowanie wszystkich takich przypadków w celu wykrywania włamania. Jeżeli system ma być chroniony i bezpieczny, środki zapobiegawcze muszą na wskroś go przenikać. Łańcuch jest tylko tylek taki wytrzymały, jak jego najsłabsze ogniwo.

3.3 ■ Funkcje systemowe

Funkcje systemowe^{*} tworzą interfejs między wykonywanym programem a systemem operacyjnym. Ogólnie biorąc, można z nich korzystać za pomocą rozkazów w języku asemblera. Ich wykazy znajdują się na ogół w podręcznikach używanych przez programujących w języku asemblera.

Niektóre systemy umożliwiają wywoływanie funkcji systemowych bezpośrednio w programie napisanym w języku wyższego poziomu. W tym przypadku wywołania funkcji systemowych są podobne do wywołań funkcji standardowych lub podprogramów. Mogą powodować wywołanie specjalnej procedury, która podczas wykonywania programu wykonuje funkcje systemowe, albo też funkcje systemowe mogą być dodawane bezpośrednio do wykonywanego programu.

Kilka języków, na przykład język C, Bliss, PL/360 i PERL, utworzono po to, aby nimi zastąpić języki asemblerowe przy programowaniu systemów operacyjnych. Języki te umożliwiają bezpośrednie wykonywanie wywołań funkcji systemowych. Niektóre implementacje Pascala również pozwalają wywoływać funkcje systemowe wprost z programu napisanego w tym języku. Realizacje języków C i PERL zawierają bezpośredni dostęp do funkcji systemowych. W podobne możliwości jest wyposażona większość implementacji Fortranu, często na zasadzie procedur bibliotecznych.

Aby zilustrować sposób używania funkcji systemowych, rozważmy opracowanie prostego programu czytającego dane z jednego pliku i kopiującego je do innego pliku. Pierwszymi danymi wejściowymi, których taki program będzie potrzebował, są nazwy obu plików – wejściowego i wyjściowego. W zależności od specyfiki systemu operacyjnego nazwy te można określić na wiele sposobów. Program może na przykład zapytać użytkownika o nazwy plików. W systemie interakcyjnym będzie to wymagało ciągu odwołań do systemu, najpierw w celu wypisania na ekranie symbolu zięblety do pisania, potem w celu czytania z klawiatury znaków definiujących oba pliki. Innym sposobem, stosowanym w systemach wsadowych, jest określenie nazw plików za pomocą instrukcji sterujących. W tym przypadku musi istnieć mechanizm przekazania tych parametrów z instrukcji sterujących do wykonywanego programu. W systemach, w których można korzystać z myszki oraz ikon, menu z nazwami plików jest zwykle wyświetlane w oknie. Użytkownik może wtedy posłużyć się myszką do wybrania nazwy pliku źródłowego, po czym może otworzyć okno do określenia nazwy docelowej.

Po otrzymaniu nazw obu plików program musi otworzyć plik wejściowy i utworzyć plik wyjściowy. Każda z tych czynności wymaga wywołania od-

* Czyli wywołania systemowe (ang. *system calls*). – Przyp. tłum.

rębnej funkcji systemowej. Przy okazji wykonywania każdej z nich jest możliwe wystąpienie błędu. Kiedy program próbuje otworzyć plik wejściowy, może się okazać, że nie istnieje żaden plik o podanej nazwie lub dany plik jest chroniony przed dostępem. W tych przypadkach program powinien wydrukować komunikat na konsoli (jeszcze inny ciąg wywołań funkcji systemowych), po czym zakończyć działanie na warunkach wyjątkowych (kolejne wywołanie funkcji systemowej). Jeśli plik wejściowy istnieje, to należy utworzyć nowy plik wyjściowy. Może się okazać, że jest już plik o takiej samej nazwie. Ta sytuacja może pociągnąć zaniechanie pracy programu (wywołanie innej funkcji systemowej) lub konieczność usunięcia istniejącego pliku (wywołanie jeszcze innej funkcji systemowej). Inną możliwością – w systemie interakcyjnym – jest spytanie użytkownika (ciąg wywołań funkcji systemowych w celu wyprowadzenia komunikatu zachęty oraz odczytania odpowiedzi z terminalu), czy zastąpić istniejący plik, czy też zaniechać działania.

Teraz, gdy oba pliki są przygotowane, można rozpoczęć pętlę czytania z pliku wejściowego (wywołanie funkcji systemowej) i pisania do pliku wyjściowego (wywołanie innej funkcji systemowej). Każda operacja czytania i pisania musi przekazać informację o wyniku jej wykonania, uwzględniającą różne możliwe przyczyny błędów. Na wejściu program może wykryć wystąpienie końca pliku lub awarii sprzętowej podczas czytania danych (np. błędu parzystości). W czasie operacji pisania mogą pojawić się błędy zależne od urządzenia wyjściowego (brak miejsca na dysku, fizyczny koniec taśmy, brak papieru w drukarce itd.).

W końcu, po skopiowaniu całego pliku, program może zamknąć oba pliki (kolejne wywołanie funkcji systemowej), wypisać komunikat na konsoli (dalejsze wywołania funkcji systemowych), po czym skończyć działanie w normalny sposób (ostatnie wywołanie funkcji systemowej). Okazuje się więc, że programy mogą intensywnie używać systemu operacyjnego.

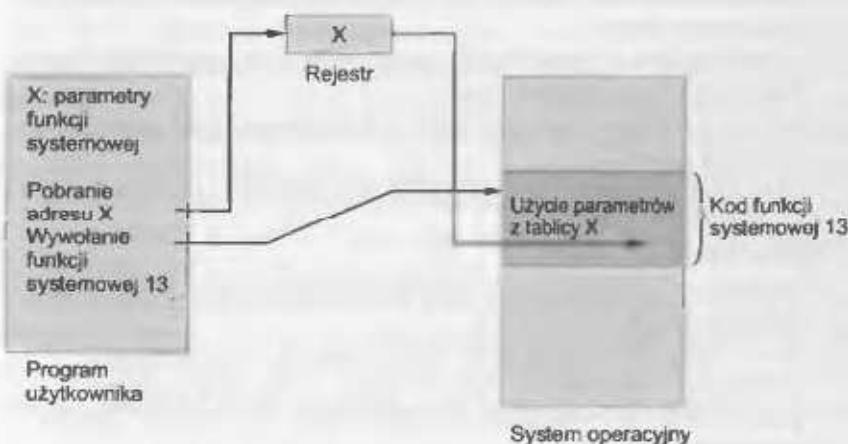
Jednakże większość użytkowników nigdy nie widzi tego rodzaju szczegółów. W większości języków programowania systemy wykonawcze programów tworzą znacznie prostsze interfejsy. Na przykład instrukcja *write* w Pascalu lub Fortranie prawdopodobnie zostanie przełożona na wywołanie procedury wspierającej wykonanie programu, która spowoduje wykonanie niezbędnych funkcji systemowych, sprawdzi, czy nie wystąpiły błędy, po czym powróci do programu użytkownika. W ten sposób dzięki kompilatorowi oraz jego systemowi wykonawczemu większość szczegółów interfejsu systemu operacyjnego pozostaje przed programistą ukryta.

Postać wywołań funkcji systemowych zależy od komputera. Zazwyczaj oprócz identyfikacji danej funkcji systemowej są potrzebne dodatkowe informacje. Dokładne określenie typu i ilości informacji zmienia się zależnie od systemu operacyjnego i wywoływanej funkcji. Na przykład, aby otrzymać

dane wejściowe, należy określić plik lub urządzenie, które zostanie użyte jako źródło, oraz adres i długość bufora w pamięci, do którego mają być przekazywane odczytywane dane. Oczywiście, urządzenie lub plik oraz długość bufora mogą być określone w wywołaniu niejawnie.

Istnieją zasadniczo trzy metody przekazywania parametrów do systemu operacyjnego. Najprostsza polega na umieszczeniu parametrów w *rejestrach* jednostki centralnej. W niektórych przypadkach może być jednak więcej parametrów niż rejestrów. Wówczas parametry przechowuje się w bloku lub tablicy w pamięci, adres zaś tego bloku przekazuje się jako parametr za pośrednictwem rejestru (rys. 3.1). Parametry można również przekazywać lub składać na stosie za pomocą programu, skąd będą zdejmowane przez system operacyjny. W pewnych systemach operacyjnych dano pierwszeństwo metodom bloków lub stosów, ponieważ nie ograniczają one liczby ani długości przekazywanych parametrów.

Funkcje systemowe można z grubsza podzielić na pięć podstawowych kategorii: *nadzorowanie procesów, operacje na plikach, operacje na urządzeniach, utrzymywanie informacji oraz komunikacja*. Poszczególne rodzaje funkcji systemowych, które mogą występować w systemie operacyjnym są pokrótkę omówione w p. 3.3.1-3.3.5. Niestety, nasz opis może wydać się całkiem powierzchowny, ponieważ większość z tych funkcji umożliwia działania lub jest realizowana sposobami, które opisujemy w następnych rozdziałach. Rysunek 3.2 zawiera zestawienie rodzajów funkcji systemowych zazwyczaj udostępnianych przez system operacyjny.



Rys. 3.1 Przekazywanie parametrów za pomocą tablicy

- Nadzorowanie procesów
 - zakończenie (*end*), zaniechanie (*abort*);
 - załadowanie (*load*), wykonanie (*execute*);
 - utworzenie procesu (*create process*), zakończenie procesu (*terminate process*);
 - pobranie atrybutów procesu (*get process attributes*), określenie atrybutów procesu (*set process attributes*);
 - czekanie czasowe (*wait for time*);
 - oczekiwanie na zdarzenie (*wait for event*), sygnaлизacja zdarzenia (*signal event*);
 - przydział i zwolnienie pamięci (*allocate and free memory*).
- Operacje na plikach
 - utworzenie pliku (*create file*), usunięcie pliku (*delete file*);
 - otwarcie (*open*), zamknięcie (*close*);
 - czytanie (*read*), pisanie (*write*), zmiana położenia (*reposition*);
 - pobranie atrybutów pliku (*get file attributes*), określenie atrybutów pliku (*set file attributes*).
- Operacje na urządzeniach
 - zamówienie urządzenia (*request device*), zwolnienie urządzenia (*release device*);
 - czytanie (*read*), pisanie (*write*), zmiana położenia (*reposition*);
 - pobranie atrybutów urządzenia (*get device attributes*), określenie atrybutów urządzenia (*set device attributes*);
 - logiczne przyłączanie lub odłączanie urządzeń (*logically attach or detach devices*).
- Utrzymywanie informacji
 - pobranie czasu lub daty (*get time or date*), określenie czasu lub daty (*set time or date*);
 - pobranie danych systemowych (*get system data*), określenie danych systemowych (*set system data*);
 - pobranie atrybutów procesu, pliku lub urządzenia (*get process, file, or device attributes*);
 - określenie atrybutów procesu, pliku lub urządzenia (*set process, file, or device attributes*).
- Komunikacja
 - utworzenie, usunięcie połączenia komunikacyjnego (*create, delete communication connection*);
 - nadawanie, odbieranie komunikatów (*send, receive messages*);
 - przekazanie informacji o stanie (*transfer status information*);
 - przyłączanie lub odłączanie urządzeń zdalnych (*attach or detach remote devices*).

Rys. 3.2 Rodzaje funkcji systemowych

3.3.1 Nadzorowanie procesów i zadań

Wykonywany program powinien móc zakończyć się w sposób normalny (funkcja `end`) lub wyjątkowy (funkcja `abort`). Jeżeli funkcja systemowa ma spowodować nagle zakończenie bieżącego programu lub jeśli działanie programu prowadzi do wykrycia błędu (ang. *error trap*), to jest niekiedy wykonywany zrzut zawartości pamięci, po czym jest generowana wiadomość o błędzie. Zawartość pamięci jest zapisywana na dysku, gdzie można ją przeanalizować za pomocą *programu diagnostycznego*^{*} (ang. *debugger*) w celu określenia przyczyny błędu. Niezależnie od tego, czy powstała sytuacja jest normalna, czy też nie, system operacyjny musi przekazać sterowanie do interpreta polecen, który czyta wówczas następne polecenie. W systemie interakcyjnym interpreter polecen po prostu wykonuje następne polecenie; zakłada się, że użytkownik wyda właściwe polecenie w odniesieniu do dowolnego błędu. W systemie wsadowym interpreter polecen musi zazwyczaj zaprzestać wykonywania całego zadania i przejść do wykonywania następnego zadania. Niektóre systemy pozwalają na stosowanie instrukcji sterujących ze wskaźówkami określającymi postępowanie w przypadku wystąpienia błędu. Jeśli program wykryje w danych wejściowych błąd zmuszający go do awaryjnego zatrzymania, to może również chcieć określić poziom błędu. Poważniejsze błędy można sygnalizować za pomocą większych wartości parametru poziomu błędu. Można bowiem ujednolicić zatrzymanie normalne i awaryjne w ten sposób, że zatrzymanie normalne definiuje się jako mające poziom błędu równy 0. Interpreter polecen lub następny wykonywany program mogą posłużyć się uzyskanym poziomem błędu do podjęcia automatycznej decyzji o dalszym postępowaniu.

Proces lub zadanie wykonujące jakiś program może wymagać *załadowania* (ang. *load*) i *wykonania* (ang. *execute*) innego programu. Interpreter polecen rozpoczyna wykonywanie takiego nowego programu zupełnie tak samo, jak gdyby zostało to nakazane przez polecenia użytkownika, naciśnięcie przycisku myszki lub wsadową instrukcję sterującą. Powstaje ciekawe pytanie, dokąd przekazać sterowanie wtedy, gdy tak załadowany program zakończy pracę. Pytanie to wiąże się z kwestią, czy stracimy wykonywany dotychczas program, czy będzie on przechowany, czy też pozwoli mu się kontynuować działanie współbieżnie z nowym programem.

Jeśli po zakończeniu nowego programu sterowanie ma powrócić do poprzedniego programu, to musimy przechować obraz pamięci pierwszego programu. Uzyskamy dzięki temu efektywny mechanizm wywoływania jednego programu z wnętrza innego programu. Natomiast gdy oba programy pracują

* Inaczej: *programu uruchomieniowego* – Przyp. tłum.

współbieżnie, wówczas otrzymujemy nowe zadanie lub proces, który trzeba uwzględnić w algorytmie wieloprogramowości. Służy do tego zazwyczaj specjalna funkcja systemowa (utworzenie procesu lub zadania).

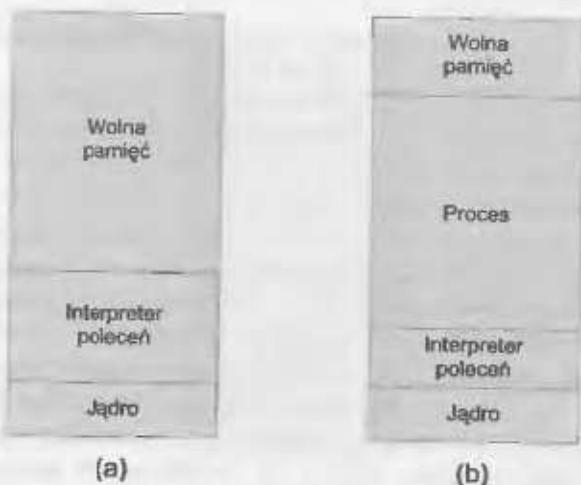
Jeśli utworzymy nowe zadanie lub proces – albo nawet zbiór zadań lub procesów – to powinniśmy móc wpływać na jego wykonanie. Wymaga to możliwości określania i ustalania wartości początkowych atrybutów zadania lub procesu, w tym priorytetu zadania, maksymalnego czasu przeznaczonego na jego wykonanie itd. (pobranie atrybutów procesu, określenie atrybutów procesu). Możemy również chcieć zakończyć wykonywanie utworzonego zadania lub procesu, jeśli okaże się, że są niepoprawne lub już niepotrzebne.

Po utworzeniu nowych zadań lub procesów możemy chcieć poczekać na ich zakończenie. Może to być czekanie przez określony czas (ang. *wait time*) lub, co bardziej prawdopodobne, na określone zdarzenie (ang. *wait event*). Zadania lub procesy powinny w związku z tym sygnalizować występowanie zdarzeń. Tego rodzaju funkcje systemowe koordynujące procesy współbieżne są bardziej szczegółowo omówione w rozdz. 6.

Odrębny zbiór funkcji systemowych jest pomocny przy sprawdzaniu poprawności programu. W wielu systemach są funkcje systemowe umożliwiające wykonywanie zrzutów zawartości pamięci (ang. *dump*). Jest to przydatne przy sprawdzaniu poprawności programów. Ślad działania programu (ang. *trace*) rejestruje ciąg instrukcji w kolejności ich wykonywania; możliwość taką ma mniej systemów. Nawet w mikroprocesorach istnieje tryb pracy jednostki centralnej, zwany jednokrokowym (ang. *single step*), w którym procesor po wykonaniu każdego rozkazu wchodzi w tryb obsługi specjalnej pułapki (ang. *trap*). Pułapka taka jest zwykle przechwytywana przez systemowy program diagnostyczny, pomagający programistę lokalizować i usuwać błędy.

Wiele systemów umożliwia oglądanie profilu czasowego programu. Profil czasowy ukazuje, ile czasu program spędza w poszczególnych komórkach lub grupach komórek. Sporządzenie profilu czasowego wymaga możliwości śledzenia programu lub regularnych przerwań zegarowych. Po każdym wystąpieniu przerwania zegarowego jest zapamiętywany stan licznika rozkazów programu. Przy odpowiednio częstych przerwaniach zegarowych można otrzymać statystyczny obraz czasu zużywanego przez różne części programu.

Sterowanie procesami i zadaniami ma wiele aspektów i odmian. Aby wyjaśnić te koncepcje, posłużymy się przykładami. System operacyjny MS-DOS jest przykładem systemu jednozadaniowego z interpreterem poleceń wywoływanym na początku pracy komputera (rys. 3.3(a)). Ponieważ MS-DOS jest jednozadaniowy, używa prostej metody wykonywania programu, nie powodującej tworzenia nowego procesu. System wprowadza program do pamięci operacyjnej, nawet kosztem większości własnego kodu, aby zapewnić mu możliwie jak największej przestrzeni (rys. 3.3(b)). Następnie



Rys. 3.3 Działanie systemu MS-DOS: (a) rozruch systemu; (b) wykonawanie programu

ustawia licznik rozkazów na pierwszy rozkaz programu. Program rozpoczyna działanie, w wyniku którego albo wystąpi błąd i uaktywni się odpowiednia pułapka systemowa, albo program zatrzyma się, wykonując odpowiednią funkcję systemową. W obu przypadkach kod błędu zostanie przechowany w pamięci systemu do późniejszego użytku. Po wykonaniu tych czynności wznowia działanie mały fragment interpretera poleceń, który nie został zniszczony przez kod programu. Najpierw powtórnie ładuje z dysku resztę swojego kodu, a po zakończeniu tej czynności – już jako pełny interpreter poleceń – udostępnia ostatni kod błędu użytkownikowi lub następnemu programowi.

Chociaż system operacyjny MS-DOS nie jest, ogólnie biorąc, wielozadaniowy, dostarcza sposobu na okrojone wykonywanie współbieżne. Program TSR „przechwytyuje przerwanie”, po czym kończy pracę za pomocą funkcji systemowej „zakończ i pozostań w pogotowiu” (ang. *terminate and stay resident*). Może on na przykład przechwytywać przerwania zegarowe wskutek umieszczenia adresu jednego z jego podprogramów na wykazie procedur obsługi przerwał wywoływanych po wyzerowaniu się systemowego czasomierza. W ten sposób procedura TSR będzie wykonywana wiele razy w ciągu sekundy, przy każdym impulsie zegarowym. Funkcja systemowa „zakończ i pozostań w pogotowiu” powoduje zarezerwowanie przez system MS-DOS obszaru na pomieszczenie programu TSR, dzięki czemu obszar ten nie będzie mógł być zapisany przez wprowadzany ponownie do pamięci interreter poleceń.

Wersja systemu UNIX opracowana w University of California w Berkeley jest przykładem systemu wielozadaniowego. Gdy użytkownik rejestruje się w systemie, wówczas wybiera odpowiednią do swych potrzeb *wersję interpretera polecen*.



Rys. 3.4 System UNIX wykonujący wiele programów

interpretatora poleceń (programu nazywanego *powłoką*; ang. *shell*). Program ten jest podobny do interpretatora poleceń systemu MS-DOS pod względem akceptowania poleceń i wykonywania programów na życzenie użytkownika. Jednak, ponieważ UNIX jest systemem wielozadaniowym, interpretator poleceń może kontynuować działanie podczas wykonywania innego programu (rys. 3.4). W celu zapoczątkowania nowego procesu program *shell* wykonyuje funkcję systemową *fork*. Wówczas, za pomocą funkcji systemowej *exec*, nowy program zostaje załadowany do pamięci i rozpoczyna pracę. W zależności od sposobu wydania polecenia program *shell* oczekuje na zakończenie działania procesu albo powoduje, że proces będzie wykonywany „w tle”^{*} (ang. *in the background*). W drugim przypadku program *shell* jest natychmiast gotowy do przyjmowania następnych poleceń. Proces, który jest wykonywany w tle, nie może otrzymywać danych z klawiatury, ponieważ zasobu tego używa powłoka. Toteż proces w tle wykonyuje operacje wejścia-wyjścia za pośrednictwem plików. Użytkownik ma prawo jednocześnie zlecić programowi *shell* uruchomienie innych programów, nadzorowanie przebiegu bieżącego procesu, zmianę priorytetu programu itd. Pod koniec pracy proces wywołuje funkcję systemową *exit*, która kończy jego działanie i powoduje przekazanie procesowi, który go wywołał, kodu stanu równego zero albo różnego od zera kodu błędu. Kod stanu lub błędu może później posłużyć programowi *shell* lub innym programom. Procesy są omówione w rozdz. 4.

* Czyli jako drugoplanowy. – Przyp. tłum.

3.3.2 Działania na plikach

System plików omówimy szczegółowo w rozdz. 10 i 11. Możemy jednak wyodrębnić kilka głównych funkcji systemowych dotyczących plików.

Przede wszystkim chcemy *tworzyć i usuwać* pliki (ang. *create, delete*). Każde wywołanie funkcji systemowej wymaga podania nazwy pliku i, być może, jakichś jego atrybutów. Aby móc użyć pliku po jego utworzeniu, trzeba go będzie najpierw *otworzyć* (ang. *open*). Potrzebne będą również operacje *czytania i pisania* (ang. *read, write*) oraz *zmiany punktu odniesienia* w pliku (ang. *reposition*) – na przykład przewijanie lub przeskakiwanie na koniec pliku. Do zaznaczenia, że plik nie będzie więcej używany, będzie na koniec wymagana operacja *zamknięcia* pliku (ang. *close*).

Jeśli system plików zawiera strukturę katalogów, to będzie potrzebny taki sam zbiór operacji dla katalogów. W dodatku, zarówno dla plików, jak i dla katalogów należy określić wartości różnych atrybutów, a nie wykluczone, że i nadawać im wartości początkowe. Do atrybutów pliku zalicza się nazwę pliku, typ pliku, kody zabezpieczające, informacje używane do rozliczania użytkownika itd. Potrzebne są do tego przynajmniej dwie funkcje systemowe: *pobrania atrybutu pliku* i *określenia atrybutu pliku*. Niektóre systemy operacyjne udostępniają w tym celu znacznie więcej funkcji.

3.3.3 Zarządzanie urządzeniami

Wykonywany program może potrzebować dodatkowych zasobów: większej pamięci, udostępnienia plików lub przewijaków taśm itd. Jeżeli te zasoby są dostępne, to będą mu przyznane i sterowanie powróci do programu użytkownika; w przeciwnym razie program musi czekać dopóty, dopóki nie będzie wystarczającej ilości zasobów.

Pliki można rozumieć jako abstrakcyjne lub wirtualne urządzenia. Wiele zatem funkcji systemowych dla plików jest również potrzebnych w przypadku urządzeń. Jeżeli system ma wielu użytkowników, to należy najpierw *poprosić o urządzenie*, aby zapewnić sobie wyłączność jego użytkowania. Po skończeniu użytkowania urządzenia należy je *zwolnić*. Te czynności są podobne do systemowego otwierania i zamykania plików.

Po założaniu przydziału urządzenia (i uzyskaniu go) do urządzenia można odnosić operacje czytania, pisania i (być może) zmiany położenia nośnika danych – jak w operacjach na zwykłych plikach. W istocie, podobieństwo między urządzeniami wejścia-wyjścia i plikami jest tak duże, że wiele systemów operacyjnych, w tym UNIX i MS-DOS, łączy je w jedną strukturę plików-urządzeń. Urządzenia wejścia-wyjścia są wówczas rozpoznawane jako pliki o specjalnych nazwach.

3.3.4 Utrzymywanie informacji

Wiele funkcji systemowych służy tylko do przesyłania informacji między programem użytkownika a systemem operacyjnym. Na przykład większość systemów ma funkcję systemową przekazującą bieżący czas i datę. Inne funkcje systemowe mogą przekazywać informacje o systemie, takie jak liczbę bieżących użytkowników, numer wersji systemu operacyjnego, ilość wolnej pamięci lub miejsca na dysku itd.

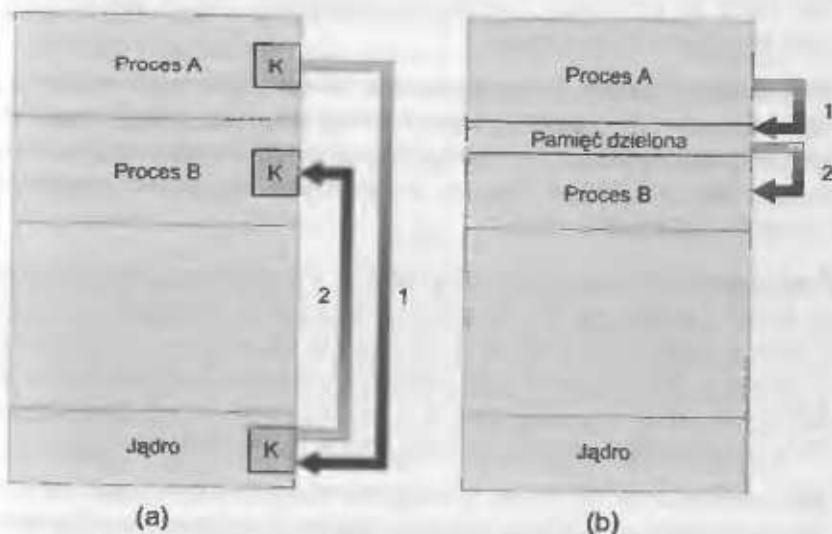
System operacyjny przechowuje ponadto informacje o wszystkich swoich procesach oraz zawiera funkcje systemowe udostępniające te informacje (pobranie atrybutów procesu). Mówiąc ogólnie, są także funkcje operujące na stanie informacji o procesach (określenie atrybutów procesu). Rodzaje informacji przechowywanych zazwyczaj w tych strukturach omówimy w p. 4.1.3.

3.3.5 Komunikacja

Są dwa główne modele komunikacji. W *modelu przesyłania komunikatów* (ang. *message-passing model*) informacja jest wymieniana przez międzyprocesowe środki komunikacji, które dostarcza system operacyjny. Przed rozpoczęciem komunikacji należy nawiązać połączenie. Musi być znana nazwa odbiorcy; może być nim inny proces w tej samej jednostce centralnej lub proces w innym komputerze. Każdy komputer w sieci ma swoją *nazwę sieciową* (ang. *host name*), która go identyfikuje. Podobnie, każdy proces ma swoją *nazwę procesu*, tłumaczoną na równoważny identyfikator, za pomocą którego system operacyjny odwołuje się do procesu. Odpowiednich tłumaczeń dokonują funkcje systemowe *pobrania nazwy sieciowej* (ang. *get hostid*) i *pobrania nazwy procesu* (ang. *get processid*). Uzyskane identyfikatory są potem parametrami ogólnych systemowych funkcji otwierania i zamknięcia plików lub specyficznych funkcji systemowych *otwierania połączenia* (ang. *open connection*) i *zamykania połączenia* (ang. *close connection*) – zależnie od systemowego modelu komunikacji. Proces odbiorcy musi zazwyczaj udzielić zgody na nawiązanie komunikacji za pomocą funkcji *akceptującej połączenie* (ang. *accept connection*). Większość procesów realizujących połączenia stanowią tzw. *demony* (ang. *daemons*), będące przeznaczonymi specjalnie do tego celu programami systemowymi. Wywolują one funkcję *czekania na połączenie* (ang. *wait for connection*) i są budzone wtedy, gdy połączenie zostanie nawiązane. Źródło komunikacji, nazywane *klientem*, i demon odbiorczy, nazywany *serwerem*, wymieniają wówczas komunikaty za pomocą funkcji systemowych *czytania komunikatu* (ang. *read message*) i *wysypania komunikatu* (ang. *write message*). Wywołanie funkcji *zamknięcia połączenia* kończy komunikację.

W modelu pamięci dzielonej (ang. *shared-memory model*) procesy posługują się systemowymi funkcjami odwzorowania pamięci (ang. *memory map*), aby uzyskać dostęp do obszarów pamięci należących do innych procesów. Zauważmy, że system operacyjny na ogół próbuje zapobiegać dostawianiu się jednego procesu do pamięci innego procesu. Dzielenie pamięci wymaga, aby dwa lub więcej procesów zgodoły się na usunięcie tego ograniczenia. W następstwie tego procesy mogą wymieniać informację przez czytanie i pisanie do wspólnie użytkowanych obszarów. Procesy określają postać danych i ich miejsce w pamięci; nie podlega to kontroli systemu operacyjnego. Procesy muszą także dopilnować, aby nie zapisywały jednocześnie tego samego miejsca. Odpowiednie mechanizmy są przedstawione w rozdz. 6.

Obydwa modele komunikacji występują w systemach operacyjnych, czasami nawet równocześnie. Przesyłanie komunikatów jest przydatne do bezkonfliktowej wymiany mniejszych ilości danych. Jest także łatwiejsze do zrealizowania w komunikacji międzykomputerowej niż metoda pamięci dzielonej. Pamięć dzielona zapewnia maksymalną szybkość i wygodę komunikacji, gdyż w obrębie jednego komputera komunikacja może przebiegać z szybkością działania pamięci operacyjnej. Metoda ta jest jednak obarczona problemami z zakresu ochrony i synchronizacji. Oba modele komunikacji są pokazane na rys. 3.5.



Rys. 3.5 Modele komunikacji: (a) przekazywanie komunikatów; (b) pamięć dzielona

3.4 ■ Programy systemowe

Kolejne zagadnienie dotyczące współczesnego systemu wiąże się z jego zbiorem programów systemowych. Powróćmy do rys. 1.1 obrazującego logiczną hierarchię systemu komputerowego. Na najniższym poziomie znajduje się oczywiście sprzęt. Wyżej lokuje się system operacyjny, następnie programy systemowe, potem dopiero programy użytkowe. Programy systemowe tworzą wygodniejsze środowisko do opracowywania i wykonywania innych programów. Niektóre z nich są po prostu interfejsami użytkownika do funkcji systemowych, podeszawszy gdy inne są stosunkowo bardziej złożone. Można je podzielić na kilka kategorii:

- **Manipulowanie plikami:** Są to programy, które służą do tworzenia, usuwania, kopowania, przemianowywania, składowania i wyprowadzania zawartości plików bądź katalogów.
- **Informowanie o stanie systemu:** Pewne programy po prostu pobierają z systemu dane określające datę, czas, ilość dostępnej pamięci lub miejsca na dysku, liczbę użytkowników lub podobne informacje o stanie systemu i komputera. Uzyskana informacja jest potem formatowana i drukowana na terminalu lub innym urządzeniu wyjściowym, względnie pośylana do pliku.
- **Tworzenie i zmienianie zawartości plików:** Rozmaite odmiany edytorów służą do tworzenia i zmieniania zawartości plików przechowywanych na dyskach lub taśmiech.
- **Translatory języków programowania:** Wraz z systemem operacyjnym użytkownikom dostarcza się często kompilatory, asemblyery oraz interpretory popularnych języków programowania (takich jak: Fortran, Cobol, Pascal, Basic, C, Lisp). Obecnie wiele z tych programów wycenia się i sprzedaje oddzielnie.
- **Ladowanie i wykonywanie programów:** Po przetłumaczeniu programu za pomocą asemblera lub kompilatora należy go wprowadzić do pamięci w celu wykonania. System może dostarczać programów ladowania kodu o adresach absolutnych i kodu przemieszczalnego, konsolidatorów oraz ladowaczy nakładek. Potrzebne są również systemy uruchomieniowe dla kodu w językach wyższego poziomu oraz w języku maszynowym.
- **Komunikacja:** W tej grupie znajdują się programy realizujące mechanizmy tworzenia wirtualnych połączeń między procesami, użytkownikami i różnymi systemami komputerowymi. Pozwalają one użytkownikom przesyłać komunikaty wyświetlane na ekranach ich odbiorców, wysyłać

większe komunikaty za pomocą poczty elektronicznej lub przesyłać pliki z jednej maszyny do drugiej, a nawet zdalnie korzystać z komputerów, tak jakby były to maszyny lokalne (jest to tzw. zdalne rejestrowanie – ang. *remote login*).

Większość systemów operacyjnych jest dostarczana wraz z programami do rozwiązywania typowych zadań lub wykonywania typowych działań. Do programów takich zalicza się: przeglądarki WWW, programy przetwarzania tekstu, arkusze kalkulacyjne, systemy baz danych, metatranslatory, czyli kompilatory kompilatorów (ang. *compiler compilers*), pakiety programów graficznych i statystycznych oraz gry.

Można zaryzykować ocenę, że najważniejszym programem systemowym jest *interpretator poleceń* (ang. *command interpreter*), którego głównym zadaniem jest pobieranie i wykonywanie kolejnych poleceń określanych przez użytkownika.

Wiele poleceń wykonywanych na tym poziomie dotyczy plików. Są to polecenia tworzenia, usuwania, wyprowadzania, drukowania, kopирования, wykonywania itd. Istnieją dwa podstawowe sposoby realizacji takich poleceń. Jeden polega na tym, że interpretator poleceń sam zawiera kod wykonujący polecenia. Na przykład polecenie usunięcia pliku może powodować skok interpretera do części jego własnego kodu, która obsługuje pobranie parametrów polecenia i wywoła odpowiednią funkcję systemową. Przy takim rozwiązaniu liczba możliwych poleceń wpływa na rozmiar interpretera, ponieważ każde polecenie wymaga odrębnego, realizującego je kodu.

Alternatywną metodę zastosowano między innymi w systemie UNIX. Wszystkie polecenia są wykonywane przez specjalne programy systemowe. W tym przypadku interpretator poleceń sam „nie rozumie” polecenia; służy mu ono jedynie do zidentyfikowania pliku, który ma być załadowany do pamięci i wykonany. W ten sposób polecenie

delete G

spowoduje odnalezienie pliku o nazwie *delete*, załadowanie go do pamięci i wykonanie z parametrem *G*. Funkcja związana z poleceniem *delete* jest w całości określona przez kod zawarty w pliku *delete*. W ten sposób programiści uzyskują łatwość dołączania nowych poleceń do systemu – wystarczy utworzyć nowy plik z odpowiednią nazwą. W takim potraktowaniu interpretator poleceń może być zupełnie niewielki i nie trzeba zmieniać go przy dodawaniu nowych poleceń.

Przy takim podejściu do projektowania interpretera poleceń musimy rozwiązać kilka problemów. Zauważmy, że skoro kod wykonujący polecenie jest osobnym programem, to system operacyjny musi zawierać mechanizm przekazywania parametrów od interpretera poleceń do programu systemowego.

Przedsięwzięcie to może okazać się kłopotliwe, ponieważ interpreter poleceń i program systemowy nie muszą przebywać w tym samym czasie w pamięci. Ponadto załadowanie i wykonanie programu trwa dłużej niż zwyczajny skok do innego miejsca kodu bieżąco wykonywanego programu.

Źródłem innego problemu jest pozostawienie interpretacji parametrów do dyspozycji autora programu systemowego. Może to prowadzić do niespojnych konwencji oznaczania parametrów w programach, które z punktu widzenia użytkownika są do siebie podobne, lecz zostały napisane w różnym czasie przez różnych programistów.

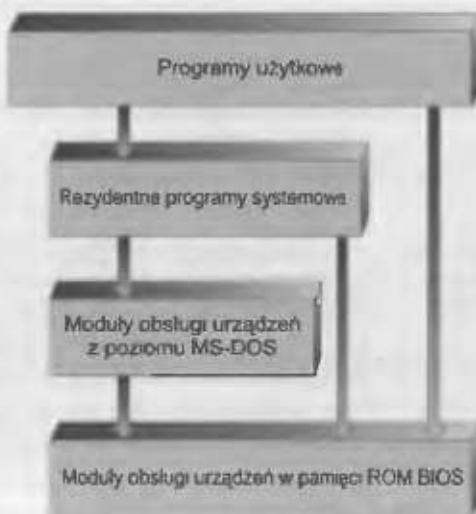
Sposób, w jaki większość użytkowników postrzega system operacyjny, jest zatem określany w większym stopniu przez programy systemowe arystotele przez funkcje systemowe. Rozważmy komputery osobiste: działając w systemie operacyjnym Microsoft Windows, użytkownik może oglądać powłokę poleceń MS-DOS albo interfejs graficzny z myszką i oknami. W każdym przypadku będzie w użyciu ten sam zbiór funkcji systemowych, lecz będą one wyglądały i działały odmiennie. W konsekwencji to, co widzi użytkownik, może być całkiem oddalone od rzeczywistej struktury systemu. Udostępnianie użytecznych i przyjaznych sposobów porozumiewania się z użytkownikiem nie należy przeto do bezpośrednich zadań systemu operacyjnego. W tej książce koncentrujemy się na podstawowych zagadnieniach związanych z zapewnieniem właściwej obsługi programów użytkowych. Spoglądając na rzeczy z pozycji systemu operacyjnego, nie wprowadzimy rozróżnienia między programami użytkownika a programami systemowymi.

3.5 ■ Struktura systemu

System tak wielki i złożony, jak współczesny system operacyjny, musi być konstruowany starannie, jeśli ma działać właściwie i dawać się łatwo modyfikować. Zamiast zajmować się jednym, monolitycznym systemem, powszechnie stosuje się podejście polegające na podziale tego przedsięwzięcia na małe części. Każda z takich części powinna być dobrze zdefiniowanym fragmentem systemu, ze starannie określonym wejściem, wyjściem i funkcjami. Omówiliśmy już zwięzłe główne składowe systemów operacyjnych (p. 3.1). Obecnie zajmiemy się sposobem łączenia ze sobą tych składowych, tak aby utworzyły jądro systemu.

3.5.1 Prosta struktura

Istnieje wiele komercyjnych systemów nie mających dobrze określonej struktury. Systemy te powstały najczęściej jako małe, proste i ograniczone systemy operacyjne, które następnie rozrastały się, przekraczając pierwotne

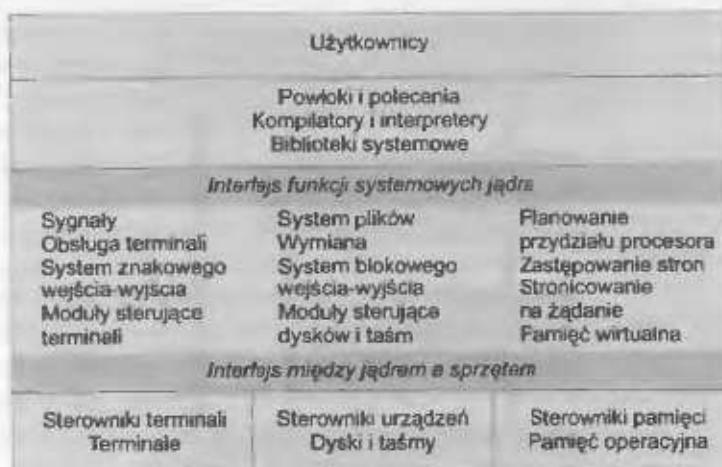


Rys. 3.6 Warstwowa struktura systemu MS-DOS

założenia. Przykładem takiego systemu jest MS-DOS. Został on początkowo zaprojektowany i zrealizowany przez niewielki zespół ludzi, którzy nie przypuszczali, że ich system stanie się tak popularny. Napisano go pod kątem osiągnięcia maksymalnej funkcjonalności przy oszczędności miejsca, gdyż sprzęt, na którym system ten miał działać, wymuszał takie ograniczenia. Nie zabrano więc o jego staranną modularyzację. Struktura systemu MS-DOS jest pokazana na rys. 3.6.

W systemie MS-DOS interfejsy i poziomy funkcjonalne nie są wyraźnie wydzielone. Na przykład programy użytkowe mogą korzystać z podstawowych procedur wejścia-wyjścia w celu bezpośredniego pisania na ekran lub dyski. Swoboda tego rodzaju powoduje, że MS-DOS nie jest odporny na błędne (lub złośliwe) programy użytkowe, które mogą doprowadzić do zatrzymania się całego systemu. Oczywiście możliwości systemu MS-DOS były również ograniczone przez sprzęt z jego epoki. Ponieważ mikroprocesor Intel 8088, dla którego system MS-DOS napisano, nie ma ani dualnego trybu pracy, ani ochrony sprzętowej, projektanci systemu nie mieli innej możliwości, niż udostępnić programistom podstawowe właściwości sprzętu.

Inny przykład ograniczonej strukturalizacji systemu stanowi oryginalny system operacyjny UNIX, który początkowo również był ograniczany przez cechy sprzętu. UNIX składa się z dwóch odrębnych części: jądra i programów systemowych. Jądro dzieli się dalej na ciąg interfejsów i programów obsługi urządzeń, które dodawano i rozszerzano przez lata rozbudowy systemu. Można powiedzieć, że system UNIX jest złożony z warstw, tak jak to widać na



Rys. 3.7 Struktura systemu UNIX

rys. 3.7. To wszystko, co znajduje się poniżej interfejsu funkcji systemowych a powyżej sprzętu, stanowi jądro. Za pośrednictwem funkcji systemowych jądro udostępnia system plików, planowanie przydzielu procesora, zarządzanie pamięcią i inne czynności systemu operacyjnego. Podsumowując, jest to bardzo wiele możliwości zebranych na jednym poziomie. Programy systemowe korzystają z udostępnianych przez jądro funkcji systemowych w celu wykonywania użytkowych działań, takich jak kompilowanie programów lub operowanie plikami.

Funkcje systemowe definiują *interfejs programisty* z systemem UNIX. Zbiór ogólnie dostępnych programów systemowych określa *interfejs użytkownika*. Oba interfejsy wyznaczają kontekst, który musi udostępniać jądro systemu. Opracowano kilka wersji systemu UNIX, w których jądro uległo dalszemu funkcjonalnemu podziałowi. W systemie AIX, wersja systemu UNIX firmy IBM, jądro dzieli się na dwie części. W systemie Mach z Carnegie-Mellon University jądro jest zredukowane do małego zbioru funkcji rdzeniowych, a wszystkie mniej ważne operacje są przeniesione do programów systemowych lub nawet programów z poziomu użytkownika. To, co pozostaje, jest systemem operacyjnym z mikrojądrkiem realizującym jedynie mały zbiór niezbędnych operacji elementarnych.

3.5.2 Podejście warstwowe

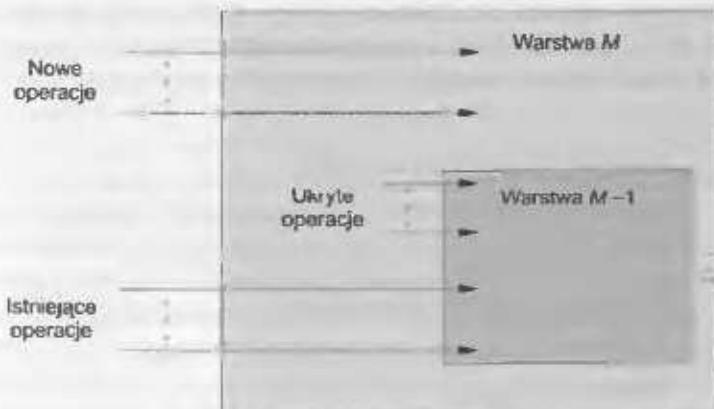
Nowe wersje systemu UNIX zaprojektowano z myślą o ulepszonym sprzęcie. Jeśli dysponuje się odpowiednim sprzętem, to można podzielić system operacyjny na mniejsze, lepiej dobrane fragmenty, niż było to możliwe w oryginal-

nych systemach MS-DOS lub UNIX. To z kolei sprawia, że system operacyjny ma znacznie większą kontrolę nad komputerem i korzystającymi z niego programami użytkowymi. Implementatorzy zyskują większą swobodę przy dokonywaniu zmian w wewnętrznym działaniu systemu. W tworzeniu modułarnych systemów operacyjnych pomagają z dawna ugruntowane techniki. Przy użyciu zstępującej metody projektowania można zdefiniować ogólne funkcje i cechy systemu oraz wyodrębnić jego części składowe. Cenna jest również zasada ukrywania informacji, gdyż pozostawia programistom swobodę w realizacji procedur niskopoziomowych, pod warunkiem, że zewnętrzne interfejsy z procedurami pozostały nie zmienione oraz że procedury będą wykonywały przewidziane zadania.

Modularyzację systemu można uzyskać na wiele sposobów, ale najbardziej obiecujące jest podejście warstwowe, polegające na dzieleniu systemu operacyjnego na warstwy (poziomy), przy czym każda następna warstwa jest zbudowana powyżej niższych warstw. Najniższą warstwę (warstwę 0) stanowi sprzęt; najwyższą (warstwę N) jest interfejs z użytkownikiem.

Warstwa systemu operacyjnego jest implementacją abstrakcyjnego obiektu, wyizolowanym zbiorem danych i operacji, które mogą danymi tymi manipulować. Typową warstwę systemu operacyjnego – powiedzmy warstwę M – widać na rys. 3.8. Zawiera ona struktury danych i procedury, które mogą być wywołane z wyższych warstw. Natomiast warstwa M może wywołać operacje dotyczące niższych warstw.

Główną zaletą podejścia warstwowego jest *modularność*. Warstwy są wybrane w ten sposób, że każda używa funkcji (operacji) i korzysta z usług tylko niżej położonych warstw. To podejście upraszcza wyszukiwanie błędów i weryfikację systemu. Pierwsza warstwa może być poprawiana bez żadnej



Rys. 3.8 Warstwa systemu operacyjnego

trośli o resztę systemu, ponieważ – z definicji – do realizacji swoich funkcji używa tylko podstawowego sprzętu (o którym zakłada się, że działa prawidłowo). Po uruchomieniu pierwszej warstwy, zakładając jej poprawne działanie, przystępuje się do opracowania drugiego poziomu itd. Jeśli podczas uruchamiania którejś warstwy zostanie wykryty błąd, to wiadomo, że musi on pochodzić z danej warstwy, ponieważ warstwy niższe już sprawdzono. W ten sposób podział systemu na warstwy upraszcza jego projektowanie i implementację.

Implementacja każdej warstwy opiera się wyłącznie na operacjach dostarczanych przez warstwy niższe. Warstwa nie musi nic wiedzieć o implementacji tych operacji; wie tylko, co operacje te robią. Każda warstwa ukrywa zatem istnienie pewnych struktur danych, operacji i sprzętu przed warstwami wyższych poziomów.

Po raz pierwszy warstwowe podejście w projektowaniu zostało zastosowane w systemie operacyjnym THE pochodzący z Technische Hogeschool w Eindhoven. System THE został zdefiniowany w sześciu warstwach, co widać na rys. 3.9. Dolną warstwę stanowił sprzęt. Następna warstwa implementowała planowanie przydziału procesora. Kolejna warstwa realizowała zarządzanie pamięcią. Schemat zarządzania pamięcią stanowiła pamięć wirtualna (rozdz. 9). Warstwa 3 zawierała program obsługi konsoli operatora. Ponieważ zarówno on, jak i buforowanie wejścia-wyjścia lokalizowane w warstwie 4 były powyżej zarządzania pamięcią, więc bufory urządzeń mogły być umieszczone w pamięci wirtualnej. Buforowanie wejścia-wyjścia znajdowało się również powyżej konsoli operatora, dzięki czemu informacje o błędach wejścia-wyjścia mogły być wyprowadzane na konsolę operatora.

Podejście warstwowe można stosować różnorodnie. Na przykład system Venus również zaprojektowano w sposób warstwowy. Niższe poziomy (0 do 4), przeznaczone do planowania przydziału procesora i do zarządzania pamięcią, zostały następnie napisane jako mikroprogramy. Taka decyzja przyniosła zysk w postaci szybszego działania i przejrzyste określonego interfejsu między warstwami mikroprogramowanymi a warstwami wyższymi (rys. 3.10).

Warstwa 5: programy użytkowe

Warstwa 4: buforowanie urządzeń wejścia i wyjścia

Warstwa 3: program obsługi konsoli operatora

Warstwa 2: zarządzanie pamięcią

Warstwa 1: planowanie przydziału procesora

Warstwa 0: sprzęt

Rys. 3.9 Struktura warstw systemu THE

Warstwa 6: programy użytkowe

Warstwa 5: programy obsługi i planowania przydziału urządzeń

Warstwa 4: pamięć wirtualna

Warstwa 3: kanał wejścia-wyjścia

Warstwa 2: planowanie przydziału procesora

Warstwa 1: interpreter instrukcji

Warstwa 0: sprzęt

Rys. 3.10 Struktura warstw systemu Venus

Główną trudnością w podejściu warstwowym jest odpowiednie zdefiniowanie poszczególnych warstw. Niezbędne jest staranne zaplanowanie podporządkowań wynikających z wymogu, że dana warstwa może używać tylko warstw niższych poziomów. Na przykład program obsługi pamięci pomocniczej (obszaru w pamięci dyskowej używanego przez algorytmy pamięci wirtualnej) powinien być na niższym poziomie niż procedury zarządzania pamięcią, ponieważ wymagają one możliwości korzystania z pamięci pomocniczej.

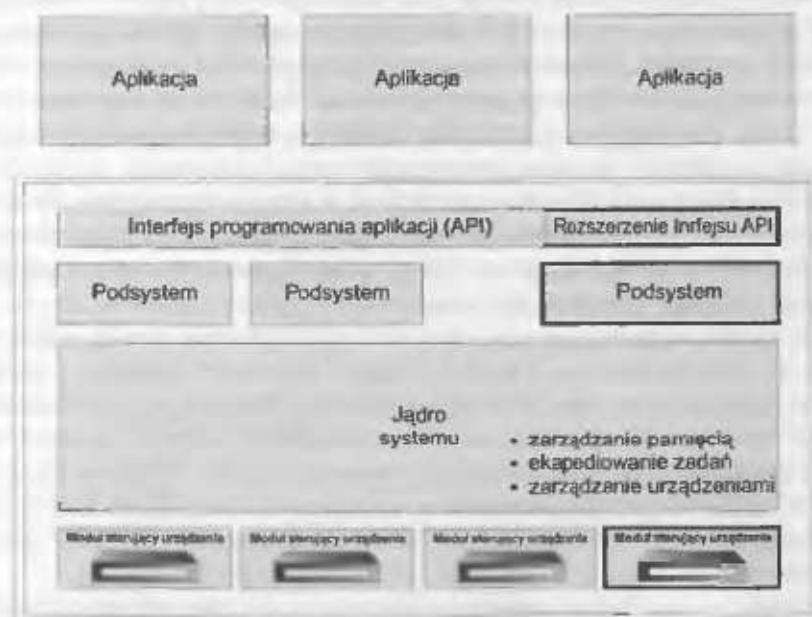
Inne wymagania mogą nie być już tak oczywiste. Program obsługi pamięci pomocniczej powinien być zazwyczaj powyżej programu planującego przydział procesora, ponieważ program obsługi pamięci pomocniczej może być zmuszony do czekania na operację wejścia-wyjścia, a w tym czasie procesor może otrzymać nowy przydział. Jednak w wielkim systemie program planujący przydział procesora może mieć więcej informacji o wszystkich aktywnych procesach, niż daje się pomieścić w pamięci operacyjnej. Powstaje więc potrzeba wymiany tych informacji między pamięciami, a to z kolei obliguje do umieszczenia programu obsługi pamięci pomocniczej poniżej programu planującego przydział procesora.

Na koniec wypada zauważyc, że realizacje warstwowe bywają mniej wydajne od innych. Jeśli na przykład program użytkowy wykonuje operację wejścia-wyjścia, to wywołuje funkcję systemową, która prowadzi do warstwy wejścia-wyjścia, a ta wywołuje warstwę zarządzania pamięcią i poprzez warstwę planowania przydziału procesora dociera do sprzętu. W każdej warstwie mogą występować zmiany parametrów, przenoszenie danych itd. Każda warstwa zwiększa koszt odwołania do systemu, zatem łącznie wykonanie funkcji systemowej trwa dłużej niż w systemie nie podzielonym na warstwy.

Ograniczenia tego rodzaju spowodowały, że w ostatnich latach zaczęto nieznacznie wycofywać się ze stosowania metod warstwowych. Projektuje się systemy złożone z niewielu, lecz bardziej funkcjonalnych warstw, zyskując większość zalet modułarnego programowania i unikając trudności

związkach z definiowanymi wzajemnymi zależnościami między warstwami. Na przykład w systemie OS/2 – następcy systemu MS-DOS – wprowadzono wielozadaniowość, podwójny tryb operacji i inne, nowe cechy. Ze względu na większą złożoność systemu i silniejszy sprzęt, dla którego OS/2 zaplanowano, system ten został zrealizowany z większym uwzględnieniem warstwości. Porównajmy strukturę systemu MS-DOS z tym, co jest przedstawione na rys. 3.11. Widać wyraźnie, że zarówno pod względem projektu, jak i implementacji, system OS/2 ma przewagę. Użytkownik nie może na przykład korzystać bezpośrednio z udoskonalen niskiego poziomu: umożliwia to systemowi operacyjnemu powiększenie kontroli nad sprzętem i lepsze rozeznanie co do zasobów eksploatowanych przez programy poszczególnych użytkowników.

Jako kolejny przykład rozważmy historię systemu Windows NT. Jego pierwsze wydanie miało wysoce warstwową organizację. Jednak wersja ta okazała się mało wydajna w porównaniu z systemem Windows 95. W wersji Windows NT 4.0 naprawiono niektóre z tych niedomagań, przesuwając warstwy z przestrzeni użytkownika do przestrzeni jądra i ścisłe je integrując ze sobą.



Rys. 3.11 Struktura warstw systemu OS/2

(Zaczerpnięty z Iacobucci *OS/2 Programmers Guide*, © 1988, McGraw-Hill, Inc., New York, New York, Fig. 1.7, p.20. Reprinted with permission of the publisher)

3.6 ■ Maszyny wirtualne

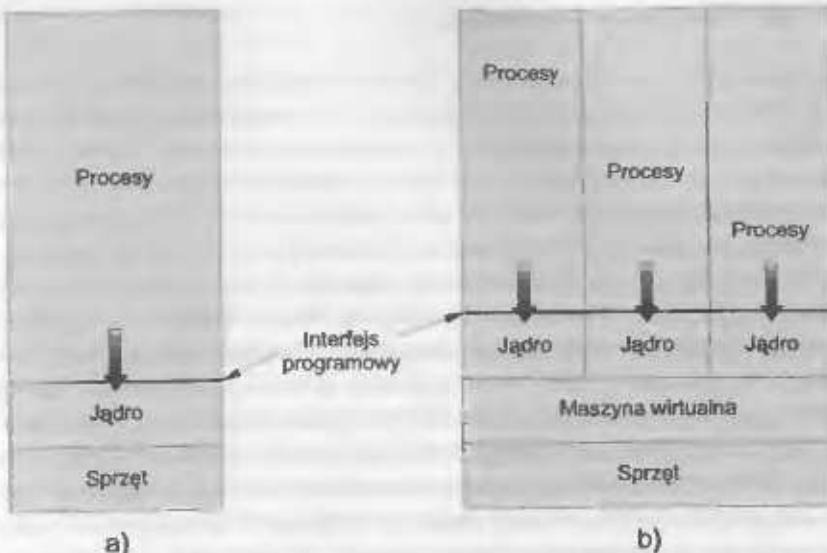
Z ogólnego punktu widzenia system komputerowy jest zbudowany z warstw. Sprzęt znajduje się na najniższym poziomie we wszystkich takich systemach. W jądrze systemu, pracującym na następnym poziomie, użyto rozkazów sprzętowych do utworzenia zbioru funkcji systemowych, z których korzysta się w zewnętrznych warstwach. W programach systemowych, występujących powyżej jądra systemu, można używać funkcji systemowych na równi z rozkazami sprzętowymi i, pod pewnymi względami, nie odróżniać funkcji od rozkazów. Chociaż oba rodzaje operacji są różnie osiągane, tak jedne, jak i drugie dostarczają środków pozwalających zaprogramować bardziej złożone funkcje. W związku z tym w programach systemowych traktuje się sprzęt i funkcje systemowe tak, jak gdyby należały one do tego samego poziomu.

Niektóre systemy przedłużają nawet ten schemat o następny krok, umożliwiając łatwe wywoływanie programów systemowych przez programy użytkowe. Tak jak poprzednio, chociaż programy systemowe są o szczebel wyżej niż inne procedury, programy użytkowe mogą to wszystko, co znajduje się w hierarchii poniżej nich, traktować jako stanowiące część maszyny. Logiczną konkluzją tak pojmowanej warstwości jest koncepcja *maszyny wirtualnej*. System operacyjny VM dla maszyn IBM jest najlepszym przykładem koncepcji maszyny wirtualnej, jako że firma IBM była pionierem prac na tym polu.

Stosując planowanie przydziału procesora (rozdz. 5) i technikę pamięci wirtualnej (rozdz. 9), system operacyjny może tworzyć złudzenie, że wiele procesów pracuje na własnych procesorach z własną (wirtualną) pamięcią. Oczywiście zazwyczaj proces taki ma dodatkowe możliwości, jak funkcje systemowe czy system plików, których nie dostarcza sam sprzęt. Z drugiej strony, dzięki koncepcji maszyny wirtualnej nie uzyskuje się żadnych dodatkowych funkcji, lecz tworzy jedynie interfejs *identyczny* z podstawowym sprzętem. Każdy proces otrzymuje (wirtualną) kopię komputera będącego podstawą systemu (rys. 3.12).

Zasoby fizycznego komputera są dzielone w celu utworzenia maszyn wirtualnych. Do podziału procesora i wywołania wrażenia, że każdy użytkownik ma własny procesor, można zastosować planowanie przydziału procesora. Spooling i system plików pozwalają utworzyć wirtualne czytniki kart i wirtualne drukarki wierszowe. Zwykły terminal do pracy z podziałem czasu przejmuje rolę wirtualnej konsoli operatorskiej.

Największa trudność związana z maszyną wirtualną dotyczy systemów dyskowych. Założymy, że fizyczna maszyna z trzema napędami dysków ma zasymułować siedem maszyn wirtualnych. Jest jasne, że nie można przydzieleć po jednym napędzie każdej maszynie wirtualnej. Należy pamiętać, że samo oprogramowanie maszyny wirtualnej zajmuje spory obszar dysku na obsługę



Rys. 3.12 Modele systemu: (a) maszyna niewirtualna; (b) maszyna wirtualna

pamięci wirtualnej i spoolingu. Rozwiązaniem jest zastosowanie dysków wirtualnych, które są pod każdym względem identyczne z fizycznymi z wyjątkiem rozmiaru. Tego rodzaju konstrukcji nadano w systemie operacyjnym VM dla maszyn IBM miano *minidysków*. System implementuje minidysk, przydzielając mu taką liczbę ścieżek dysku fizycznego, na jaką minidysk zgłosi zapotrzebowanie. Rzeczą oczywista, suma rozmiarów wszystkich minidysków musi być mniejsza od aktualnie dostępnej wielkości fizycznej przestrzeni dyskowej.

Użytkownicy otrzymują zatem własne maszyny wirtualne. Mogą w nich wykonywać dowolny system operacyjny lub pakiet oprogramowania dostępny w maszynie bazowej. W systemie IBM VM użytkownik na ogół korzysta z systemu CMS – interakcyjnego systemu operacyjnego dla indywidualnego użytkownika. Oprogramowanie maszyny wirtualnej zajmuje się wieloprogramową realizacją wielu maszyn wirtualnych w jednej maszynie fizycznej, lecz nie musi uwzględniać wspomagania oprogramowania użytkownika. Taka organizacja może wnieść w zagadnienie zaprojektowania wielostanowiskowego systemu interakcyjnego pozytyczny podział na dwa mniejsze fragmenty.

3.6.1 Realizacja

Koncepcja maszyny wirtualnej – przy całej swej użyteczności – jest trudna do implementacji. Zrealizowanie dokładnej kopii maszyny bazowej wymaga wielkiego wysiłku. Przypomnijmy, że fizyczna maszyna bazowa ma dwa

tryby pracy: tryb użytkownika i tryb monitora. Oprogramowanie maszyny wirtualnej może pracować w trybie monitora, ponieważ jest ono systemem operacyjnym. Natomiast sama maszyna wirtualna może działać tylko w trybie użytkownika. Niemniej jednak, skoro maszyna fizyczna ma dwa tryby, to maszyna wirtualna powinna mieć je również. W konsekwencji są potrzebne dwa wirtualne tryby – użytkownika i monitora, z których każdy pracuje w fizycznym trybie użytkownika. Czynności, które zmieniają tryb użytkownika na tryb monitora w rzeczywistej maszynie (jak wywołanie funkcji systemowej lub usiłowanie wykonania rozkazu uprzywilejowanego), muszą również powodować przejście w maszynie wirtualnej od wirtualnego trybu użytkownika do wirtualnego trybu monitora.

Na ogół to przejście może być wykonane dość łatwo. Kiedy na przykład program pracujący w maszynie wirtualnej w wirtualnym trybie użytkownika wywoła funkcję systemową, wtedy nastąpi przejście do wirtualnego monitora maszyny rzeczywistej. Kiedy monitor wirtualnej maszyny przejmie sterowanie, wtedy może zmienić zawartości rejestrów i licznika rozkazów maszyny wirtualnej, aby zasymułować efekt wywołania funkcji systemowej. Może on następnie wznowić działanie maszyny wirtualnej, zaznaczając, że jest ona teraz w trybie wirtualnego monitora. Jeśli maszyna wirtualna spróbuje wtedy na przykład czytać z jej wirtualnego czytnika kart, to wykona uprzywilejowany rozkaz wejścia-wyjścia. Ponieważ maszyna wirtualna pracuje w fizycznym trybie użytkownika, rozkaz ten spowoduje przejście do monitora maszyny wirtualnej. Monitor maszyny wirtualnej musi wówczas zasymułować wynik wykonania rozkazu wejścia-wyjścia. Najpierw odnajduje plik buforowy implementujący dany wirtualny czytnik kart. Następnie tłumaczy czytanie z wirtualnego czytnika kart na czytanie utożsamionego z nim buforowego pliku dyskowego, po czym przesyła kolejny wirtualny „obraz karty” do wirtualnej pamięci maszyny wirtualnej. Na koniec wznowia działanie maszyny wirtualnej. Stan maszyny wirtualnej uległ dokładnie takiej samej zmianie, jak gdyby wykonano rozkaz wejścia-wyjścia za pomocą prawdziwego czytnika kart w rzeczywistej maszynie pracującej w rzeczywistym trybie monitora.

Głównym czynnikiem rozróżniającym działanie maszyny wirtualnej i rzeczywistej jest czas. Podczas gdy rzeczywista transmisja mogłaby zabrać 100 µs, wirtualne operacje wejścia-wyjścia mogą trwać krócej (ponieważ są symułowane przy użyciu buforowego pliku dyskowego) albo dłużej (ponieważ są interpretowane). Ponadto procesor musi pracować wieloprogramowo dla wielu maszyn wirtualnych, co powoduje dalsze, nie dające się przewidzieć spowolnienie maszyny wirtualnej. W skrajnym przypadku, aby uzyskać prawdziwą maszynę wirtualną, może być konieczne symułowanie wszystkich rozkazów. System VM sprawdza się w działaniu na maszynach IBM, ponieważ zwykłe rozkazy dla jego maszyn wirtualnych mogą być wykonywane

wprost za pomocą sprzętu. Symulacja dotyczy tylko rozkazów uprzywilejowanych (potrzebnych głównie do wykonywania operacji wejścia-wyjścia), które z tego powodu działają wolniej.

3.6.2 Korzyści

Koncepcja maszyny wirtualnej ma kilka zalet. Zauważmy, że w tym środowisku istnieje pełna ochrona różnorodnych zasobów systemowych. Każda maszyna wirtualna jest całkowicie odizolowana od innych maszyn wirtualnych, nie ma więc kłopotów związanych z zapewnieniem bezpieczeństwa. Z drugiej strony nie ma bezpośredniej możliwości wspólnego użytkowania zasobów. Aby umożliwić dzielenie zasobów, zastosowano dwa podejścia. Po pierwsze – jest możliwe wspólne użytkowanie minidysku. Modeluje się je tak jak dla dysku fizycznego, a realizuje programowo. Ta technika umożliwia wspólne korzystanie z plików. Po drugie – można zdefiniować sieć maszyn wirtualnych, z których każda może wysyłać informacje przez wirtualną sieć komunikacyjną. Również i tu sieć jest wzorowana na fizycznych sieciach komunikacyjnych, ale jest realizowana przez oprogramowanie.

System maszyn wirtualnych stanowi znakomitą odszkocznę do badań nad systemami operacyjnymi i poszukiwań kierunków ich rozwoju. W normalnych warunkach zmiana systemu operacyjnego jest trudnym przedsięwzięciem. Ponieważ systemy operacyjne są wielkimi i złożonymi programami, nie można wykluczyć, iż zmiana w jednym miejscu nie spowoduje ukrytego błędu gdzie indziej. Sytuacja taka może być szczególnie niebezpieczna, zważywszy na możliwości systemu operacyjnego. Ponieważ system operacyjny pracuje w trybie monitora, złe ustawienie jakiegoś wskaźnika może okazać się błędem prowadzącym do zniszczenia całego systemu plików. Jest zatem niezbędne staranne testowanie każdej zmiany wprowadzonej do systemu.

Jednakże system operacyjny steruje pracą całej maszyny. Kiedy trzeba wprowadzić do niego zmiany i wykonać testy, wtedy bieżący system musi zostać zatrzymany i wyłączone z eksploatacji. Okres taki zwyczko się nazywać *czasem konserwacji systemu*. Ponieważ uniemożliwia on korzystanie z systemu przez użytkowników, planuje się go zwykle w późnych godzinach nocnych lub podczas weekendów, gdy obciążenie systemu jest małe.

System maszyn wirtualnych pozwala uniknąć większości takich kłopotów. Programiści systemowi dostają własną maszynę wirtualną i to na niej, zamiast na maszynie fizycznej, odbywa się konserwacja systemu. Konieczność zaburzenia normalnego działania systemu z powodu prowadzonych prac rozwijowych zdarza się rzadko.

Maszyny wirtualne stają się znów modne jako sposób rozwijywania zagadnień zgodności systemów. Istnieją na przykład tysiące programów dostęp-

nych w systemie MS-DOS na procesorach firmy Intel. Dostawcy komputerów, tacy jak firmy Sun Microsystems i Digital Equipment Corporation (DEC), stosują inne, szybsze procesory, lecz pragnęliby, aby ich nabywcy mogli korzystać z owych programów systemu MS-DOS. Rozwiązaniem jest utworzenie wirtualnej maszyny Intel nadbudowanej nad rdzennym procesorem. Program systemu MS-DOS działa w takim środowisku, a jego rozkazy dla procesora Intel są tłumaczone na zbiór rdzennych rozkazów. System MS-DOS również działa na takiej maszynie wirtualnej, programy mogą więc korzystać z jego funkcji tak jak zwykle. W rezultacie otrzymuje się program sprawiający wrażenie, że jest wykonywany w systemie opartym na procesorze Intel, choć w rzeczywistości działa on na zupełnie innym procesorze. Jeśli procesor ten jest wystarczająco szybki, to program systemu MS-DOS działa szybko, chociaż każdy jego rozkaz w celu wykonania jest tłumaczony na kilka rozkazów rodzimego procesora. Analogicznie, komputer Apple Macintosh z procesorem PowerPC zawiera maszynę wirtualną Motorola 68000, umożliwiającą wykonywanie programów binarnych napisanych dla starszego modelu komputera Macintosh – z procesorem 68000.

3.6.3 Java

Jeszcze innym przykładem ciągłości koncepcji maszyn wirtualnych jest język Java. Java jest bardzo popularnym językiem zaprojektowanym przez firmę Sun Microsystems. Kompilator języka Java wytwarza tzw. *kod pośredni* (ang. *bytecode*). Kod pośredni (zwany też bajtkodem) zawiera instrukcje wykonywane przez *wirtualną maszynę Javy* (ang. *Java Virtual Machine* – JVM). Aby program w języku Java mógł być wykonywany na jakiejs platformie, musi w niej działać maszyna JVM. Maszyna JVM działa na wielu typach komputerów, w tym na zgodnych ze standardem IBM komputerach osobistych, na komputerach Macintosh, stacjach roboczych i serwerach systemu UNIX, jak również na minikomputerach i komputerach głównych typu IBM. Maszyna JVM jest także zrealizowana w przeglądarkach sieciowych, takich jak Microsoft Explorer lub Netscape Communicator. Z kolei te przeglądarki działają na różnorodnym sprzęcie i w różnych systemach operacyjnych. Maszyna JVM jest także zrealizowana w malym systemie operacyjnym JavaOS, który implementuje ją wprost na sprzęcie, aby uniknąć kosztów powodowanych przez działanie systemu Java w systemach operacyjnych ogólnego przeznaczenia. Na koniec urządzenia specjalizowane, jak telefony komórkowe, mogą być oprogramowane za pomocą języka Java przez użycie mikroprocesorów wykonujących jako rdzenne instrukcje kod pośredni Javy.

Maszyna wirtualna JVM implementuje z użyciem stoso zbior rozkazów zawierający niezbędne działania arytmetyczne, logiczne, operacje przemiesz-

czania danych oraz rozkazy sterujące. Ponieważ jest to maszyna wirtualna, może również realizować instrukcje, które są zbyt złożone, aby można było wbudować je w sprzęt, w tym instrukcje tworzenia obiektów, działań na obiektach oraz instrukcje wywoływanego metod. Kompilatory języka Java mogą po prostu generować instrukcje kodu pośredniego, a maszyna JVM musi zapewniać ich realizację na poszczególnych typach komputerów.

W projekcie Java korzysta się z kompletnego środowiska realizowanego przez maszynę wirtualną. Kod pośredni jest na przykład sprawdzany pod kątem występowania rozkazów mogących naruszać bezpieczeństwo lub niezawodność podstawowej maszyny. Jeśli program w języku Java nie zaliczy takiego sprawdzianu, to nie zezwala się na jego wykonanie. Dzięki zrealizowaniu języka Java w konwencji maszyny wirtualnej firma Sun utworzyła wydajne, dynamiczne, bezpieczne i przenośne udoskonalenie obiektowe. Choć programy w języku Java nie są tak szybkie, jak programy tłumaczone na rodzime zbiory instrukcji, są one wydajniejsze od programów interpretowanych i mają kilka zalet w stosunku do języków komplikowanych na kod rodzimy, takich jak C.

3.7 ■ Projektowanie i implementacja systemu

Omówimy teraz zagadnienia projektowania i implementowania systemu operacyjnego. Nie podamy – rzecz jasna – pełnych rozwiązań zagadnień projektowych, lecz opiszemy podejścia sprawdzone w praktyce.

3.7.1 Założenia projektowe

Pierwszym zagadnieniem przy projektowaniu systemu jest określenie celów i specyfikacji systemu. Na najwyższym poziomie projektu systemu będzie w dużym stopniu zależał od wyboru sprzętu i typu systemu – czy ma to być system wsadowy, z podziałem czasu, dla jednego użytkownika czy wielozadaniowy, rozproszony, pracujący w czasie rzeczywistym czy też ogólnego przeznaczenia.

Wyższy poziom najwyższy, sformułowanie wymagań może być znacznie trudniejsze. Wymagania te można zasadniczo podzielić na dwie grupy: *cele użytkownika i cele systemu*.

Użytkownicy oczekują od systemu operacyjnego pewnych oczywistych właściwości: powinien on być wygodny i łatwy w użyciu, łatwy do nauki, niezawodny, bezpieczny i szybki. Jest zrozumiałe, że ze względu na brak powszechnej zgody co do sposobu osiągania tak określonych celów, tego rodzaju specyfikacje nie są zbyt przydatne przy projektowaniu systemu.

Podobny zestaw wymagań mogą sformułować osoby, których zadaniem jest taki system zaprojektować, utworzyć, utrzymywać i obsługiwać. System operacyjny powinien być łatwy do zaprojektowania, realizacji i pielęgnowania; powinien być elastyczny, niezawodny, wolny od błędów i wydajny. I znowu, tak postawione wymagania są niejasne i nie ma ogólnego sposobu ich zrealizowania.

Problem zdefiniowania wymagań względem systemu operacyjnego nie ma jednoznacznego rozwiązania. Można się przekonać na podstawie dużej rozmaitsci systemów, że różne wymagania prowadzą do różnaitych rozwiązań w różnych środowiskach. Na przykład wymagania względem jednozadaniowego systemu MS-DOS dla mikrokomputerów musiały być całkiem różne od wymagań postawionych systemowi MVS – wielkiemu wielozadaniowemu, wielodostępnemu systemowi operacyjnemu dużych komputerów IBM.

Sformułowanie specyfikacji i projektowanie systemu operacyjnego jest pracą twórczą. W żadnym podręczniku nie znajdzie się gotowych rozwiązań. Istnieją jednak pewne ogólne zasady, z których można skorzystać. *Inżynieria oprogramowania* stanowi bazę tych zasad; niektóre spośród nich są szczególnie przydatne w systemach operacyjnych.

3.7.2 Mechanizm a polityka

Jedną z ważnych zasad jest oddzielenie *polityki* od *mechanizmu*. Mechanizmy określają, jak czegoś dokonać. Natomiast polityka decyduje o tym, co ma być zrobione. Na przykład mechanizmem gwarantującym ochronę procesora jest czasomierz (p. 2.5). Decyzja o tym, na jak długo czasomierz jest nastawiony dla poszczególnych użytkowników, leży w sferze polityki.

Odseparowanie polityki od mechanizmu jest bardzo ważne ze względu na elastyczność. Zasady polityki zmieniają się zależnie od miejsca i czasu. W najgorszym przypadku każda zmiana polityki może pociągać konieczność zmiany leżącego u jej podłoża mechanizmu. O wiele bardziej pożądany jest mechanizm ogólny. Zmiana polityki wymaga wtedy co najwyżej przedefiniowania pewnych parametrów w systemie. Na przykład, jeśli w jednej instalacji podjęto decyzję polityczną dającą programom intensywnie korzystającym z wejścia-wyjścia pierwszeństwo nad programami używającymi intensywnie procesora, to gdy stosowny mechanizm został odpowiednio wydzielony i uniezależniony od polityki, wówczas w innym systemie komputerowym można łatwo zrealizować politykę przeciwną.

Systemy operacyjne mające jako podstawę mikrojądro traktują rozdział między mechanizmem a polityką w sposób skrajny, realizując podstawowy zbiór elementarnych działań składowych. Składowe te są niemal całkowicie niezależne od sposobu ich użycia, umożliwiając dołączanie bardziej zaawan-

sowanych mechanizmów i polityk w formie modułów jądra utworzonych przez użytkowników lub wprost za pomocą programów użytkowych. Na przeciwnym krańcu znajduje się system operacyjny taki jak Apple Macintosh, w którym zakodowano zarówno mechanizmy, jak i sposoby ich użycia, żeby wymusić ogólny wygląd i odbiór systemu. Wszystkie aplikacje mają podobne interfejsy, gdyż sam interfejs jest wbudowany w jądro.

Decyzje polityczne są ważne we wszystkich zagadnieniach planowania i przydzielu zasobów. Ilekroć staje się niezbędne rozstrzygnięcie o przydzieleniu bądź nieprzydzieleniu zasobu, tylekroć musi zostać podjęta decyzja podkutowana jakąś polityką. Skoro tylko pada pytanie „jak” zamiast pytania „co”, wówczas oznacza ono potrzebę określenia jakiegoś mechanizmu.

3.7.3 Implementacja systemu

Po zaprojektowaniu system operacyjny trzeba zrealizować. Tradycyjnie systemy operacyjne pisano w językach asemblerowych. Jednak zasada ta przestała już obowiązywać. Systemy operacyjne można obecnie pisać w językach wyższego poziomu.

Pierwszym systemem, przy pisaniu którego nie posłużyono się asemblerem, był prawdopodobnie Master Control Program (MCP) dla komputerów firmy Burroughs. System operacyjny MCP napisano w odmianie Algolu. System MULTICS, opracowany w MIT, został napisany głównie w PL/I. System operacyjny Primos dla komputerów Prime napisano w dialekcie Fortranu. Systemy operacyjne: UNIX, OS/2 i Windows NT napisano niemal w całości w języku C. Zaledwie 900 wierszy kodu oryginalnego systemu UNIX zaprogramowano w asemblerze, z czego większość dotyczyła programu planującego i modułów obsługi urządzeń.

Zalety użycia do zaprogramowania systemu operacyjnego języka wyższego poziomu lub przynajmniej języka przeznaczonego do implementacji systemów są takie same jak w przypadku stosowania tych języków w programach użytkowych. Programuje się szybciej, uzyskując kod bardziej zwarty, łatwiejszy do zrozumienia i sprawdzenia. Ponadto, jeśli zastąpi się kompilator jego lepszą wersją, to zwykle powtórne przetłumaczenie całego systemu operacyjnego poprawi jakość jego kodu. Wreszcie, system operacyjny jest znacznie łatwiejszy do *przenoszenia*, czyli instalowania na innym sprzęcie, jeśli jest napisany w języku wysokiego poziomu. Na przykład system MS-DOS napisano w asemblerze mikroprocesora Intel 8088. W efekcie jest on dostępny tylko w rodzinie procesorów firmy Intel. Natomiast system operacyjny UNIX, który napisano w C, może pracować na wielu różnych procesorach, w tym na procesorze Intel 80X86, Motorola 680X0, SPARC i Mips RX000.

Za główne wady realizacji systemu operacyjnego w języku wyższego poziomu uznaje się spowolnienie jego działania i większe zapotrzebowanie na pamięć. Chociaż znawca programowania w asemblerze może wytworzyć bardzo wydajne, małe procedury, w przypadku wielkich programów nowoczesny kompilator, wykonując niezwykle skomplikowaną analizę i stosując wymyślne optymalizacje, wyprodukuje bardzo dobry kod. Odnosi się to w szczególności do nowoczesnych procesorów, w znacznym stopniu korzystających z przetwarzania potokowego i dysponujących licznymi modułami funkcjonalnymi, w których ogarnięcie szczegółów ich złożonych zależności może przekraczać ograniczone zdolności ludzkiego umysłu.

Tak jak w innych systemach, główne ulepszenia działania są zwykle rezultatem lepszych struktur danych i algorytmów aniżeli doskonałego kodowania w języku asemblerowym. W dodatku, choć rozmiary systemów operacyjnych są bardzo wielkie, na efektywność ich działania ma istotny wpływ tylko niewielka część kodu; najważniejszymi procedurami są tu prawdopodobnie: zarządcą pamięci i planista przydziału procesora. Kiedy system zostanie napisany i pracuje poprawnie, wtedy można zidentyfikować procedury będące jego wąskimi gardłami i zastąpić je asemblerowymi odpowiednikami.

Aby wykryć wąskie gardła, trzeba mieć możliwość kontrolowania działania systemu. Konieczne okaże się dodanie kodu, który obliczy i wyświetli pomiary zachowania systemu. Wszystkie interesujące zdarzenia, wraz z ich czasem i ważnymi parametrami, są rejestrowane i zapisywane w pliku. Potem program analizujący może przetworzyć plik z zapisem zdarzeń w celu zbadania zachowania systemu oraz zidentyfikowania jego wąskich gardel i nieefektywnych zachowań. Ten sam ślad programu może również posłużyć jako dane wejściowe do symulacji proponowanego ulepszenia systemu. Ślady programów znajdują także zastosowanie przy wykrywaniu błędów w zachowaniu systemu operacyjnego.

Alternatywna możliwość polega na obliczeniu i wyświetleniu pomiarów wydajności systemu w czasie rzeczywistym. Takie podejście pozwala operatorom systemu lepiej poznać zachowanie systemu i na bieżąco modyfikować jego działanie.

3.8 ■ Generowanie systemu

Można zaprojektować i zrealizować system operacyjny wyłącznie dla jednej maszyny w jednej instalacji. Częściej jednak projektuje się systemy operacyjne przeznaczone do działania na pewnej klasie maszyn w rozmaitych instalacjach ze zmienną konfiguracją urządzeń zewnętrznych. System musi wówczas podlegać konfigurowaniu lub generowaniu dla każdej specyficznej instalacji

komputerowej. Proces ten nazywa się niekiedy *generowaniem systemu* (SYSGEN).

System operacyjny jest zazwyczaj rozpowszechniany na taśmie lub dysku. Do generowania systemu uzywa się specjalnego programu. Informacje dotyczące specyfiki konfiguracji danego wyposażenia sprzętowego program SYSGEN systemu czyta z określonego pliku lub pyta o nie operatora bądź też sam sonduje sprzęt, aby określić jego składowe. Należy ustalić następujące informacje:

- Jaki zastosowano procesor centralny? Jakie dodatkowe możliwości (rozszerzony zbiór rozkazów, arytmetyka zmiennopozycyjna itd.) zostały zainstalowane? W systemach wieloprocesorowych należy opisać każdy procesor.
- Ile jest dostępnej pamięci? Niektóre systemy same określają ten parametr – dopóty odwołując się do kolejnych komórek pamięci, dopóki nie wystąpi błąd: „niedozwolony adres”. Procedura ta określa maksymalny pozwalany adres, a zatem i ilość dostępnej pamięci.
- Jakie są dostępne urządzenia? System będzie wymagał podania sposobu adresowania każdego urządzenia (numeru urządzenia), numeru przerwania od danego urządzenia, opisu typu i modelu urządzenia oraz wszelkich specjalnych parametrów technicznych urządzeń.
- Jakich oczekuje się możliwości systemu operacyjnego? Mówiąc inaczej – jakie mają być wartości parametrów instalacyjnych systemu. Te wartości mogą zawierać między innymi informacje o liczbie i wielkości buforów, pożdanym typie algorytmu planowania przydziału procesora, maksymalnej przewidywanej liczbie procesów itp.

Określone w ten sposób dane mogą być wykorzystane na kilka sposobów. W skrajnym przypadku mogą posłużyć do wprowadzenia zmian do kopii kodu źródłowego systemu. Program systemu operacyjnego musi zostać potem w całości skompilowany. Zmiany w deklaracjach danych, wartościach początkowych i stałych oraz warunkowa komplikacja spowodują wyprodukowanie wynikowej wersji systemu, przykrojonej na miarę sformułowanych wymagań.

W bardziej elastycznej metodzie opis systemu może spowodować utworzenie tablic i wybranie zawczasu skompilowanych modułów z biblioteki. Wygenerowanie systemu nastąpi w wyniku skonsolidowania tych modułów. Selektywne wybieranie modułów z biblioteki pozwala na umieszczenie w niej modułów sterujących dla wszystkich przewidywanych urządzeń wejścia-wyjścia, a jednocześnie dołączenie do systemu operacyjnego tylko tych, które naprawdę będą używane. Ponieważ system nie musi być ponownie kompi-

lowany, jego wygenerowanie będzie szybsze. Może jednak powstac system bardziej ogólny niż faktycznie potrzebny.

Na przeciwnym biegunie leży możliwość skonstruowania systemu całkowicie sterowanego tablicami. System zawiera zawsze cały kod, a wybór dokonuje się nie podczas komplikacji lub konsolidacji, lecz w czasie wykonania. Generowanie systemu sprowadza się do wytworzenia odpowiednich tabel opisujących system.

Główne różnice w tych podejściach przejawiają się w rozmiarze i ogólności wygenerowanego systemu oraz łatwości dokonywania w nim zmian w wypadku zmian w sprzęcie. Za przykład może posłużyć nakład pracy potrzebny do usytuowania w systemie nowo nabytego terminalu graficznego albo dodatkowego napędu dysku. Na wyważanie tego kosztu ma – oczywiście – wpływ częstotliwość (lub rzadkość) takich zmian.

Po wygenerowaniu systemu operacyjnego należy go udostępnić do użytku przez sprzęt komputerowy. Powstaje pytanie, w jaki sposób sprzęt rozpoznaje położenie jądra lub dokonuje załadowania. Procedura rozpoczęcia pracy komputera przez załadowanie jądra nosi nazwę rozruchu (ang. *booting*) systemu. W większości systemów komputerowych istnieje mały fragment kodu, przechowywany w pamięci ROM, określany jako program rozruchowy (ang. *bootstrap program*) lub elementarny program ładowający (ang. *bootstrap loader*). Kod ten jest w stanie zlokalizować jądro, wprowadzić je do pamięci i rozpocząć jego wykonywanie. W niektórych systemach komputerowych, np. w komputerach PC, stosuje się postępowanie dwuetapowe, w którym bardzo uproszczony elementarny program ładowający sprowadza z dysku bardziej złożony program ładowający i dopiero ten program powoduje załadowanie jądra. Rozruch systemu omawiamy jeszcze w p. 13.3.2 i w rozdz. 21.

3.9 ■ Podsumowanie

Systemy operacyjne dostarczają wielu usług. Na najniższym poziomie funkcje systemowe pozwalają, aby wykonywany program zgłaszał zapotrzebowania na usługi bezpośrednio do systemu operacyjnego. Na wyższym poziomie interpreter poleceń, czyli powłoka, tworzy mechanizm, za pomocą którego użytkownik może kierować polecenia pod adresem systemu bez konieczności pisania programu. Polecenia mogą pochodzić z plików, przy działaniu w trybie wsadowym, lub wprost z terminalu – w interakcyjnym systemie z podziałem czasu. Mechanizm przeznaczony do spełniania typowych życzeń użytkownika tworzą programy systemowe.

Rodzaje zadań zależą od poziomu zgłoszenia. Poziom funkcji systemowych musi zapewnić podstawowe operacje, takie jak nadzór nad procesami

oraz działania na plikach i urządzeniach. Zadania wyższego poziomu, spełniane przez interpreter polecen lub programy systemowe, są tłumaczone na ciągi wywołań funkcji systemowych. Usługi systemu można podzielić na kilka kategorii: nadzór nad wykonywaniem programów, podawanie informacji o stanie systemu oraz obsługa zamówień na operacje wejścia-wyjścia. Błędy programowe można uważać za niejawne żądania obsługi.

Gdy określi się usługi systemu operacyjnego, wówczas można przystąpić do opracowania jego struktury. Znajdują tu zastosowanie liczne tablice, w których odnotowuje się stan systemu komputerowego i stan zadań systemowych.

Zaprojektowanie nowego systemu operacyjnego jest poważnym przedsięwzięciem. Jest bardzo ważne, aby przed przestąpieniem do pracy nad projektem zostały dobrze zdefiniowane cele systemu. Punktem wyjścia do wybierania niezbędnych rozwiązań spośród różnych algorytmów i strategii jest określenie pożądanego typu systemu.

Wielkie rozmiary systemu operacyjnego obligują do zatroszczenia się o jego modułarność. Ważną techniką projektowania jest rozważanie systemu jako układu warstw. Konsepcja maszyny wirtualnej, stanowiąc esencję podejścia warstwowego, zacieka różnice między jądrem systemu operacyjnego a sprzętem, traktując jedno i drugie jak sprzęt. Na szczycie tak rozumianej maszyny wirtualnej można umieszczać inne systemy operacyjne.

Przez cały czas cyklu projektowania systemu operacyjnego należy starannie oddzielać decyzje dotyczące polityki od szczegółów realizacyjnych. Przestrzeganie tego rozdziału zapewnia maksimum elastyczności w przypadku późniejszych zmian w decyzjach politycznych.

Obecnie systemy operacyjne są prawie zawsze pisane w językach implementacji systemów lub w językach wyższego poziomu. Polepsza to warunki implementacji, pielęgnowania i przenośności. Aby utworzyć system operacyjny dla konkretnej konfiguracji sprzętu, musimy ten system wygenerować.

■ Ćwiczenia

- 3.1 Określ pięć głównych czynności wykonywanych przez system operacyjny w związku z zarządzaniem procesami.
- 3.2 Określ trzy główne czynności wykonywane przez system operacyjny w związku z zarządzaniem pamięcią.
- 3.3 Określ trzy główne czynności wykonywane przez system operacyjny w związku z zarządzaniem pamięcią pomocniczą.
- 3.4 Określ pięć głównych czynności wykonywanych przez system operacyjny w związku z zarządzaniem plikami.

- 3.5 Co należy do zadań interpretera poleceń? Dlaczego na ogół jest on oddzielony od jądra?
- 3.6 Wymień pięć usług wykonywanych przez system operacyjny. Wyjaśnij, na czym polega wygoda dla korzystającego z nich użytkownika. Wyjaśnij, w których przypadkach nie byłoby możliwe zrealizowanie tych usług w programie pracującym na poziomie użytkownika.
- 3.7 Do czego służą funkcje systemowe?
- 3.8 Do czego służą programy systemowe?
- 3.9 Na czym polega główna zaleta podejścia warstwowego przy projektowaniu systemu?
- 3.10 Co w używaniu maszyny wirtualnej stanowi główną zaletę dla projektanta systemu operacyjnego? A jaka jest tu główna korzyść dla użytkownika?
- 3.11 Dlaczego jest wskazane oddzielanie mechanizmu od polityki?
- 3.12 Rozważmy eksperymentalny system operacyjny Synthesis z asemblerem wbudowanym w jądro. W celu zoptymalizowania działania funkcji systemowych jądro umieszcza przekład odpowiadających im procedur we własnej przestrzeni adresowej w celu zminimalizowania ścieżki, którą odwołanie do systemu musi w nim przechodzić. Podejście to jest antytezą podejścia warstwowego, w którym ścieżkę przez jądro się wydłuża, dzięki czemu budowanie systemu operacyjnego staje się łatwiejsze. Omów za i przeciw takiego podejścia do projektowania jądra oraz do optymalizowania działania systemu.

Uwagi bibliograficzne

Można przyjąć, że języki polecen są specjalistycznymi językami programowania. Brunt i Tuffs w artykule [59] twierdzą, że język polecen powinien zawierać bogaty zestaw funkcji; Frank w pracy [140] opowiada się za bardziej ograniczonym, uproszczonym językiem polecen. Znakomitym przykładem takiego języka jest powłoka systemu UNIX, opisana przez Bourne'a w artykule [47].

Podejście warstwowe w projektowaniu systemów operacyjnych zalecał Dijkstra w pracy [112], w której opisał system THE. System Venus przedstawił Liskov w artykule [259].

Brinch Hansen jest autorem zamieszczonej w artykule [54] wcześniejszej propozycji budowy systemu operacyjnego w postaci jądra (lub rdzenia; ang.

nucleus), na którym można opierać bardziej złożone systemy. Architekturę komputerów odpowiednich do implementacji systemów o strukturze wielopoziomowej opisali Bernstein i Siegel w artykule [32].

Pierwszym systemem operacyjnym realizującym maszynę wirtualną był CP/67 na IBM 360/67. Opisali go Meyer i Seawright [286]. System CP/67 zaopatrywał każdego użytkownika w maszynę wirtualną IBM 360 Model 65, włącznie z urządzeniami wejścia-wyjścia. Komercyjny system operacyjny IBM VM/370 wywodził się z systemu CP/67; jego opis zawierają artykuły Seawrighta i MacKinnona [385], Holleya i in. [175] oraz Creasy'ego [88]. Hall i in. w pracy [162] promowali użycie maszyn wirtualnych w celu zwiększenia przenośności systemu operacyjnego. Jones w artykule [199] sugerował posłużenie się maszynami wirtualnymi do wymuszenia izolacji procesów w celach ochronnych. Ogólną analizę maszyn wirtualnych zaprezentowali Hendricks i Hartmann [168], MacKinnon [266] i Schultz [382].

MS-DOS w wersji 3.1 jest przedstawiony w książce [288]. System Windows NT opisała Helen Custer [91]. Omówienie systemu operacyjnego Apple Macintosh można znaleźć w dokumentacji [13]. Opis systemu UNIX BSD z Berkeley znajduje się w dokumentacji [89]. Standard AT&T UNIX System V jest opisany w podręczniku [18]. Dobry opis systemu OS/2 podał Iacobucci [185]. Wstęp do systemu Mach można znaleźć w artykule Accetty [2], a system AIX jest zaprezentowany w pracy Loucksa i Sauera [264]. Eksperymentalny system operacyjny Synthesis omówili Massalin i Pu [272]. Więcej informacji dotyczących języka Java i maszyny JVM można uzyskać bezpośrednio pod adresem <http://www.javasoftware.com>, a także w książce Meyera i Downinga [285].

Część 2

ZARZĄDZANIE PROCESAMI

Przez *proces* można rozumieć wykonujący się program. Do wypełnienia swojego zadania proces potrzebuje pewnych zasobów, takich jak czas procesora, pamięć, pliki i urządzenia wejścia-wyjścia. Zasoby te są przydzielane procesowi w chwili jego tworzenia lub podczas późniejszego działania.

W większości systemów proces jest jednostką pracy. W takim ujęciu system składa się ze zbioru procesów. Procesy systemu operacyjnego wykonują kod systemowy, a procesy użytkownika działają według kodu dostarczonego przez użytkownika. Wszystkie procesy mogą być w zasadzie wykonywane współbieżnie.

System operacyjny odpowiada za następujące działania dotyczące zarządzania procesami: tworzenie i usuwanie zarówno procesów użytkownika, jak i procesów systemowych, planowanie porządku wykonywania procesów oraz za mechanizmy synchronizacji, komunikacji i usuwanie zakleszczeń procesów.



Rozdział 4

PROCESY

Pierwsze systemy komputerowe umożliwiały wykonywanie tylko jednego programu w danej chwili. Program taki sprawował całkowity nadzór nad systemem, a do jego dyspozycji pozostawały wszystkie zasoby systemu. Współczesne systemy komputerowe pozwalają na umieszczenie w pamięci operacyjnej wielu programów i współbieżne ich wykonywanie. Te zmiany pociągnęły za sobą konieczność ostrzejszej kontroli i większego odseparowania od siebie poszczególnych programów. Doprowadziło to do powstania pojęcia *procesu*, przez które rozumie się program będący w trakcie wykonywania. Proces jest jednostką pracy w nowoczesnym systemie z podziałem czasu.

Im bardziej wzrasta złożoność systemu operacyjnego, tym większego oczekuje się od niego zakresu usług świadczonych dla użytkowników. Choć do głównych obowiązków systemu należy wykonywanie programów użytkowych, musi on również troszczyć się o rozmaite zadania systemowe, które wygodniej jest pozostawić poza jądrem. System składa się więc ze zbioru procesów: procesy systemu operacyjnego wykonują kod systemowy, a procesy użytkowe działają według kodu należącego do użytkowników. Wszystkie te procesy potencjalnie mogą być wykonywane współbieżnie dzięki podziałowi między nie mocy obliczeniowej procesora (lub procesorów). Przelaczając procesor do poszczególnych procesów, system operacyjny może zwiększyć wydajność komputera.

4.1 ■ Koncepcja procesu

Jedną z trudniejszych kwestii napotykanych podczas omawiania systemów operacyjnych jest pytanie o to, jak nazwać wszystkie czynności procesora. System wsadowy wykonuje *zadania* (ang. *jobs*), podczas gdy system z po-

działem czasu ma do czynienia z programami użytkownika (ang. *user programs*) albo pracami (ang. *tasks*). Nawet w systemie dla indywidualnego użytkownika, takim jak MS-DOS lub Macintosh OS, można wykonywać kilka programów jednocześnie: jeden interakcyjny i kilka programów wsadowych. Nawet jeśli użytkownik może wykonywać tylko jeden program w danym czasie, to system operacyjny może wykonywać swoje wewnętrznie zaprogramowane czynności, takie jak spooling. Wszystkie te działania są pod wieloma względami podobne, toteż będziemy je wspólnie nazywać *procesami*.

Terminy *zadanie* i *proces* są w niniejszym tekście używane prawie zamiennie. Choć osobiście preferujemy termin *proces*, znaczna część teorii i terminologii systemów operacyjnych powstała w czasach, kiedy podstawową czynnością systemów operacyjnych było przetwarzanie zadań. Byłoby nieporozumieniem rezygnować z używania powszechnie akceptowanych terminów, w których występuje słowo *zadanie* (np. planowanie zadań), tylko z tego powodu, że zastąpiono je słowem *proces*.

4.1.1 Proces

Określając nieformalnie, proces jest wykonywanym programem. Wykonanie procesu musi przebiegać w sposób sekwencyjny. Oznacza to, że w dowolnej chwili na zamówienie danego procesu może być wykonywany co najwyżej jeden rozkaz kodu programu.

Proces jest czymś więcej niż samym kodem programu (niekiedy nazywanym *sekcją tekstu*, ang. *text section*). W pojęciu procesu mieści się również bieżąca czynność reprezentowana przez wartość *licznika rozkazów* (ang. *program counter*) oraz zawartość rejestrów procesora. Do procesu na ogół należą także: *stos procesu* (ang. *process stack*) przechowujący dane tymczasowe (takie jak parametry procedur, adresy powrotne i zmienne tymczasowe) oraz *sekcja danych* (ang. *data section*) zawierająca zmienne globalne.

Podkreślamy, że sam program nie jest procesem. Program jest obiektem *pasywnym*, podobnie jak zawartość pliku przechowywanego na dysku. Natomiast proces jest obiektem *aktywnym*, z licznikiem rozkazów określającym następny rozkaz do wykonania i ze zbiorem przydzielonych mu zasobów.

Chociaż dwa procesy mogą być związane z jednym programem, będą one zawsze traktowane jako dwie oddzielne sekwencje wykonania. Na przykład wielu użytkowników może korzystać z działania kopii programu pocztowego lub jeden użytkownik może zapoczątkować pracę wielu kopii edytora. W każdym z tych przypadków mamy do czynienia z osobnymi procesami, które – niezależnie od równoważności sekcji tekstu – będą się różniły sekcjami danych. Na porządku dziennym jest też tworzenie przez wykonywany proces wielu nowych procesów. Zagadnienie to omówimy w p. 4.4.



Rys. 4.1 Diagram stanów procesu

4.1.2 Stan procesu

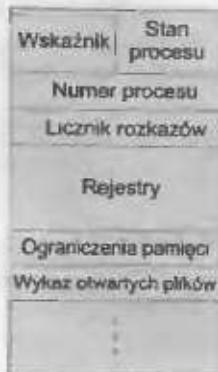
Wykonujący się proces zmienia swój *stan* (ang. *state*). Stan procesu jest po części określony przez bieżącą czynność procesu. Kazdy proces może się znajdować w jednym z następujących stanów:

- **nowy** – proces został utworzony;
- **aktywny** – są wykonywane instrukcje;
- **oczekiwanie** – proces czeka na wystąpienie jakiegoś zdarzenia (np. na zakończenie operacji wejścia-wyjścia);
- **gotowy** – proces czeka na przydział procesora;
- **zakończony** – proces zakończył działanie.

Powyzsze nazwy są dobrane dość dowolnie i w różnych systemach operacyjnych można napotkać inne nazwy. Niemniej jednak określane przez nie stany występują we wszystkich systemach. W niektórych systemach operacyjnych stany procesu rozróżnia się bardziej szczegółowo. Ważne jest, aby zdawać sobie sprawę z tego, że w każdej chwili w dowolnym procesorze tylko jeden proces może być *aktywny*, ale wiele procesów może być *gotowych* do działania lub *czekających*. Diagram opisanych stanów widać na rys. 4.1.

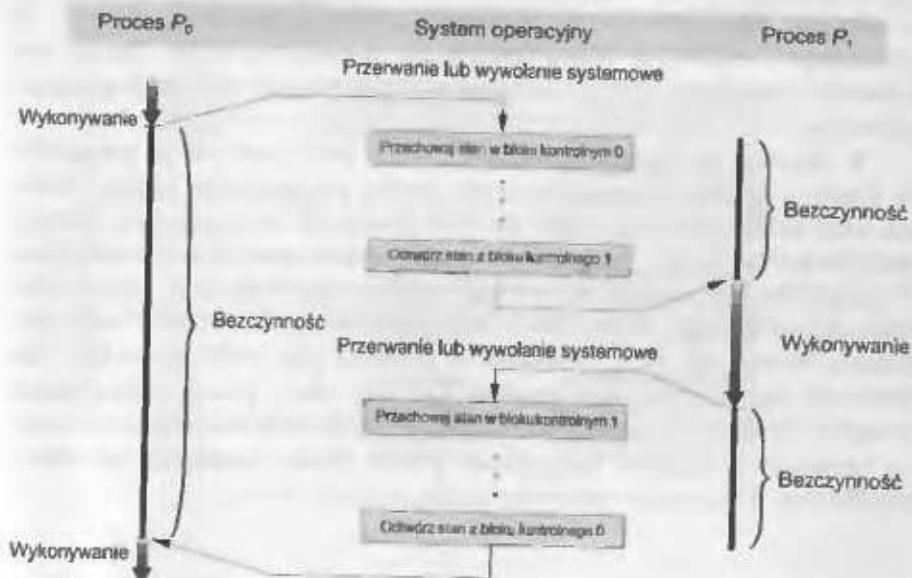
4.1.3 Blok kontrolny procesu

Każdy proces jest reprezentowany w systemie operacyjnym przez *blok kontrolny procesu* (ang. *process control block* – PCB), nazywany również blokiem kontrolnym zadania. Blok kontrolny procesu jest pokazany na rys. 4.2. Zawiera on wiele informacji związanych z danym procesem, które obejmują:



Rys. 4.2 Blok kontrolny procesu

- **Stan procesu:** Stan może być określony jako nowy, gotowy, aktywny, oczekiwanie, zatrzymanie itd.
- **Licznik rozkazów:** Licznik ten wskazuje adres następnego rozkazu do wykonania w procesie.
- **Rejestry procesora:** Liczba i typy rejestrów zależą od architektury komputera; są tu akumulatory, rejesty indeksowe, wskaźniki stosu, rejesty ogólnego przeznaczenia oraz rejesty warunków. Informacje dotyczące tych rejestrów i licznika rozkazów muszą być przechowywane podczas przerwań, aby proces mógł być później poprawnie kontynuowany (rys. 4.3).
- **Informacje o planowaniu przydziału procesora:** Należą do nich: priorytet procesu, wskaźniki do kolejek porządkujących zamówienia, a także inne parametry planowania (planowanie procesów opisujemy w rozdz. 5).
- **Informacje o zarządzaniu pamięcią:** Mogą to być informacje takie, jak zawartości rejestrów granicznych, tablice stron lub tablice segmentów – zależnie od systemu pamięci używanej przez system operacyjny (rozdz. 8).
- **Informacje do rozliczeń:** Do tej kategorii informacji należy ilość użytego czasu procesora i czasu rzeczywistego, ograniczenia czasowe, numery kont, numery zadań lub procesów itp.
- **Informacje o stanie wejścia-wyjścia:** Miesząc się tu informacje o urządzeniach wejścia-wyjścia (np. przewijakach taśmy) przydzielonych do procesu, wykaz otwartych plików itd.



Rys. 4.3 Procesor może być przełączany do różnych procesów

Blok kontrolny procesu służy po prostu jako magazyn przechowujący wszelkie informacje różne dla różnych procesów.

4.2 ■ Planowanie procesów

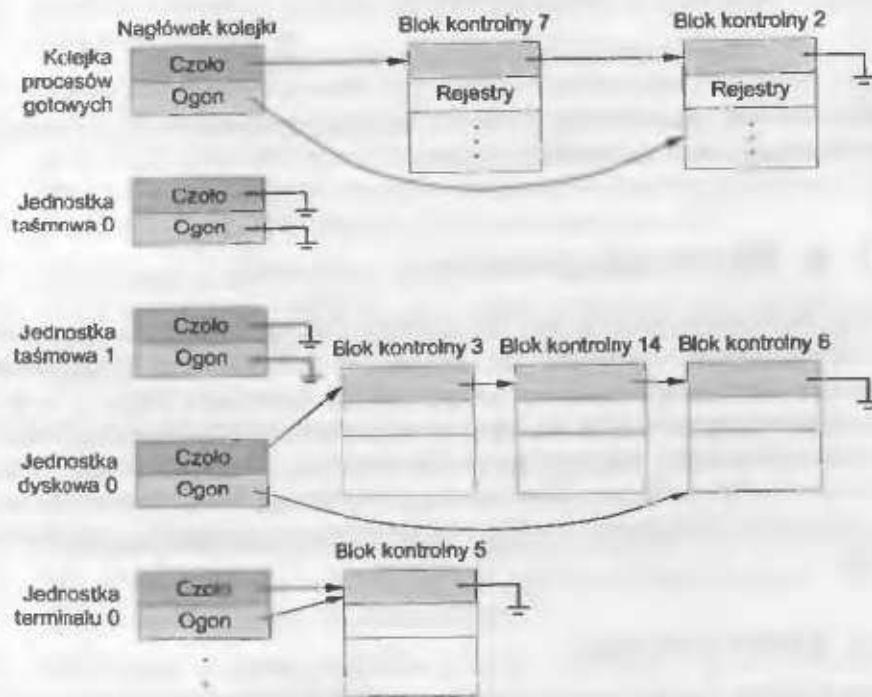
Celem wieloprogramowania jest jak najlepsze wykorzystanie procesora – powinien on zawsze wykonywać jakiś proces. W podziale czasu przełączanie procesora do procesów powinno następować tak często, aby każdy z wykonywanych programów mógł współpracować z użytkownikami w sposób interakcyjny. W systemie jednoprocesorowym aktywny może być zawsze tylko jeden proces. Wszystkie pozostałe procesy, o ile istnieją, muszą czekać do czasu, aż procesor będzie wolny i będzie można ponownie przydzielić go jednemu z nich.

4.2.1 Kolejki planowania

Wchodzące do systemu procesy trafiają do *kolejki zadań* (ang. *job queue*). Kolejka ta zawiera wszystkie procesy systemu. Gotowe do działania procesy, które oczekują w pamięci głównej, są trzymane na liście zwanej *kolejką pro-*

cesów gotowych (ang. *ready queue*). Kolejka ta ma zazwyczaj postać listy powiązanej. Nagłówek kolejki procesów gotowych zawiera wskaźniki do pierwszego i ostatniego bloku kontrolnego procesu na liście. Każdy blok kontrolny procesu ma pole wskazujące następną pozycję w kolejce procesów gotowych.

W systemie są również inne kolejki. Kiedy procesowi zostaje przydzielony procesor, wtedy proces działa przez chwilę, po czym albo kończy działanie, albo zostaje przerwany, albo zaczyna oczekiwania na wystąpienie jakiegoś specjalnego zdarzenia – na przykład na zakończenie operacji wejścia-wyjścia. W przypadku zamówienia na operację wejścia-wyjścia może to być na przykład żądanie dostępu do osobnego przewijaka taśmy lub do urządzenia dzielonego, takiego jak dysk. Ponieważ w systemie jest wiele procesów, dysk może być zajęty przez inne procesy. Tak więc dany proces będzie musiał poczekać na dysk. Lista procesów czekających na konkretne urządzenie zwie się kolejką do urządzenia (ang. *device queue*). Każde urządzenie ma własną kolejkę (rys. 4.4).



Rys. 4.4 Kolejka procesów gotowych do wykonania | kolejki do różnych urządzeń wejścia-wyjścia



Rys. 4.5 Diagram kolejek w planowaniu procesów

Typową reprezentacją ułatwiającą omawianie planowania (szeregowania) procesów jest *diagram kolejek*, taki jak na rys. 4.5. Każdy prostokąt przedstawia kolejkę. Występują dwa typy kolejek: kolejka procesów gotowych i zbiór kolejek do urządzeń. Koła oznaczają zasoby obsługujące kolejki, a strzałki pokazują kierunek przepływu procesów w systemie.

Nowy proces jest na początku umieszczany w kolejce procesów gotowych. Oczekuje w niej do czasu, aż zostanie wybrany do wykonania (czyli wyekspediowany; ang. *dispatched*) i otrzyma procesor. Po przydzieleniu procesora w procesie może wystąpić jedno z kilku zdarzeń:

- proces może zamówić operację wejścia-wyjścia, wskutek czego trafia do kolejki procesów oczekujących na wejście-wyjście;
- proces może utworzyć nowy podproces i oczekiwać na jego zakończenie;
- w wyniku przerwania proces może zostać przymusowo usunięty z procesora i przeniesiony z powrotem do kolejki procesów gotowych.

W pierwszych dwóch przypadkach proces zostanie w końcu przełączony ze stanu oczekiwania do stanu gotowości i przeniesiony do kolejki procesów gotowych. W tym cyklu proces kontynuuje działanie aż do zakończenia, kiedy to usuwa się go ze wszystkich kolejek i zwalnia jego blok kontrolny oraz przydzielone mu zasoby.

4.2.2 Planiści

Proces wędruje między różnymi kolejkami przez cały czas swego istnienia. W celu planowania działań system operacyjny musi w jakiś sposób wybierać procesy z tych kolejek. Selekcji dokonuje odpowiedni proces systemowy zwany *planistą* (programem szeregującym; ang. *scheduler*).

W systemie wsadowym często występuje więcej procesów niż można by ich natychmiast wykonać. Procesy te są przechowywane w urządzeniach pamięci masowej (zazwyczaj na dyskach), gdzie oczekują na późniejsze wykonanie. *Planista długoterminowy* (ang. *long-term scheduler*), nazywany też *planistą zadań* (ang. *job scheduler*), wybiera procesy z tej puli i ładuje je do pamięci w celu wykonania. *Planista krótkoterminowy* (ang. *short-term scheduler*), czyli *planista przydziału procesora* (ang. *CPU scheduler*), wybiera jeden proces spośród procesów gotowych do wykonania i przydziela mu procesor.

Podstawową różnicą między obydwoema planistami jest częstotliwość ich aktywnień. Planista krótkoterminowy musi bardzo często wybierać nowy proces dla procesora. Proces może działać zaledwie kilka milisekund, a potem przejść w stan oczekiwania, wydawszy zamówienie na operację wejścia-wyjścia. Często planista krótkoterminowy podejmuje działanie co najmniej raz na każde 100 ms. Ze względu na krótkie odcinki czasu między kolejnymi wykonaniami planista krótkoterminowy musi być bardzo szybki. Jeśli decyzja o wykonaniu procesu przez 100 ms zabiera 10 ms, to $10/(100 + 10) = 9\%$ pracy procesora jest zużywane (marnowane) na samo zaplanowanie działania.

Natomiast planista długoterminowy działa o wiele rzadziej. Między utworzeniem nowych procesów w systemie mogą upływać minuty. Planista długoterminowy nadzoruje *stopień wieloprogramowości*, tj. liczbę procesów w pamięci. Jeśli stopień wieloprogramowości jest stabilny, to średnia liczba utworzonych procesów musi się równać średniej liczbie procesów usuwanych z systemu. Toteż planista długoterminowy może być wywoływany tylko wtedy, gdy jakiś proces opuszcza system. Wskutek dłuższych przerw między wykonaniami planista długoterminowy może mieć więcej czasu na rozstrzyganie, który proces należy wybrać do wykonania.

Właściwe dokonanie wyboru przez planistę długoterminowego ma duże znaczenie. Większość procesów można zazwyczaj podzielić na takie, które są albo ograniczone przez wejście-wyjście, albo ograniczone przez dostęp do procesora. *Procesem ograniczonym przez wejście-wyjście* jest taki, który spędza większość czasu na wykonywaniu operacji wejścia-wyjścia, mniej zajmując się obliczeniami. Na odwrót, *proces ograniczony przez dostęp do procesora* sporadycznie generuje zamówienia na wejście-wyjście, spędzając na obliczeniach wykonywanych przez procesor znacznie więcej czasu niż proces ograniczony przez wejście-wyjście. Jest ważne, aby planista długoterminowy

dobrał dobrą mieszanek procesów (ang. process mix) zawierającą zarówno procesy ograniczone przez wejście-wyjście, jak i ograniczone przez dostęp do procesora. Jeśli wszystkie procesy są ograniczone przez wejście-wyjście, to kolejka procesów gotowych będzie prawie zawsze pusta i planista krótkoterminowy będzie miał za mało do roboty. Jeśli wszystkie procesy są ograniczone przez dostęp do procesora, to kolejka do urządzeń będzie prawie zawsze pusta i, jak poprzednio, system nie będzie zrównoważony. Najlepiej zachowuje się system, w którym jest zachowana właściwa proporcja procesów ograniczonych przez wejście-wyjście i przez procesor.

Planista długoterminowy może być w niektórych systemach nieobecny lub mocno zredukowany. Na przykład systemy z podziałem czasu często nie mają żadnego planisty długoterminowego i umieszczają po prostu każdy nowy proces w pamięci pod opieką planisty krótkoterminowego. Stabilność tych systemów zależy od fizycznych ograniczeń (takich jak liczba dostępnych terminali) lub od zdolności adaptacyjnych użytkujących je ludzi. Jeżeli działanie systemu nadmiernie się pogarsza, to niektórzy użytkownicy po prostu z niego rezygnują i zajmują się czym innym.

W niektórych systemach operacyjnych, takich jak systemy z podziałem czasu, może występować dodatkowy, pośredni poziom planowania. Takiego planistę średnioterminowego (ang. medium-term scheduler) widać na rys. 4.6. Podstawowym argumentem przemawiającym za użyciem planisty średnioterminowego jest fakt, że niekiedy może okazać się korzystne usunięcie procesów z pamięci (i z aktywnej rywalizacji o procesor) w celu zmniejszenia stopnia wieloprogramowości. Usunięte procesy można później znów wprowadzić do pamięci, a ich wykonanie kontynuować od tych miejsc, w których je przerwano. Postępowanie takie jest często nazywane *wymianą* (ang. swapping). Proces jest wymieniany, czyli wysyłany z pamięci na zewnątrz, a później wprowadzany do niej ponownie przez planistę średnioterminowego. Wymiana może być niezbędna do uzyskania lepszego doboru procesów lub wówczas, gdy



Rys. 4.6 Uzupełnienie diagramu kolejek o planowanie średnioterminowe

żądania przydziału pamięci przekraczają jej bieżący obszar i wymagają zwolnienia miejsca w pamięci. Wymianę omawiamy bardziej szczegółowo w rozdz. 8.

4.2.3 Przelaczanie kontekstu

Przelaczanie procesora do innego procesu wymaga przechowania stanu starego procesu i załadowania przechowanego stanu nowego procesu. Czynność tę nazywa się *przelaczaniem kontekstu* (ang. *context switch*). Czas przelaczania kontekstu jest czystą daniną na rzecz systemu, gdyż podczas przelaczania system nie wykonuje żadnej użytecznej pracy. Wartość czasu przelaczania kontekstu zmienia się w zależności od typu maszyny, szybkości pamięci, liczby rejestrów oraz istnienia specjalnych instrukcji (jak np. pojedyncza instrukcja do ładowania lub przechowywania wszystkich rejestrów). Na ogół waha się od 1 do 1000 µs.

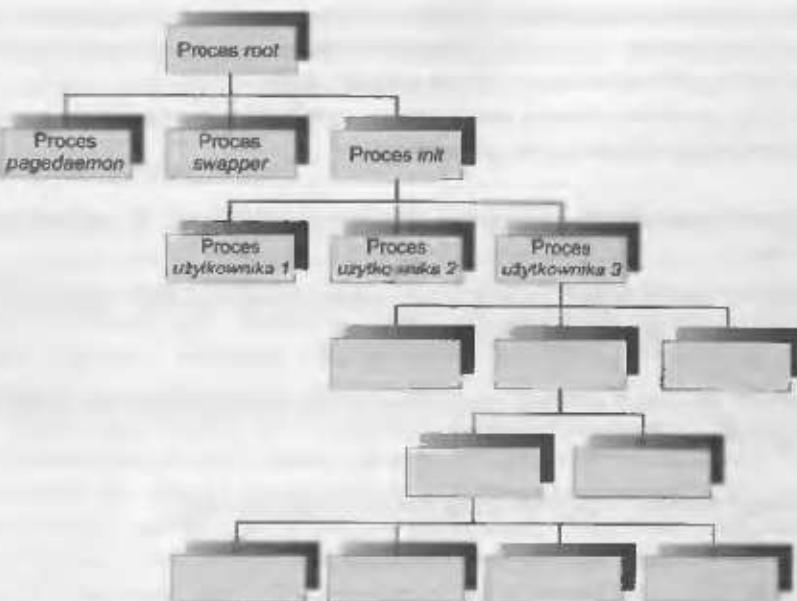
Czasy przelaczania kontekstów w dużym stopniu zależą od możliwości sprzętu. Niektóre procesory (np. DECSYSTEM-20) mają po kilka zbiorów rejestrów. Przeloczenie kontekstu sprowadza się wtedy do zmiany wartości wskaźnika bieżącego zbioru rejestrów. Oczywiście, jeśli procesów jest więcej niż zbiorów rejestrów, to system powraca do metody kopiowania stanu rejestrów do i z pamięci, tak jak poprzednio. Regułą jest również, że im bardziej jest złożony system operacyjny, tym więcej pracy musi być wykonane podczas przelaczania kontekstu. Zaawansowane techniki zarządzania pamięcią wymagają przy każdym przelaczeniu kontekstu wymiany dodatkowych danych, o czym przekonamy się w rozdz. 8. Przygotowując na przykład pamięć dla następnego zadania, należy przechować przestrzeń adresową bieżącego procesu. Sposób, w jaki się tego dokona, oraz ilość potrzebnej w tym celu pracy zależą od przyjętej w systemie operacyjnym metody zarządzania pamięcią. Jak zobaczymy w p. 4.5, przelaczanie kontekstu stało się tak wąskim gardłem, ze do unikania go, gdzie tylko to jest możliwe, korzysta się z nowych struktur (wątków).

4.3 ■ Działania na procesach

Procesy w systemie mogą być wykonywane współbieżnie oraz dynamicznie tworzone i usuwane. System operacyjny musi więc zawierać mechanizm tworzenia i kończenia procesu.

4.3.1 Tworzenie procesu

Proces może tworzyć nowe procesy za pomocą wywołania systemowego „utwórz-proces”. Proces tworzący nazywa się *procesem macierzystym* (ang. *parent process*), a utworzone przez niego nowe procesy zwą się jego *potom-*



Rys. 4.7 Drzewo procesów w typowym systemie UNIX

kami (ang. *children*). Kazdy nowy proces może tworzyć kolejne procesy, wskutek czego powstaje drzewo procesów (rys. 4.7).

Do wypełnienia swych zadań proces potrzebuje na ogół pewnych zasobów (czasu procesora, pamięci operacyjnej, plików, urządzeń wejścia-wyjścia). Gdy proces tworzy podproses, ten ostatni może otrzymać swoje zasoby bezpośrednio od systemu operacyjnego bądź zasoby podprocesu ulegającemu zawężeniu do podzbioru zasobów procesu macierzystego. Proces macierzysty może rozdzielać własne zasoby między procesy potomne albo powodować, że niektóre zasoby (np. pamięć lub pliki) będą przez potomków użytkowane wspólnie. Ograniczenie zasobów procesu potomnego do podzbioru zasobów procesu macierzystego zapobiega nadmiernemu „rozmnażaniu” procesów, które mogłyby prowadzić do przeciążenia systemu.

W uzupełnieniu rozmaitych fizycznych i logicznych zasobów, które proces otrzymuje w chwili powstania, mogą do niego dotrzeć – jako do potomka – dane początkowe (wejście) określone przez jego twórcę. Rozważmy na przykład proces, którego zadaniem jest wyświetlenie na ekranie terminalu stanu pliku – powiedzmy *F1*. Podczas tworzenia proces taki otrzyma od swojego procesu macierzystego daną wejściową będącą nazwą pliku *F1*, którą posłuży się do uzyskania potrzebnych informacji. Może on także otrzymać nazwę urządzenia wyjściowego. Niektóre systemy operacyjne przekazują

zasoby procesom potomnym. W tego rodzaju systemie nowy proces może dostać dwa otwarte pliki: *F1* i urządzenie terminalu, a wówczas jego zadaniem będzie tylko przesłanie danych między nimi.

Kiedy proces wytwarza nowy proces, wtedy w odniesieniu do jego działania praktykuje się dwojakie postępowanie:

- proces macierzysty kontynuuje działanie współbieżnie ze swoimi potomkami;
- proces macierzysty oczekuje na zakończenie działań niektórych lub wszystkich swoich procesów potomnych.

Przestrzeń adresowa nowego procesu może być zagospodarowana również na dwa sposoby:

- proces potomny staje się kopią procesu macierzystego;
- proces potomny otrzymuje nowy program.

W celu zilustrowania tych różnych implementacji rozważmy system operacyjny UNIX. W systemie UNIX każdemu procesowi jest przyporządkowany jednoznaczny *identyfikator procesu*, będący liczbą całkowitą. Nowy proces tworzy się za pomocą funkcji systemowej **fork**. Nowy proces zawiera kopię przestrzeni adresowej procesu pierwotnego. Ułatwia to procesowi macierzystemu komunikację z procesem potomnym. Oba procesy (przodek i potomek) kontynuują działanie od instrukcji występującej po **fork** z jedną różnicą – wartością funkcji **fork** przekazywaną nowemu (potomnemu) procesowi jest zero, natomiast proces macierzysty otrzymuje niezerowy identyfikator swojego potomka.

Zazwyczaj po funkcji **fork** w jednym z dwóch procesów jest wykonywana funkcja systemowa **execve**, która powoduje zastąpienie zawartości pamięci procesu przez nowy program. Funkcja systemowa **execve** ładuje plik binarny do pamięci (niszcząc obraz pamięci programu zawierającego wywołanie **execve**) i rozpoczyna jego wykonanie. W ten sposób oba procesy mogą się skontaktować, a potem potoczyć się osobnymi torami. Proces macierzysty może wtedy tworzyć kolejnych potomków lub – jeśli podczas działania procesu potomnego nie ma on nic do roboty – może wywołać funkcję systemową **wait**, aby usunąć się z kolejki procesów gotowych do chwili zakończenia działania procesu potomnego.

Na odmiانę, system operacyjny DEC VMS tworzy nowy proces, umieszcza w nim określony program i rozpoczyna jego działanie. W systemie operacyjnym Microsoft Windows NT istnieją oba modele: przestrzeń adresowa

procesu macierzystego może zostać podwojona albo proces macierzysty może określić nazwę programu, który system operacyjny powinien umieścić w przestrzeni adresowej nowego procesu.

4.3.2 Kończenie procesu

Proces kończy się wówczas, gdy wykona swoją ostatnią instrukcję i za pomocą wywołania funkcji `exit` poprosi system operacyjny, aby go usunął. W tej chwili proces może przekazać dane (wyjście) do procesu macierzystego (za pośrednictwem funkcji systemowej `wait`^{*}). Wszystkie zasoby procesu, w tym pamięć fizyczna i wirtualna, otwarte pliki i bufora wejścia-wyjścia zostają odebrane przez system operacyjny.

Oprócz tego istnieją inne przyczyny zakończenia procesu. Proces może spowodować zakończenie innego procesu za pomocą odpowiedniej funkcji systemowej (np. `abort`). Zazwyczaj funkcję tę może wywołać tylko przodek procesu, który ma być zakończony; w przeciwnym razie użytkownicy mogliby likwidować sobie wzajemnie dowolne zadania. Zauważmy, że proces macierzysty musi znać identyfikatory swoich potomków. Dlatego też, gdy jakiś proces tworzy nowy proces, wówczas identyfikator nowo utworzonego procesu jest przekazywany do procesu macierzystego.

Proces macierzysty może zakończyć któryś ze swoich procesów potomnych z różnych przyczyn, takich jak:

- potomek nadużył któregoś z przydzielonych mu zasobów;
- wykonywane przez potomka zadanie stało się zbędne;
- proces macierzysty kończy się, a w tej sytuacji system operacyjny nie pozwala potomkowi na dalsze działanie.

Aby ustalić, że wystąpił pierwszy przypadek, proces macierzysty musi mieć mechanizm sprawdzania stanu swoich potomków.

Wiele systemów, w tym system VMS, nie pozwala na istnienie potomka po zakończeniu procesu macierzystego. W takich systemach zakończenie procesu (zwykle lub wyjątkowe) wymusza zakończenie wszystkich jego potomków. Zjawisko to nazywa się *kończeniem kaskadowym* (ang. *cascading termination*) i jest na ogół inicjowane przez system operacyjny.

Aby zilustrować wykonanie i zakończenie procesu, rozważmy znów system UNIX. W systemie UNIX proces może zakończyć pracę za pomocą funkcji systemowej `exit`, a jego proces macierzysty może oczekiwac na to zdarze-

* Użytej w procesie macierzystym. – Przyp. tłum.

nie, wywoławszy funkcję systemową `wait`. Funkcja ta przekazuje identyfikator procesu potomnego, który skończył działanie, dzięki czemu przodek otrzymuje informację o tym, który – z być może wielu jego potomków – zakończył pracę. Jeżeli jednak kończy się proces macierzysty, to wszystkie jego procesy potomne są kończone przez system operacyjny. Gdyby zabrakło procesu macierzystego, system UNIX nie potrafiłby określić, komu przekazać sprawozdanie z działania procesu potomnego.

4.4 ■ Procesy współpracujące

Procesy współbieżne wykonywane w systemie operacyjnym mogą być niezależne lub mogą ze sobą współpracować. Proces jest *niezależny* (ang. *independent*), jeżeli nie może oddziaływać na inne procesy wykonywane w systemie, a te z kolei nie mogą wpływać na jego działanie. Mówiąc jaśniej, dowolny proces jest niezależny, jeśli nie dzieli żadnych danych (tymczasowych lub trwałych) z żadnym innym procesem. O procesie powiemy natomiast, że jest *współpracującym* (ang. *cooperating*), jeżeli może on wpływać na inne procesy w systemie lub inne procesy mogą oddziaływać na niego. Dokładniej – dowolny proces dzielący dane z innymi procesami jest procesem współpracującym.

Na rzecz środowiska umoczniającego współpracę procesów przedmawia kilka powodów:

- **Dzielenie informacji:** Ponieważ kilku użytkowników może być zainteresowanych tymi samymi informacjami (np. wspólnym użytkowaniem pliku), musimy zapewnić możliwość współbieżnego dostępu do tego rodzaju zasobów.
- **Przyspieszanie obliczeń:** W celu przyspieszenia wykonywania jakiegoś zadania możemy je podzielić na podzadania, z których każde będzie wykonywane równolegle z pozostałymi. Zauważmy, że w tym wypadku przyspieszenie będzie możliwe tylko wtedy, gdy komputer ma wiele elementów przetwarzających (takich jak procesory lub kanały wejścia-wyjścia).
- **Modularność:** Możemy chcieć konstruować system w sposób modularny, dzieląc go na osobne procesy, tak jak to przedstawiliśmy w rozdz. 3.
- **Wygoda:** Nawet indywidualny użytkownik może mieć wiele zadań do wykonania w jednym czasie. Użytkownik może na przykład równolegle zajmować się edytowaniem, drukowaniem i kompilowaniem.

Współbieżność działań wymagających współpracy między procesami pociąga za sobą konieczność opracowania mechanizmów umożliwiających procesom wzajemne komunikowanie się (p. 4.6) i synchronizowanie działań (rozdz. 6).

Aby zilustrować koncepcję procesów współpracujących, rozważmy zagadnienie producenta-konsumenta (ang. *producer-consumer problem*), będące popularnym wzorcem współpracujących ze sobą procesów. Proces *producent* wytwarza informacje, które zużywa proces *konsument*. Na przykład program drukujący wytwarza znaki, które są pobierane przez program obsługi drukarki. Kompilator może „produkować” kod asemblerowy, który jest „konsumowany” przez asembler. Z kolei asembler może wytwarzac moduły programu wynikowego „konsumowane” przez program ładujący.

Aby umożliwić producentowi i konsumentowi działanie współbieżne, musimy dysponować buforem jednostek, który będzie mógł być zapełniany przez producenta i opróżniany przez konsumenta. Podczas gdy producent tworzy pewną jednostkę, konsument może zużywać inną. Procesy producenta i konsumenta muszą podlegać synchronizacji, aby konsument nie próbował konsumować tych jednostek, które nie zostały jeszcze wyprodukowane. W takiej sytuacji konsument musi czekać na wyprodukowanie tego, co chce konsumować.

Problem producenta-konsumenta z *nieograniczonym buforem* (ang. *unbounded-buffer*) nie ma żadnych praktycznych ograniczeń na rozmiar bufora. Może się zdarzać, że konsument musi czekać na nowe jednostki, ale producent może je produkować nieustannie. W problemie producenta-konsumenta z *ograniczonym buforem* (ang. *bounded-buffer*) zakkłada się, że bufor ma ustaloną długość. W tym przypadku konsument musi czekać, jeśli bufor jest pusty, a producent musi czekać, jeśli bufor jest pełny.

Bufor może być udostępniany przez system operacyjny za pośrednictwem komunikacji międzyprocesowej (IPC – p. 4.6) lub może być jawnie zakodowany przez programistę aplikacji, który używa w tym celu pamięci dzielonej. Przedstawimy teraz rozwiązanie problemu z ograniczonym buforem i z użyciem pamięci dzielonej. Procesy producenta i konsumenta korzystają z następujących wspólnych zmiennych:

```
var n;  
type jednostka = ...;  
var bufor: array [0..n-1] of jednostka;  
we, wy: 0..n-1;
```

przy czym *we* i *wy* mają nadaną wartość początkową 0. Dzielony bufor jest zrealizowany jako tablica cykliczna z dwoma wskaźnikami logicznymi: *we* oraz *wy*. Zmienna *we* wskazuje na następne wolne miejsce w buforze, zmien-

na wy wskazuje na pierwsze zajęte miejsce w buforze. Bufor jest pusty, gdy $we = wy$, bufor jest pełny, gdy $we + 1 \bmod n = wy$.

Poniżej są podane kody procesów producenta i konsumenta. Instrukcja *nic* jest instrukcją pustą, tzn. *while warunek do nic* powoduje po prostu nieustanne sprawdzanie warunku dopóty, dopóki nie będzie on falszywy.

Proces producenta ma lokalną zmienną *nastp*, w której przechowuje nowo produkowaną jednostkę:

```

repeat
    produkuj jednostkę w nastp
    while we + 1 mod n = wy do nic;
        bufor[we] := nastp;
        we := we + 1 mod n;
    until false;

```

Proces konsumenta ma lokalną zmienną *nastk*, w której jest przechowywana jednostka do skonsumowania:

```

repeat
    while we = wy do nic;
    nastk := bufor[wy];
    wy := wy + 1 mod n;
    konsumuj jednostkę z nastk
until false;

```

Dany schemat powala na użycie co najwyżej $n - 1$ jednostek w buforze w tym samym czasie. Znalezienie rozwiązania umożliwiającego używanie n jednostek w buforze w tym samym czasie pozostawiamy czytelnikom jako zadanie.

W rozdziale 6 omówimy szczegółowo sposoby skutecznego implementowania synchronizacji procesów współpracujących w środowisku pamięci dzielonej.

4.5 ■ Wątki

Przypomnijmy, że proces jest określony za pomocą używanych przez siebie zasobów i miejsca, w którym działa. Istnieje jednak wiele sytuacji, w których zagospodarowanie zasobów byłoby lepsze, gdyby można ich było używać

wspólnic i współbieżnie. Sytuacje takie są podobne do przypadku, w którym funkcja systemowa **fork**, użyta z nową wartością licznika rozkazów, powoduje podjęcie nowego wątku sterowania przebiegającego w obrębie tej samej przestrzeni adresowej. Koncepcja ta jest do tego stopnia użyteczna, że kilka nowych systemów operacyjnych realizuje ją w postaci mechanizmu wątków.

4.5.1 Struktura wątku

Wątek (ang. *thread*), nazywany niekiedy *procesem lekkim* (ang. *lightweight process* – LWP), jest podstawową jednostką wykorzystania procesora, w skład której wchodzą: licznik rozkazów, zbiór rejestrów i obszar stosu. Wątek współużytkuje wraz z innymi równorzędnymi wątkami sekcję kodu, sekcję danych oraz takie zasoby systemu operacyjnego, jak otwarte pliki i sygnały, co łącznie określa się jako *zadanie* (ang. *task*). Proces tradycyjny, czyli *ciężki* (ang. *heavyweight*), jest równoważny zadaniu z jednym wątkiem. Zadanie nie robi nic, jeśli nie ma w nim ani jednego wątku; z kolei wątek może przebiegać w dokładnie jednym zadaniu. Daleko posunięte dzielenie zasobów powoduje, że przełączanie procesora między równorzędnymi wątkami, jak również tworzenie wątków, jest tanie w porównaniu z przełączaniem kontekstu między tradycyjnymi procesami ciężkimi. Choć przełączanie kontekstu między wątkami nadal wymaga przełączania zbioru rejestrów, jednak nie trzeba wykonywać żadnych prac związanych z zarządzaniem pamięcią. Jak w każdym środowisku przetwarzania równoległego, podział procesu na wątki może prowadzić do zagadnień sterowania współbieżnością i konieczności stosowania sekcji krytycznych lub zamków.

W niektórych systemach zrealizowano też *wątki poziomu użytkownika* (ang. *user-level threads*), z których korzysta się za pośrednictwem wywołań bibliotecznych zamiast odwołań do systemu, dzięki czemu przełączanie wątków nie wymaga wzywania systemu operacyjnego i przerwań związanych z przechodzeniem do jego jądra. Przełączanie między wątkami poziomu użytkownika może być wykonywane niezależnie od systemu operacyjnego, może więc się odbywać bardzo szybko. Blokowanie jednego wątku i przełączanie do innego wątku jest zatem rozsądnym rozwiązaniem problemu wydajnego obsługiwanego przez serwer wielu zamówień. Wątki poziomu użytkownika mają jednak też swoje wady. Na przykład, jeśli jądro jest jednowątkowe, to każdy wątek poziomu użytkownika odwołujący się do systemu będzie powodował oczekiwanie całego zadania na zakończenie wywołania systemowego.

Aby zrozumieć działanie wątków, porównamy sterowanie wieloma wątkami ze sterowaniem wieloma procesami. Każdy z wielu procesów działa niezależnie od innych, tzn. ma własny licznik rozkazów, rejestr stosu i przestrzeń adresową. Taka organizacja jest wygodna wówczas, gdy zadania wy-

konywane przez procesy nie są ze sobą powiązane. Wiele procesów może też wykonywać to samo zadanie. Na przykład w realizacji sieciowego systemu plików wiele procesów może dostarczać danych do odległych maszyn. Jednak do obsługi tego samego przedsięwzięcia wydajniej jest zastosować jeden proces z wieloma wątkami. W realizacji wieloprocesowej każdy proces wykonuje ten sam program, ale we własnej pamięci i przy użyciu własnych plików. Jeden proces wielowątkowy zużywa mniej zasobów niż z wielokrotnione procesy, biorąc pod uwagę pamięć operacyjną, otwarte pliki i planowanie procesora. I tak na przykład w trakcie rozwoju systemu Solaris doszło do przepisania demonów sieciowych na wątki jądra, aby znacznie zwiększyć wydajność pewnych funkcji sieciowych.

Działanie wątków pod wieloma względami przypomina działanie procesów. Wątki mogą znajdować się w jednym z kilku stanów: gotowości, zablokowania, aktywności lub kończenia. Podobnie jak procesy, wątki użytkują wspólnie jednostkę centralną i w danej chwili tylko jeden wątek jest aktywny (wykonywany). Wykonanie wątku w procesie przebiega sekwencyjnie, a każdy wątek ma własny stos i licznik rozkazów. Wątki mogą tworzyć wątki potomne i mogą blokować się do czasu zakończenia wywołań systemowych. Jeśli jeden wątek jest zablokowany, to może działać inny wątek. Jednak w odróżnieniu od procesów wątki nie są niezależne od siebie. Ponieważ wszystkie wątki mają dostęp do każdego adresu w zadaniu, dany wątek może czytać i zapisywać stosy dowolnych innych wątków*. Struktura taka nie umożliwia ochrony na poziomie wątków. Ochrona ta nie powinna jednakże być konieczna. Podczas gdy procesy pochodzą od różnych użytkowników i mogą być do siebie wrogo usposobione, zadanie z wieloma wątkami może należeć tylko do jednego użytkownika. W tym przypadku wątki zostaną prawdopodobnie tak zaprojektowane, aby pomagać sobie wzajemnie, więc nie będą wymagały wzajemnej przed sobą ochrony. Na rysunku 4.8 widać zadanie z wieloma wątkami.

Powróćmy do przykładu zablokowanego procesu serwera plików w modelu jednoprocesowym. W tym scenariuszu żaden inny proces usługowy nie może działać dopóty, dopóki pierwszy proces nie zostanie odblokowany. Przeciwnie jest w przypadku zadania zawierającego wiele wątków: w czasie gdy jeden z wątków usługowych jest zablokowany i czeka, w tym samym zadaniu może działać inny wątek. W danym zastosowaniu współpraca wielu wątków będących elementami jednego zadania daje przewagę w postaci większej przepustowości i polepszonej wydajności. W innych zastosowaniach, takich jak problem producenta-konsumenta, wymaga się dzielenia wspólnego bufora, więc również mogą one zyskać na skorzystaniu z właści-

* W ramach jednego procesu. – Przyp. tłum.



Rys. 4.8 Wiele wątków w jednym zadaniu

wości wątków – producent i konsument mogą być wątkami jednego zadania. Przełączanie między nimi będzie się odbywać tanim kosztem, a w systemie wieloprocesorowym oba wątki będą mogły działać równolegle na dwu procesorach w celu osiągnięcia maksymalnej wydajności.

Wątki tworzą mechanizm umożliwiający procesom sekwencyjnym wykonywanie blokowanych wywołań systemowych przy jednoczesnym osiąganiu równoległości. Aby zilustrować zaletę tego mechanizmu, zastanowimy się nad zaprogramowaniem serwera plików w systemie, w którym wątki nie są osiągalne. Widzieliśmy już, że w jednowątkowym serwerze plików proces serwera przed podjęciem nowej pracy musi obsłużyć poprzednie zamówienie do końca. Jeśli zamówienie to obejmuje oczekивание na dostęp do dysku, to podczas tego oczekiwania procesor pozostaje bezczynny. Z tego powodu liczba zamówień, które można przetworzyć w ciągu sekundy, jest znacznie mniejsza niż przy wykonywaniu równoległym. Nie mogąc skorzystać z wielu wątków, projektant systemu dążący do minimalizacji spowolnienia działania jednowątkowych procesów będzie musiał naśladować strukturę wątków równoległych za pomocą procesów ciężkich. Jest to możliwe, lecz za cenę złożonej, niesekwencyjnej struktury programu.

Abstrakcja, jaką tworzy grupa procesów lekkich, polega na tym, że wiele wątków sterowania jest powiązanych z pewną liczbą dzielonych zasobów.

Istnieje sporo alternatywnych sposobów traktowania wątków, wspomnijmy krótko o niektórych z nich. Wątki mogą być obsługiwane przez jądro (jak w systemach operacyjnych Mach i OS/2). W tym przypadku systemy zawierają zbiór funkcji podobnych do tych, które obsługują procesy. Inne podejście polega na tworzeniu wątków powyżej jądra systemu za pomocą zbioru funkcji bibliotecznych wykonywanych na poziomie użytkownika (takie rozwiązanie przyjęto w projekcie Andrew z uniwersytetu CMU).

Co przemawia za przyjęciem w systemie operacyjnym jednej wersji lub drugiej? Wątki poziomu użytkownika nie angażują jądra, więc dają się przełączać szybciej niż wątki realizowane przez jądro. Jednakże dowolne wywołanie systemu operacyjnego może powodować oczekивание całego procesu, ponieważ jądro planuje tylko procesy (nic nie wiedząc o wątkach), a proces czekający nie otrzymuje przydziału czasu procesora. Planowanie może odbywać się też niesprawiedliwie. Rozważmy dwa procesy, z których jeden ma tylko jeden wątek (proces *a*), a drugi ma 100 wątków (proces *b*). Oba procesy otrzymają na ogół tę samą liczbę kwantów czasu, wskutek czego wątek w procesie *a* będzie działał 100 razy szybciej niż wątek w procesie *b*. W systemach, w których wątki są organizowane przez jądro, przełączanie wątków zabiera więcej czasu, gdyż zajmuje się tym jądro (za pośrednictwem przerwań). Każdy wątek może jednak podlegać indywidualnemu planowaniu, dzięki czemu proces *b* otrzyma 100 razy więcej czasu procesora niż proces *a*. Ponadto proces *b* może mieć wykonywanych współbieżnie 100 funkcji systemowych, więc może zrobić znacznie więcej niż taki sam proces działający w systemie z wątkami tylko na poziomie użytkownika.

Kompromisem między tymi dwoma podejściami do organizacji wątków jest zastosowanie w niektórych systemach rozwiązania mieszanego, w którym są zrealizowane zarówno wątki poziomu użytkownika, jak i wątki w jądrze. Systemem takim jest Solaris 2, omówiony w następnym punkcie.

Wątki zyskują na popularności, ponieważ – mając pewne cechy procesów ciężkich – są efektywniejsze w działaniu. Jest wiele zastosowań, w których taka kombinacja jest pozytywna. Na przykład niektóre implementacje jądra systemu UNIX są jednozadaniowe – kod jądra może wykonywać w danej chwili tylko jedno zadanie. Unika się w ten sposób wielu problemów, takich jak synchronizacja dostępu do danych (blokowanie struktur danych na czas ich zmieniania), ponieważ tylko jeden proces ma prawo wykonać takie zmiany. Natomiast system Mach jest wielowątkowy, jego jądro może obsługiwać wiele zamówień jednocześnie. W tym przypadku wątki działają synchronicznie – nowy wątek z tej samej grupy zadziała dopiero wtedy, gdy wątek bieżący odda sterowanie. Oczywiście wątek bieżący powinien oddawać sterowanie tylko w takiej chwili, w której nie zmienia on wspólnych danych. W sys-

mach z asynchronicznymi wątkami musi istnieć jawnny mechanizm blokowania danych, taki jak w systemach, w których wiele procesów dzieli dane. Synchronizację procesów omówimy w rozdz. 6.

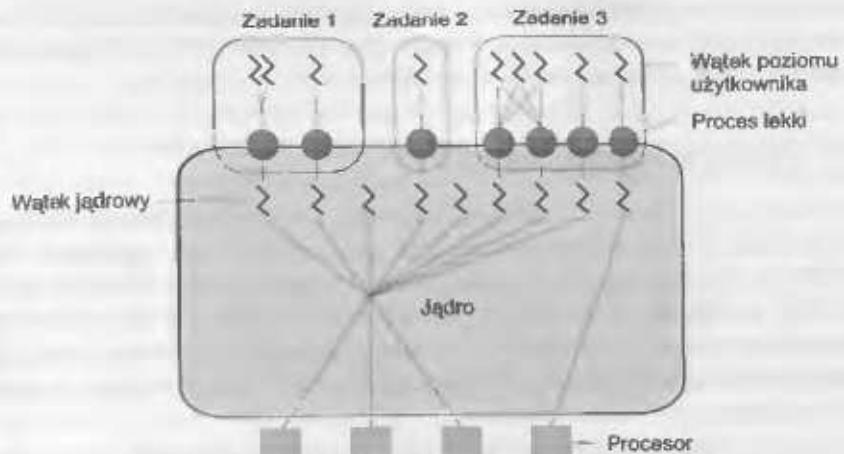
4.5.2 Przykład – system Solaris 2

Przyjrzenie się systemowi wątków w aktualnie używanym systemie operacyjnym powinno nam pomóc w jaśniejszym zrozumieniu wielu zagadnień. Wybraliśmy w tym celu system Solaris 2, czyli wersję systemu UNIX, w której do 1992 r. istniały tylko tradycyjne procesy ciężkie. Przekształcono go w nowoczesny system operacyjny z wątkami dostępnymi na poziomie jądra i użytkownika, symetrycznym wieloprzetwarzaniem i planowaniem w czasie rzeczywistym.

System Solaris 2 realizuje wątki na poziomie użytkownika, jak to było opisane w p. 4.5.1. Można z nich korzystać za pomocą biblioteki funkcji służących do ich tworzenia i planowania, przy czym jądro nie ma o takich wątkach żadnych danych. Zakłada się, że w systemie Solaris 2 o cykle procesora mogą rywalizować potencjalnie tysiące wątków.

System Solaris 2 definiuje również pośredni poziom wątków. Między wątkami poziomu użytkownika a wątkami poziomu jądra występują procesy lekkie. Każde zadanie (nadal nazywane „procesem” w terminologii systemu SunOS) zawiera co najmniej jeden proces lekki (LWP). Procesami lekkimi zawiaduje biblioteka wątków. Wątkami poziomu użytkownika obdziela się procesy LWP procesu, przy czym wykonywaniem zadania zajmują się tylko te wątki poziomu użytkownika, które są aktualnie podłączone do procesów LWP. Pozostałe są zablokowane lub czekają na proces LWP, który umożliwi im działanie.

Wszystkie operacje w jądrze są wykonywane za pomocą standardowych wątków poziomu jądra. Dla każdego procesu lekkiego (LWP) istnieje wątek na poziomie jądra, istnieją też wątki jądra działające na zamówienie jądra i nie powiązane z procesami lekkimi (np. wątek do obsługi zamówień dyskowych). Cały system wątków jest przedstawiony na rys. 4.9. Jedynymi obiektami podlegającymi planowaniu w systemie są wątki poziomu jądra (rozdz. 5). Niektórymi z wątków poziomu jądra obdziela się naprzemiennie procesory systemu, inne wątki są powiązane na stałe z określonymi procesorami. Na przykład wątek jądrowy skojarzony ze sterownikiem urządzenia przydzielonego do konkretnego procesora działa tylko na tym procesorze. Wątek może być również przypięty (ang. pinned) do procesora na życzenie. Wątek taki działa tylko na procesorze, który został mu przydzielony (zob. skrajny wątek po prawej stronie na rys. 4.9).



Rys. 4.9 Wątki w systemie Solaris 2

Rozważmy działanie takiego systemu. Dowolne zadanie może mieć wiele wątków poziomu użytkownika. Wątki te mogą być planowane i przełączane na realizowane przez jądro procesy lekkie bez interwencji jądra. Zablokowanie jednego z wątków użytkownika i podjęcie działania przez inny wątek nie wymaga przełączania kontekstu, więc wątki poziomu użytkownika są niezwykle wydajne.

Zapleczeniem wątków poziomu użytkownika są procesy lekkie. Każdy proces lekki (LWP) jest przyłączony dokładnie do jednego wątku poziomu jądra, natomiast wszystkie wątki poziomu użytkownika są od jądra niezależne. W zadaniu może być wiele procesów LWP, lecz są one potrzebne tylko wówczas, gdy wątki muszą kontaktować się jądem. Na przykład każdy wątek, który ma się współbieżnie blokować na wywołaniu systemowym, wymaga jednego procesu lekkiego. Rozpatrzmy pięć różnych zamówień na czytanie pliku, które mogą wystąpić jednocześnie. Będzie wtedy potrzebnych pięć procesów lekkich, gdyż każdy z nich może oczekiwac w jądrze na zakończenie operacji wejścia-wyjścia. Gdyby zadanie miało tylko cztery procesy lekkie, wówczas piąte zamówienie musiałoby czekać na powrót któregoś z procesów lekkich z jądra. Dodanie szóstego procesu LWP nie spowodowałoby żadnego zysku, gdyby pracy starczało tylko dla pięciu procesów.

Wątki jądrowe są planowane przez planistę jądra i wykonywane w procesorze (lub procesorach) systemu. Jeśli wątek jądrowy ulega zablokowaniu (zwykle z powodu oczekiwania na zakończenie operacji wejścia-wyjścia), to procesor może podjąć wykonywanie innego wątku jądra. Jeżeli zablokowany wątek działał na zamówienie procesu lekkiego, to zablokowaniu ulega rów-

nież proces lekki. Idąc za tym łańcuchem, powstaje konieczność zablokowania także aktualnie przyłączonego do procesu lekkiego wątku poziomu użytkownika. Jeśli zadanie zawierające ten wątek ma tylko jeden proces lekki, to do chwili zakończenia operacji wejścia-wyjścia zablokowaniu ulega całe zadanie. Jest to takie samo zachowanie jak w przypadku procesu w starszej wersji systemu operacyjnego.

W systemie Solaris 2 zadanie nie musi się jednak blokować w oczekiwaniu na zakończenie operacji wejścia-wyjścia. Zadanie może mieć wiele procesów lekkich – mimo że jeden z nich ulega zablokowaniu, inne mogą kontynuować działanie w ramach zadania.

Na zakończenie tego przykładu przyjrzymy się zasobom, które są potrzebne wątkom każdego z tych rodzajów.

- Wątek jądrowy ma jedynie małą strukturę danych i stos. Przelaczanie wątków jądrowych nie wymaga zmiany informacji dotyczących dostępu do pamięci, jest więc stosunkowo szybkie.
- Proces lekki (LWP) zawiera blok kontrolny procesu z danymi rejestrówymi, informacjami rozliczeniowymi i informacjami dotyczącymi pamięci. Przelaczanie procesów lekkich wymaga zatem nieco pracy i jest dość wolne.
- Wątek poziomu użytkownika wymaga tylko stosu i licznika rozkazów – nie są mu potrzebne żadne zasoby jądra. Jądro nie jest angażowane w planowanie wątków poziomu użytkownika, tak więc ich przelaczanie jest szybkie. Mogą istnieć tysiące wątków poziomu użytkownika, a jedynie, co będzie widoczne dla jądra, to procesy lekkie w procesie realizującym wątki tego poziomu.

4.6 ■ Komunikacja międzyprocesowa

W punkcie 4.4 pokazaliśmy, w jaki sposób współpracujące procesy mogą się komunikować w środowisku ze wspólną pamięcią. Wymagało to dzielenia przez procesy wspólnego bufora oraz jawnego napisania kodu implementującego ten bufor przez osobę programującą aplikację. Inna możliwość uzyskania tego samego efektu polega na tym, aby system operacyjny zawierał środki pozwalające współpracującym procesom na kontaktowanie się ze sobą, czyli udogodnienia komunikacji międzyprocesowej (ang. *interprocess-communication – IPC*).

Oprogramowanie komunikacji międzyprocesowej tworzy mechanizm umożliwiający procesom łączność i synchronizowanie działań. Komunikację międzyprocesową najlepiej realizuje się za pomocą systemu przekazywania komunikatów. System komunikatów można zdefiniować na wiele sposobów.

Systemy przekazywania komunikatów mają jeszcze inne zalety, co zostanie uwidocznione w rozdz. 16.

Zauważmy, że pamięć dzielona^{*} i schematy łączności systemów komunikatów nie wykluczają się wzajemnie i mogą być stosowane jednocześnie w obrębie jednego systemu operacyjnego lub nawet w jednym procesie.

4.6.1 Struktura podstawowa

Zadaniem systemu komunikatów jest umożliwienie procesom wzajemnej komunikacji bez uciekania się do zmiennych dzielonych. W skład narzędzi komunikacji międzyprocesowej wchodzą dwie podstawowe operacje: nadaj (*komunikat*) i odbierz (*komunikat*)^{**}.

Komunikaty wysypane przez proces mogą mieć stałą lub zmienną długość. Dopuszczenie tylko komunikatów o stałej długości upraszcza ich fizyczną implementację. Ograniczenie to utrudnia jednak programowanie. Komunikaty o zmiennej długości wymagają natomiast większych nakładów na fizyczną implementację, za to są łatwiejsze w programowaniu.

Jeśli procesy P i Q chcą się ze sobą komunikować, to muszą wzajemnie nadawać i odbierać komunikaty – musi istnieć między nimi *iącze komunikacyjne* (ang. *communication link*). Łącze to można zrealizować na wiele sposobów. Koncentrujemy się tutaj nie na fizycznej implementacji łącza (w rodzaju pamięci dzielonej, szyny sprzętowej lub sieci – zajmujemy się tym w rozdz. 15), lecz na zagadnieniach jego logicznej implementacji, na jego cechach logicznych. Oto podstawowe pytania związane z implementacją:

- Jak ustanawia się połączenia?
- Czy łącze może być powiązane z więcej niż dwoma procesami?
- Ile może być łączy między każdą parą procesów?
- Jaka jest pojemność łącza? Czy łącze ma jakiś obszar buforowy? Jeśli tak, to jak duży?
- Jaki jest rozmiar komunikatów? Czy łącze akceptuje komunikaty zmiennej, czy stałej długości?
- Czy łącze jest jednokierunkowe, czy dwukierunkowe? Jeśli istnieje łącze między P i Q , to czy komunikaty mogą przepływać tylko w jedną stronę (np. tylko od P do Q), czy w obie?

^{*}Czyli wspólnie użytkowana. – Przyp. tłum.

^{**}Ang. *send* i *receive*; w użyciu są też inne odpowiedniki tych angielskich terminów, mianowicie *wyslij* i *przymij*. – Przyp. tłum.

Pojęcie jednokierunkowości (ang. *unidirectional*) wymaga dokładniejszego potraktowania, ponieważ łącze może wiązać więcej niż dwa procesy. Mówimy przeto, że łącze jest jednokierunkowe tylko wtedy, gdy każdy podłączony do niego proces może albo nadawać, albo odbierać, ale nie może wykonywać obu czynności na przemian, oraz gdy każde łącze ma przynajmniej jeden proces odbiorczy.

Ponadto istnieje kilka metod logicznej implementacji łącza i operacji nadaj-odbierz:

- komunikacja bezpośrednia lub pośrednia;
- komunikacja symetryczna lub asymetryczna;
- buforowanie automatyczne lub jawnie;
- wysyłanie na zasadzie tworzenia kopii lub odsyłacza;
- komunikaty stałej lub zmiennej długości.

Dalej zajmiemy się omówieniem tych odmian systemów komunikatów.

4.6.2 Nazewnictwo

Procesy, które chcą się ze sobą komunikować, muszą mieć możliwość zwarcia się do siebie. Mogą to robić za pomocą komunikacji bezpośrednią (ang. *direct communication*) lub komunikacji pośredniej (ang. *indirect communication*). Obie metody przedstawiamy poniżej.

4.6.2.1 Komunikacja bezpośrednią

W komunikacji bezpośrednią każdy proces, który chce się komunikować, musi jawnie nazwać odbiorcę lub nadawcę uczestniczącego w tej wymianie informacji. W tym przypadku operacje elementarne nadaj i odbierz są zdefiniowane następująco:

nadaj (P , komunikat) – nadaj komunikat do procesu P

odbierz (Q , komunikat) – odbierz komunikat od procesu Q

Łącze komunikacyjne w tym schemacie ma następujące własności:

- łącze jest ustanawiane automatycznie między parą procesów, które mają się komunikować; do wzajemnego komunikowania się wystarczy, aby procesy znały swoje identyfikatory;

* W oryginale: *send* i *receive*. – Przyp. tłum.

- łącze dotyczy dokładnie dwu procesów;
- między każdą parą procesów istnieje dokładnie jedno łącze;
- łącze może być jednokierunkowe, choć zazwyczaj jest dwukierunkowe.

Aby to zilustrować, przedstawimy rozwiązanie problemu producenta i konsumenta. W celu umożliwienia wspólnie działających procesów producenta i konsumenta pozwalamy producentowi wytwarzać pewną jednostkę, podczas gdy konsument zużywa inną jednostkę. Gdy producent skończy wytwarzanie jednostki, wysyła ją konsumentowi za pomocą operacji **nadaj**. Konsument pobiera jednostkę za pomocą operacji **odbierz**. Jeśli jednostka nie została jeszcze wytworzona, to proces konsumenta musi zaczekać na jej powstanie. Proces producenta jest zdefiniowany następująco:

```
repeat
  ...
  wytwarzaj jednostkę w nastp
  ...
  nadaj (konsument, nastp);
until false.
```

Proces konsumenta jest zdefiniowany za pomocą instrukcji:

```
repeat
  odbierz (producent, nastk);
  ...
  konsumuj jednostkę z nastk
  ...
until false.
```

Powyższy schemat wykazuje symetrię adresowania; zarówno proces nadawczy, jak i odbiorczy w celu utrzymania ze sobą łączności muszą wzajemnie używać nazw. Istnieje też asymetryczny wariant adresowania. Tylko nadawca nazywa odbiorcę, a od odbiorcy nie wymaga się znajomości nazwy nadawcy. W tym przypadku operacje **nadaj** i **odbierz** określają się następująco:

- **nadaj (*P, komunikat*)** – nadaj komunikat do procesu *P*;
- **odbierz (*id, komunikat*)** – odbierz komunikat od dowolnego procesu; pod *id* zostanie podstawiona nazwa procesu, od którego nadszedł komunikat.

Wadą obu schematów (symetrycznego i asymetrycznego) jest ograniczona modularność wynikowych definicji procesów. Zmiana nazwy jednego procesu może pociągać za sobą konieczność zweryfikowania definicji wszystkich innych procesów. Należy zlokalizować wszystkie miejsca wystąpienia starej nazwy, aby móc ją zastąpić nową nazwą. Jest to sytuacja niepozdana, biorąc pod uwagę niezależną komplikację.

4.6.2.2 Komunikacja pośrednia

W komunikacji pośredniej komunikaty są nadawane i odbierane za pośrednictwem skrzynek pocztowych (ang. *mailboxes*), nazywanych także *portami* (ang. *ports*). Abstrakcyjna skrzynka pocztowa jest obiektem, w którym procesy mogą umieszczać komunikaty i z którego komunikaty mogą być pobierane. Każda skrzynka pocztowa ma jednoznaczną identyfikację. W tej metodzie proces może komunikować się z innym procesem za pomocą różnych skrzynek pocztowych. Możliwość komunikacji między dwoma procesami istnieje tylko wtedy, gdy mają one jakąś wspólną skrzynkę pocztową. Definicje operacji nadaj i odbierz przybierają postać:

nadaj (*A, komunikat*) – nadaj komunikat do skrzynki *A*
 odbierz (*A, komunikat*) – odbierz komunikat ze skrzynki *A*

Łącze komunikacyjne ma w tym schemacie następujące własności:

- łączce między dwoma procesami jest ustanawiane tylko wtedy, gdy dzielą one jakąś skrzynkę pocztową;
- łączce może być związane z więcej niż dwoma procesami;
- każda para komunikujących się procesów może mieć kilka różnych łącz, z których kazde odpowiada jakiejś skrzynce pocztowej;
- łączce może być jednokierunkowe lub dwukierunkowe.

Załóżmy teraz, że procesy P_1 , P_2 i P_3 mają wspólną skrzynkę pocztową A . Proces P_1 wysyła komunikat do A . natomiast każdy z procesów P_2 i P_3 kieruje do skrzynki A operację odbierz. Który proces otrzyma komunikat nadany przez P_1 ? Zagadnienie to można rozwiązywać rozmaicie:

- zezwalać na łączce tylko między dwoma procesami;
- pozwalać tylko co najwyżej jednemu procesowi na wykonywanie w danej chwili operacji odbierz;
- dopuścić, aby system wybierał dowolnie proces, do którego dotrze komunikat (tzn. komunikat otrzyma albo P_1 , albo P_2 , ale nie oba procesy); system może informować nadawcę o ostatecznym odbiorcy.

Skrzynka pocztowa może być własnością procesu albo systemu. Jeśli skrzynka należy do procesu (tzn. jest przypisana lub zdefiniowana jako część procesu), to rozróżnia się jej właściciela (który za jej pośrednictwem może tylko odbierać komunikaty) i użytkownika (który może tylko nadawać komunikaty do danej skrzynki). Ponieważ każda skrzynka ma jednoznacznie określonego właściciela, jest oczywiste, kto powinien odebrać komunikat nadesłany do danej skrzynki. Kiedy proces będący właścicielem skrzynki pocztowej kończy działanie, skrzynka znika. Proces, który po tym zdarzeniu próbowałby nadal wysyłać komunikaty do tej skrzynki, musi zostać powiadomiony, że skrzynka już nie istnieje (na zasadzie obsługi sytuacji wyjątkowych, opisanej w p. 4.6.4).

Istnieje wiele sposobów wyznaczania właściciela i użytkowników skrzynki pocztowej. Jednym z nich jest umożliwienie procesowi zadeklarowania zmiennej typu skrzynka pocztowa. Proces deklarujący skrzynkę pocztową staje się jej właścicielem. Każdy inny proces, który zna nazwę tej skrzynki, może zostać jej użytkownikiem.

Skrzynka pocztowa należąca do systemu operacyjnego istnieje bez inicjatywy procesu. Jest niezależna i nie przydziela się jej do żadnego konkretnego procesu. System operacyjny dostarcza mechanizmów pozwalających na:

- tworzenie nowej skrzynki;
- nadawanie i odbieranie komunikatów za pośrednictwem skrzynki;
- likwidowanie skrzynki.

Proces, na którego zamówienie jest tworzona nowa skrzynka pocztowa, staje się jej właścicielem na zasadzie domyślności. Początkowo właściciel jest jedynym procesem, który może odbierać komunikaty poprzez tę skrzynkę. Wszakże przywilej własności i odbierania komunikatów może zostać przekazany innym procesom za pomocą odpowiedniej funkcji systemowej. Takie postępowanie może, rzecz jasna, prowadzić do powiększenia liczby odbiorców dla każdej skrzynki. Procesy mogą również dzielić skrzynkę pocztową w wyniku tworzenia nowych procesów. Na przykład, jeśli proces P utworzy skrzynkę A , a następnie utworzy nowy proces Q , to P i Q będą wspólnie korzystać ze skrzynki A . Ponieważ wszystkie procesy, mające jakieś prawa dojęcia do skrzynki, mogą kiedyś zakończyć swoje działanie, po pewnym czasie skrzynka pocztowa może stać się niedostępna dla żadnego procesu. W tym przypadku system operacyjny powinien odzyskać cały obszar, który zajmowała dana skrzynka. Zadanie to przybiera postać swoistej formy odzyskiwania nieużytków (p. 10.3.5), kiedy to do przeszukiwania i zwalniania nie używanych obszarów pamięci stosuje się osobne operacje.

4.6.3 Buforowanie

Łącze ma pewną pojemność określającą liczbę komunikatów, które mogą w nim czasowo przebywać. Cechą ta może być postrzegana jako kolejka komunikatów przypisanych do łączego. Są trzy podstawowe metody implementacji takiej kolejki.

- **Pojemność zerowa:** Maksymalna długość kolejki wynosi 0, czyli łącze nie dopuszcza, by czekał w nim jakikolwiek komunikat. W tym przypadku nadawca musi czekać, aż odbiorca odbierze komunikat. Aby można było przesyłać komunikaty, oba procesy muszą być zsynchronizowane. Synchronizację taką nazywa się *rendez-vous*.
- **Pojemność ograniczona:** Kolejka ma skończoną długość n , może w niej zatem pozostawać co najwyżej n komunikatów. Jeśli w chwili nadania nowego komunikatu kolejka nie jest pełna, to nowy komunikat zostaje w niej umieszczony (przez skopiowanie komunikatu lub zapamiętanie wskaźnika do niego) i nadawca może kontynuować działanie bez czekania. Jednak kiedy łącze z powodu ograniczonej pojemności ulegnie zapełnieniu, wtedy nadawca będzie musiał być opóźniany, aż zwolni się miejsce w kolejce.
- **Pojemność nieograniczona:** Kolejka ma potencjalnie nieskończoną długość; może w niej oczekiwać dowolna liczba komunikatów. Nadawca nigdy nie jest opóźniany.

Metoda pojemności zerowej bywa czasami nazywana systemem komunikatów bez buforowania; w pozostałych metodach stosuje się automatyczne buforowanie.

Zwraca uwagę fakt, że w przypadkach niezerowych pojemności proces nie wie, czy komunikat dotarł do celu z chwilą zakończenia operacji *nadaj*. Jeśli ta informacja jest istotna dla dalszych obliczeń, to nadawca musi jawnie skontaktować się z odbiorcą, aby sprawdzić, czy ten ostatni otrzymał przesyłkę. Założymy na przykład, że proces P wysyła komunikat do procesu Q , przy czym dalsze jego działanie może nastąpić dopiero po odebraniu komunikatu. Proces P wykonuje instrukcję

nadaj (Q, komunikat);
odbierz (Q, komunikat);

Proces Q wykonuje instrukcję

odbierz (P, komunikat);
nadaj (P, „potwierdzenie”);

O takich procesach mówimy, że komunikują się *asynchronicznie*.

Istnieją specjalne przypadki, których nie da się dopasować wprost do żadnej z dotąd omówionych kategorii.

- Proces nadający komunikat nigdy nie jest opóźniany. Jeśli jednak odbiorca nie zdąży przyjąć komunikatu, zanim nadawca nie wyśle następnego komunikatu, to pierwszy komunikat jest tracony (ginie). Zaletą tego schematu jest to, iż długie komunikaty nie muszą być kopowane więcej niż jeden raz. Główną wadę stanowi utrudnienie programowania. Procesy wymagają jawniej synchronizacji, aby żaden z komunikatów nie został zagubiony oraz by nadawca i odbiorca nie manipulowali jednocześnie na buforze komunikatów.
- Proces nadający komunikat jest opóźniany do czasu otrzymania odpowiedzi. Schemat taki zaadaptowano w systemie operacyjnym Thoth. Komunikaty tego systemu mają stałą długość (osiem słów). Proces *P* jest po nadaniu komunikatu wstrzymywany do czasu, aż proces odbiorczy otrzyma komunikat i wyśle ośmiosłowną odpowiedź za pomocą operacji *odpowiedź (P, komunikat)*^{*}. Komunikat z odpowiedzią zapisuje się w tym samym buforze co komunikat nadany na początku. Jedyna różnica między operacją *nadaj* a operacją *odpowiedź* polega na tym, że operacja *nadaj* powoduje wstrzymywanie procesu nadawczego, a operacja *odpowiedź* pozwala natychmiast kontynuować oba procesy.

Tę synchroniczną metodę komunikacji daje się łatwo rozszerzyć do postaci ogólnego systemu *wywołań procedur zdolnych*, zwanego w skrócie RPC (ang. *remote procedure call*). Systemy RPC oparto na spostrzeżeniu, że wywołanie procedury lub funkcji w systemie jednoprocesorowym działa dokładnie tak jak system komunikatów, w którym nadawca ulega zablokowaniu do czasu otrzymania odpowiedzi. Komunikat przybiera zatem postać wywołania podprogramu, a komunikat powrotny zawiera wynik obliczenia wykonanego przez podprogram. Następnym logicznym krokiem w obranym kierunku jest umożliwienie procesom współbieżnym wzajemnego wywoływanie się jako podprogramów za pomocą systemu RPC. W istocie, w rozdz. 16 przekonamy się, że wywołania RPC można zastosować w odniesieniu do procesów wykonywanych na oddzielnych komputerach, umożliwiając harmonijną współpracę wielu komputerów.

4.6.4 Sytuacje wyjątkowe

System komunikatów jest szczególnie użyteczny w środowisku rozproszonym, w którym procesy mogą rezydować w różnych instalacjach (maszynach). W takim środowisku prawdopodobieństwo wystąpienia błędu podczas

* W oryginale operacja ta nosi nazwę *reply* – Przyp. tłum.

komunikacji (i przetwarzania) jest znacznie większe niż w jednej maszynie. Komunikaty w osobnych maszynach są zwykle implementowane za pomocą pamięci dzielonej. Wystąpienie błędu powoduje załamanie całego systemu. Natomiast w środowisku rozproszonym komunikaty są na ogół obsługiwane za pomocą linii komunikacyjnych i uszkodzenie jednej instalacji (lub łącza) niekoniecznie musi powodować awarię całego systemu.

Niezależnie od tego, czy uszkodzenie pojawi się w systemie scentralizowanym czy rozproszonym, powoduje ono podjęcie pewnych działań naprawczych (obsługa sytuacji wyjątkowych). Ponizej dokonamy krótkiego przeglądu sytuacji wyjątkowych, z którymi system powinien sobie radzić w kontekście wymiany komunikatów.

4.6.4.1 Zakończenie procesu

Może się zdarzyć, że nadawca lub odbiorca zakończy działanie przed zakończeniem przetwarzania komunikatu. Pozostaną wówczas komunikaty, których nikt nigdy nie odbierze. Lub jakieś procesy będą czekać na komunikaty, które nigdy nie zostaną wysłane. Rozważamy tu dwa przypadki:

1. Proces odbiorczy P może czekać na komunikat od procesu Q , który zakończył działanie. Jeśli nie podejmie się żadnych kroków, to proces P zostanie zablokowany na zawsze. W tym przypadku system może zakończyć proces P albo powiadomić go, że proces Q zakończył swą pracę.
2. Proces P może wysłać komunikat do procesu Q , który już zakończył działanie. Przy użyciu schematu automatycznego buforowania nie powoduje to żadnej szkody – proces P po prostu kontynuuje działanie. Jeśli proces P potrzebuje się upewnić, że komunikat został przetworzony przez proces Q , to powinien jawnie poprosić o potwierdzenie. W przypadku systemu bezbuforowego proces P zostanie zablokowany na zawsze. Tak jak w przypadku 1, system może albo zakończyć proces P , albo powiadomić go, że proces Q już nie istnieje.

4.6.4.2 Utrata komunikatów

Komunikat nadany przez proces P do procesu Q może zaginąć w sieci komunikacyjnej z powodu awarii sprzętu lub linii komunikacyjnej. Istnieją trzy podstawowe metody postępowania w takich przypadkach:

1. System operacyjny jest odpowiedzialny za wykrywanie takich zdarzeń i za ponowne nadanie komunikatu.
2. Proces nadawczy jest odpowiedzialny za wykrycie takiego zdarzenia i powtórne przesłanie komunikatu, jeśli mu na tym zależy.

3. System operacyjny odpowiada za wykrywanie takich zdarzeń. Zawiadamia on proces nadawczy, że komunikat zaginął. Proces nadawczy może postąpić według własnych potrzeb.

Wykrywanie zagubienia komunikatów nie zawsze jest niezbędne. W rzeczywistości w niektórych protokołach sieciowych zaznacza się, że przesyłanie komunikatów może być zawodne, a w innych gwarantuje się niezawodność (zob. rozdz. 15). Użytkownik musi zadecydować (powiadając system lub programując to samodzielnie), czy mają być wykrywane tego rodzaju zdarzenia.

Jak wykryć utratę komunikatu? Najpowszechniej stosowaną metodą jest *odliczanie czasu* (ang. *timeout*). Po wysłaniu komunikatu następuje zawsze nadanie komunikatu z odpowiedzią potwierdzającą odbiór. System operacyjny lub proces użytkownika może wtedy określić przedział czasu, w którym oczekuje naadejście komunikatu potwierdzającego. Jeśli ten czas upłynie przed nadejściem potwierdzenia, to system operacyjny (lub proces) może przyjąć, że komunikat został zagubiony i wysłać go powtórnie. Jest jednak możliwe, że komunikat nie zaginął, lecz podróżował przez sieć nieco dłużej niż zakładano. Musi więc istnieć mechanizm umożliwiający wyodrębnianie różnych typów takich komunikatów. Bardziej szczegółowo omówimy to zagadnienie w rozdz. 16.

4.6.4.3 Zniekształcenia komunikatów

Komunikat może dojść do celu zniekształcony po drodze (np. wskutek zakłóceń w kanale komunikacyjnym). Ten przypadek jest podobny do przypadku zagubienia komunikatu. Zazwyczaj system operacyjny wyśle powtórnie komunikat w pierwotnej postaci. Do wykrywania tego rodzaju błędów używa się powszechnie kodów sprawdzających występowanie błędów (z zastosowaniem sum kontrolnych, sprawdzania parzystości lub cyklicznej kontroli nadmiarowej – CRC).

4.6.5 Przykład – system Mach

Jako przykład systemu operacyjnego opartego na komunikatach omówimy system operacyjny Mach, opracowany w Carnegie-Mellon University. Jądro systemu Mach umożliwia tworzenie i likwidowanie wielu zadań, które są podobne do procesów, lecz mają wiele wątków sterowania. Większa część komunikacji w systemie Mach, w tym większość wywołań systemowych i cały przepływ informacji między zadaniami, odbywa się na zasadzie przesyłania komunikatów. Komunikaty są nadawane i odbierane przez skrzynki pocztowe (nazywane w systemie Mach portami).

Nawet wywołania systemowe są wykonywane za pomocą komunikatów. Przy tworzeniu każdego zadania powstają także dwie specjalne skrzynki

pocztowe – skrzynka jądra (ang. *kernel mailbox*) i skrzynka zawiadomień (ang. *notify mailbox*)*. Skrzynka jądra jest używana przez jądro do komunikacji z zadaniem. Do portu zawiadomień jądro wysyła zawiadomienia o występujących zdarzeniach. Do przesyłania komunikatów są potrzebne tylko trzy funkcje systemowe. Funkcja *msg_send* wysyła komunikat do skrzynki pocztowej. Do odbierania komunikatu służy funkcja *msg_receive*. Do uaktywnienia procedur zdalnych służy funkcja *msg_rpc*, która wysyła komunikat i czeka na dokładnie jeden komunikat z odpowiedzią od nadawcy.

Funkcja systemowa *port_allocate* tworzy nową skrzynkę pocztową i przydziela pamięć na kolejkę jej komunikatów. Przyjmowana zastępco, maksymalna długość kolejki komunikatów wynosi osiem komunikatów. Zadanie tworzące skrzynkę zostaje jej właścicielem. Właściciel dostaje również prawo odbioru komunikatów ze swojej skrzynki. Za każdym razem tylko jedno zadanie może mieć prawo własności lub odbierania komunikatów ze skrzynki, lecz w razie potrzeby prawa te można przekazać do innych zadań.

Kolejka do skrzynki pocztowej jest na początku pusta. Nadawanie komunikatów do skrzynki powoduje, że są one do niej przekopiowywane. Komunikaty są umieszczane w kolejce według priorytetów. Wszystkie komunikaty mają taki sam priorytet. System Mach gwarantuje, że komunikaty pochodzące od tego samego nadawcy zostaną ustawione w kolejce w porządku FIFO, ale nie zapewnia uporządkowania całkowitego. Na przykład komunikaty nadane przez dwóch nadawców mogą trafić do kolejki w dowolnym porządku.

Komunikaty składają się z nagłówków o stałej długości, po których następują różnej długości porcje danych. Nagłówek zawiera długość komunikatu i nazwy dwóch skrzynek pocztowych. Jedna z nich określa skrzynkę docelową, do której ma trafić nadawany komunikat. Na ogół wątek nadawczy oczekuje odpowiedzi; nazwa skrzynki pocztowej nadawcy jest przekazywana do zadania odbierającego komunikaty, które może jej użyć jako „adresu zwrotnego” w celu wysłania komunikatu potwierdzającego odbiór.

Zmienna część komunikatu jest listą jednostek danych o określonych typach. Każdy element tej listy ma typ, rozmiar i wartość. Określone w komunikacie typy obiektów mają istotne znaczenie, ponieważ za pomocą komunikatów mogą być przekazywane obiekty zdefiniowane w systemie operacyjnym – jak prawa własności lub odbioru, stany zadania i segmenty pamięci.

Operacje nadawania i odbierania komunikatów charakteryzują się dużą elastycznością. W chwili nadania komunikatu skrzynka pocztowa może być zapelniona. Jeśli skrzynka nie jest pełna, to następuje skopiowanie do niej komunikatu i wątek nadawczy kontynuuje działanie. Jeśli skrzynka jest pełna, wątek nadawczy ma do wyboru cztery możliwości:

* Przez innych autorów nazywane odpowiednio: portem jądra (ang. *kernel port*) lub portem procesu (ang. *process port*) oraz portem wyjątków (ang. *exception port*). – Przyp. tłum.

1. Czekać przez nieokreślony czas do chwili powstania wolnego miejsca w skrzynce.
2. Czekać co najwyżej n milisekund.
3. W ogóle nie czekać, powracając natychmiast do pracy.
4. Czasowo przechować komunikat. Komunikat można oddać na przechowanie systemowi operacyjnemu nawet wtedy, gdy skrzynka pocztowa, do której został nadany, jest zajęta. System operacyjny, znalazły po pewnym czasie sposobność na umieszczenie komunikatu w skrzynce, informuje o tym nadawcę. Dla danego wątku nadawczego w każdej chwili może być w ten sposób opóźniany tylko jeden komunikat.

Ostatnią możliwość przewidziano z myślą o takich zadaniach usługowych, jak wykonywanie programu obsługi drukarki wierszowej. Po wykonaniu zamówienia zadania tego rodzaju muszą wysyłać do swoich zleceniodawców jednorazowe potwierdzenia i jednocześnie muszą obsługiwać innych klientów, nawet jeśli skrzynka na odpowiedź dla danego klienta jest zapelniona.

Operacja odbiorcza musi określać, z której skrzynki lub zbioru skrzynek ma być odebrany komunikat. *Zbiór skrzynek* (ang. *mailbox set*)^{*} jest grupą skrzynek pocztowych zadeklarowanych w zadaniu i traktowanych przez zadanie jak jedna skrzynka. Wątki zadania mogą odbierać informacje tylko z tej skrzynki (lub zbioru skrzynek), z której zadanie ma prawo odbioru. Funkcja systemowa *port_status* przekazuje liczbę komunikatów znajdujących się w danej skrzynce. Operacja odbiorcza może żądać odbioru komunikatu z dowolnej skrzynki w zbiorze skrzynek lub z określonej (nazwanej) skrzynki. Jeśli nie ma komunikatów czekających na odebranie, to wątek odbiorczy może czekać przez czas nieograniczony lub co najwyżej n milisekund, lub nie czekać wcale.

System Mach skonstruowano specjalnie dla systemów rozproszonych, którymi zajmujemy się w rozdz. 15-18, lecz nadaje się on również dla systemów jednoprocessorowych. Podstawową trudność w systemie komunikatów stanowiła jego słaba wydolność powodowana kopiowaniem komunikatu najpierw od nadawcy do skrzynki pocztowej, a potem ze skrzynki do odbiorcy. W systemie Mach spróbowano uniknąć dublowania operacji kopiowania przez zastosowanie technik zarządzania pamięcią wirtualną (rozdz. 9). W rzeczywistości Mach odwzorowuje w przestrzeni adresowej odbiorcy przestrzeni adresowej zawierającej komunikat nadawcy. Sam komunikat nigdy nie jest naprawdę kopiowany. Taka technika zarządzania komunikatami znacznie poprawia wydajność, można

^{*} Inni autorzy częściej używają przy opisie systemu Mach terminu „port” zamiast „zbior skrzynek” i posługują się określeniem „zbior portów” (ang. *port set*). — Przyp. tłum.

ją jednak stosować tylko do komunikatów wewnętrzsystemowych. System operacyjny Mach omówiono szczegółowo w rozdziale pomieszczonego w naszym komputerze dostępnym w sieci usług WWW^{*}.

4.6.6 Przykład – system Windows NT

System operacyjny Windows NT jest przykładem nowoczesnego projektu, w którym zastosowano modularyzację w celu zwiększenia funkcjonalności i zmniejszenia czasu potrzebnego do wdrażania nowych cech. System NT umożliwia korzystanie z różnych środowisk operacyjnych, nazywanych podsystemami, z którymi programy użytkowe kontaktują się za pomocą mechanizmu przekazywania komunikatów. Programy użytkowe można uważać za klientów podsystemu usługodawczego (serwera) NT.

Komunikaty w systemie NT są udostępniane za pomocą tzw. *udogodnienia wywoływanego procedur lokalnych* (ang. *Local Procedure Call Facility* – LPC). Udogodnienie nazywane lokalnym wywołaniem procedury jest używane w systemie NT do komunikacji między procesami znajdującymi się w tej samej maszynie. Przypomina ono szeroko używany mechanizm RPC standar-dowego zdalnego wywołania procedury, lecz jest zoptymalizowane i specy-ficzne dla systemu NT. Do nawiązania i utrzymywania połączenia między dwoma procesami używa się w systemie NT, podobnie jak w systemie Mach, obiektu portu. Kazdemu klientowi, odwołującemu się do podsystemu, jest potrzebny kanał komunikacyjny, który jest udostępniany przez obiekt portu i nigdy nie podlega dziedziczeniu. W systemie NT stosuje się dwa typy portów: porty łączące i porty komunikacyjne – będące w istocie tym samym, lecz w zależności od sposobu wykorzystania zaopatrywane w różne nazwy. Porty łączące są *obiektami nazwanymi* (ang. *named objects*) – więcej informacji o obiektach systemu NT zawiera rozdz. 23 – widocznymi dla wszystkich pro-cesów i będącymi dla aplikacji środkiem do ustanawiania kanału komunika-cyjnego. Komunikacja taka działa, jak następuje:

- Klient zaopatruje się w uchwyt do obiektu portu łączącego podsystemu.
- Klient wysyła prośbę o połączenie.
- Serwer tworzy dwa prywatne porty komunikacyjne i przekazuje klientowi uchwyt do jednego z nich.

* Czytelnicy polscy mogą zdobyć więcej wiadomości o interesujących rozwiązaniach systemu Mach z lektury rozdziału 16 wcześniejszego wydania tej książki, jak również z książek: *Rozproszone systemy operacyjne* (rozdz. 8) oraz *Systemy rozproszone – podstawy i pro- jektowanie* (p. 18.2). – Przyp. tłum.

- Klient i serwer korzystają z odpowiednich uchwytów portowych w celu wysyłania komunikatów lub przywołań oraz nasłuchiwanie odpowiedzi.

W systemie NT stosuje się trzy sposoby przekazywania komunikatów za pomocą portu określonego przez klienta podczas ustanawiania kanału. Sposób najprostszy, używany do małych komunikatów, polega na posłużeniu się portową kolejką komunikatów jako pamięcią tymczasową i przekopiowaniu komunikatu z jednego procesu do drugiego. Za pomocą tej metody można wysyłać komunikaty nie dłuższe niż 256 bajtów.

Jeżeli klient musi nadać dłuższy komunikat, to przekazuje go za pomocą obiektu sekcji (pamięci dzielonej). Podczas tworzenia kanału klient musi zdecydować, czy będzie chciał wysłać duży komunikat. Jeśli klient uzna, że chce wysyłać wielkie komunikaty, to zamawia utworzenie obiektu sekcji¹. Podobnie, gdy serwer zdecyduje, że odpowiedzi będą duże, wówczas także tworzy obiekt sekcji. W celu użycia obiektu sekcji wysyla się mały komunikat, który zawiera wskaźnik i informację o rozmiarze obiektu sekcji. Metoda ta jest nieco bardziej skomplikowana niż pierwsza metoda, ale pozwala na uniknięcie kopowania danych. W obu przypadkach można skorzystać z mechanizmu przywołania, jeśli klient lub serwer nie mogą natychmiast sprostać zamówieniu. Mechanizm przywołania pozwala im na asynchroniczną obsługę komunikatów.

Jedną z wad stosowania przekazywania komunikatów do wykonywania typowych działań, takich jak funkcje graficzne, jest gorsza wydajność od tej, która jest osiągana w systemie obchodząącym się bez przekazywania komunikatów (tj. w przypadku pamięci dzielonej). Aby zwiększyć wydajność, w rdzennym środowisku systemu NT (Win32) stosuje się trzecią metodę przekazywania komunikatów, zwaną *szymbkimi wywołaniami LPC* (ang. *quick LPC*). Klient wysyła zamówienie połączenia do portu łączacego serwera, zaznaczając, że będzie używał szybkich wywołań LPC. Serwer deleguje do obsługi zamówień osobny wątek, obiekt sekcji wielkości 64 KB oraz obiekt pary zdarzeń. Wówczas komunikaty będą przekazywane za pośrednictwem obiektu sekcji, a synchronizacja będzie dokonywana za pomocą obiektu pary zdarzeń. Eliminuje się w ten sposób kopowanie komunikatów, koszty związane z używaniem obiektu portu oraz koszty określania, który wątek klienta go używa, gdyż każdemu wątkowi klienta jest przyporządkowany osobny wątek serwera. Jądro daje także wydzielonemu wątkowi priorytet przy planowaniu przydziału procesora. Oczywiście wadą tej metody jest to, że zużywa ona więcej zasobów niż poprzednie dwie metody. Po-

¹ Ta bardzo ogólna nazwa określa fragment pamięci. – Przyp. tłum.

nieważ z przekazywaniem komunikatów wiążą się pewne koszty, system NT może „zestawiać” kilka komunikatów w jeden „komunikat”, aby redukować te koszty. Więcej informacji na temat systemu Windows NT można znaleźć w rozdz. 23.

4.7 ■ Podsumowanie

Proces jest to wykonywany program. W trakcie postępowania procesu zachodzą zmiany w jego stanie. Stan procesu jest określony przez bieżące działanie procesu. Każdy proces może być w jednym z następujących stanów: nowy, gotowy, aktywny, oczekiwania lub zakończony. Każdy proces jest reprezentowany w systemie operacyjnym przez blok kontrolny procesu.

Proces, który nie jest wykonywany, zostaje umieszczony w oczekującej na coś kolejce. W systemie operacyjnym istnieją dwie główne klasy kolejek: kolejki zamówień wejścia-wyjścia oraz kolejka procesów gotowych. Kolejka procesów gotowych zawiera wszystkie procesy gotowe do działania i oczekujące na jednostkę centralną. Każdy proces jest reprezentowany przez blok kontrolny procesu, a bloki te można połączyć ze sobą tak, aby utworzyły kolejkę procesów gotowych. Planowanie długoterminowe (zadań) polega na dopuszczaniu procesów do rywalizacji o procesor. Na planowanie długoterminowe na ogół mają znaczny wpływ kwestie przydziału urządzeń, a zwłaszcza zarządzania pamięcią. Planowanie krótkoterminowe (procesora) polega na wybieraniu któregoś procesu z kolejki procesów gotowych.

Procesy w systemie mogą działać współbieżnie. Istnieje kilka powodów uzasadniających współbieżność działań: wspólne użytkowanie informacji, przyspieszanie obliczeń, modularność i wygoda. Wykonywanie współbieżne wymaga mechanizmów tworzenia i usuwania procesów.

Procesy w systemie operacyjnym mogą być wykonywane niezależnie lub mogą ze sobą współpracować. Współpracujące ze sobą procesy muszą mieć środki wzajemnej komunikacji. Istnieją w zasadzie dwa wzajemnie się uzupełniające schematy komunikacyjne: pamięć dzielona oraz systemy komunikatów. W metodzie pamięci dzielonej komunikujące się procesy muszą wspólnie użytkować pewne zmienne. Zakłada się, że za pomocą tych wspólnych zmiennych procesy będą wymieniać informacje. W systemach pamięci dzielonej odpowiedzialność za umożliwianie komunikacji spada na programistów aplikacji – system operacyjny powinien tylko zapewniać pamięć dzielowaną. W systemie komunikatów procesy mogą wymieniać komunikaty. Odpowiedzialność za umożliwianie komunikacji ciąży na samym systemie operacyjnym. Oba schematy nie wykluczają się wzajemnie i mogą być stosowane jednocześnie w jednym systemie operacyjnym.

Procesy współpracujące, które bezpośrednio dzielą logiczną przestrzeń adresową można implementować w postaci procesów lekkich, czyli wątków. Wątek jest podstawową jednostką wykorzystania procesora, która dzieli z innymi równorzędnymi wątkami swoją sekcję kodu i danych oraz zasoby systemu operacyjnego – łącznie nazywa się to zadaniem. Zadanie nie wykonuje niczego, jeżeli nie ma w nim żadnych wątków, natomiast wątek musi należeć dokładnie do jednego zadania. Owa znaczna wspólnota powoduje potanicie przełączania procesora do poszczególnych wątków w porównaniu z przełączaniem kontekstu przy użyciu procesów ciężkich.

■ Ćwiczenia

- 4.1 Kilka popularnych mikrokomputerowych systemów operacyjnych nie zawiera żadnych środków organizacji współbieżności albo zawiera ich niewiele. Omów podstawowe komplikacje systemu operacyjnego spowodowane przez przetwarzanie współbieżne.
- 4.2 Wyjaśnij różnice między planowaniem krótkoterminowym, średnioterminowym i długoterminowym.
- 4.3 Komputer DECSYSTEM-20 ma wiele zbiorów rejestrów. Opisz, na czym będzie polegało przełączenie kontekstu, jeżeli nowy kontekst jest już załadowany do jednego ze zbioru rejestrów. Co należy jeszcze zrobić, jeśli nowy kontekst jest przechowywany w pamięci, a nie w zbiorze rejestrów, ponieważ wszystkie zbiory rejestrów są w użyciu?
- 4.4 Jakie są dwie zalety wątków, których nie ma mnogość procesów? Jaka jest główna wada wątków? Zaproponuj zastosowanie, w którym użycie wątków mogłoby przynieść pożytek, i takie, w którym korzystanie z wątków zakończyłoby się niepowodzeniem.
- 4.5 Jakie zasoby są zużywane do tworzenia wątku? W czym różnią się one od tych, których używa się do tworzenia procesów?
- 4.6 Opisz czynności podejmowane przez jądro podczas przełączania kontekstu:
 - (a) z użyciem wątków;
 - (b) z użyciem procesów.
- 4.7 Na czym polegają różnice między wątkami poziomu użytkownika a wątkami realizowanymi przez jądro? W jakich warunkach jeden typ wątków jest „lepszy” niż drugi?

- 4.8 Poprawny algorytm producenta-konsumenta zamieszczony w p. 4.4 umożliwia jednocześnie zapelnienie tylko $n - 1$ pozycji w buforze. Zmień ten algorytm w taki sposób, aby wszystkie pozycje bufora mogły być w pełni wykorzystywane.
- 4.9 Rozważ schemat komunikacji międzyprocesowej z użyciem skrzynek pocztowych:
- Załóżmy, że proces P chce zaczekać na dwa komunikaty – jeden ze skrzynki A i drugi ze skrzynki B . Jaki ciąg operacji nadaj i odbierz powinien wykonać?
 - Jaki ciąg operacji nadaj i odbierz powinien wykonać proces P , gdyby chciał zaczekać na jeden komunikat, obojętnie z której skrzynki: A lub B (lub z obu naraz)?
 - Operacja odbierz powoduje czekanie procesu do chwili, w której skrzynka przestaje być pusta. Opracuj schemat umożliwiający procesowi oczekивание до chwili, w której skrzynka stanie się pusta, lub uzasadnij dlaczego schemat taki nie może istnieć.
- 4.10 Rozważ system operacyjny realizujący zarówno komunikację międzyprocesową (IPC), jak i wywoływanie procedur zdalnych (RPC). Podaj przykłady zagadnień, które można rozwiązać za pomocą każdego z tych schematów. Wyjaśnij, dlaczego metoda, którą obierzesz do rozwiązania owych zagadnień, jest najlepsza w danej sytuacji.

Uwagi bibliograficzne

Wezbrane prace nad implementowaniem wątków na poziomie użytkownika omówili Doeppner [114]. Zagadnienia wydajności wątków przedstawiono w artykule Andersona i in. [10]. Anderson i in. kontynuowali te prace, oszacowując wydajność wątków na poziomie użytkownika, wspieranych przez jądro. Marsh i in. [271] omówili pierwszorzędne wątki poziomu użytkownika. Bershad i in. [35] opisali połączenie wątków z wywołaniami RPC. Draves i in. [121] rozważyli zastosowanie kontynuacji do implementowania zarządzania wątkami i komunikacji w systemach operacyjnych.

Wielowątkowy system operacyjny IBM OS/2 działa na komputerach osobistych (zob. Kogan i Rawson [220]). Wysoko wydajne jądro *Synthesis* również korzysta z wątków (Massalin i Pu [272]). Implementację wątków w systemie Mach opisano w artykule Tevariana i in. [422]. Programowanie z użyciem wątków opisał Birrel [39]. Uruchamianie wielowątkowych aplikacji

cji jest trudnym zagadnieniem pozostającym w sferze badań. Caswell i Black [65] opracowali program uruchomieniowy w systemie Mach.

Strukturę wątków w systemie Solaris 2 firmy Sun Microsystems opisano w pracy Ekyholta i in. [128]. Kwestią wątków na poziomie użytkownika zajął się Stein i Shaw [405]. Peacock [326] przedstawił wielowątkowość systemu plików w systemie Solaris 2.

Zagadnienie komunikacji międzyprocesowej było omawiane przez Brincha Hansena [54] w odniesieniu do systemu RC 4000. Środki komunikacji międzyprocesowej w systemie operacyjnym Thoth omówili Cheriton i in. [71]; w systemie operacyjnym Accent opisali je Rashid i Robertson [343], a w systemie operacyjnym Mach – Accetta i in. [2]. Schlichting i Schneider [378] omówili elementarne operacje asynchronicznego przekazywania komunikatów. Środki komunikacji międzyprocesowej zrealizowane na poziomie użytkownika opisano w artykule Bershadiego i in. [35].

Rozważania na temat implementacji wywołań procedur zdalnych (RPC) przedstawili Birrell i Nelson [40]. Projekt niezawodnego mechanizmu RPC zaprezentowali Shrivastava i Panzieri [388]. Przegląd wywołań RPC opracowali Tay i Ananda [420]. Stankovic [402] i Staunstrup [404] dokonali porównania zagadnień wywołań procedur i łączności za pomocą przekazywania komunikatów.

Rozdział 5

PLANOWANIE PRZYDZIAŁU PRÓCESORA

Planowanie przydziału procesora leży u podstaw wieloprogramowych systemów operacyjnych. Dzięki przełączaniu procesora do procesów, system operacyjny może zwiększyć wydajność komputera. W tym rozdziale wprowadzamy podstawowe pojęcia dotyczące planowania i przedstawiamy kilka różnych algorytmów przydziału procesora. Omawiamy również zagadnienie doboru algorytmu do konkretnego systemu.

5.1 ■ Pojęcia podstawowe

Celem wieloprogramowania jest maksymalizowanie wykorzystania jednostki centralnej przez stałe utrzymywanie w działaniu pewnej liczby procesów. W systemie jednoprocesorowym nigdy nie jest wykonywany więcej niż jeden proces. Jeżeli istnieje więcej procesów, to pozostałe muszą czekać do chwili, gdy procesor zostanie zwolniony i będzie można ponownie przydzielić go jednemu z nich.

Idea wieloprogramowania jest dość prosta. Proces jest wykonywany do chwili, w której będzie musiał czekać, zazwyczaj na zakończenie jakiejś operacji wejścia-wyjścia. W prostym systemie komputerowym procesor jest wtedy po prostu bezczynny. Cały ów czas oczekiwania jest marnowany; nie wykonuje się żadnej użytecznej pracy. Za pomocą wieloprogramowania dąży się do produktywnego spożyciania tego czasu. W pamięci operacyjnej znajduje się kilka procesów jednocześnie. Kiedy jakiś proces musi czekać, wtedy system operacyjny odbiera mu procesor i oddaje do dyspozycji innego procesu. Schemat ten jest powtarzany. Zawsze gdy któryś proces musi czekać, inny proces może korzystać z procesora.

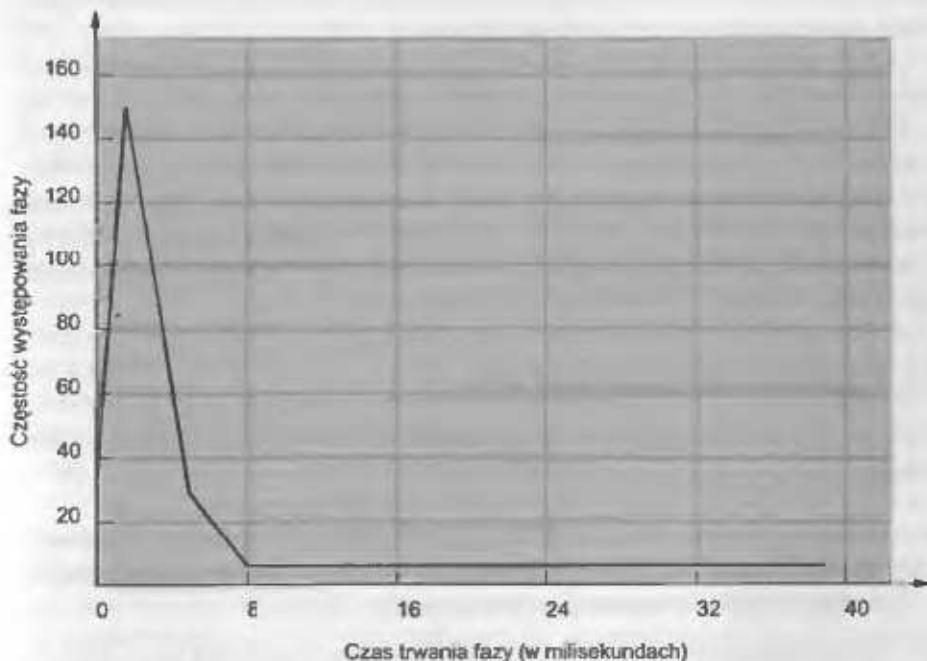
Planowanie jest podstawową funkcją systemu operacyjnego. Użytkowanie niemal wszystkich zasobów komputerowych podlega planowaniu. Procesor jest, rzecz jasna, jednym z podstawowych zasobów komputera. Dlatego planowanie jego przydziału ma zasadnicze znaczenie w projektowaniu systemu operacyjnego.

5.1.1 Cykl faz procesora i wejścia-wyjścia

Sukces w planowaniu przydziału procesora zależy od pewnej, dającej się zaobserwować właściwości procesów. Otóż wykonanie procesu składa się z następujących po sobie cykli działania procesora i oczekiwania na urządzenia zewnętrzne. Procesy naprzemiennie przechodzą od jednego do drugiego z tych dwóch stanów. Wykonanie procesu zaczyna się od *fazy procesora* (ang. *CPU burst*). Po niej następuje *faza wejścia-wyjścia* (ang. *I/O burst*), po której proces znów przechodzi do fazy procesora, po czym ponownie – do fazy wej-



Rys. 5.1 Naprzemianowy ciąg faz procesora i wejścia-wyjścia



Rys. 5.2 Histogram czasów faz procesora

ścia-wyjścia itd. W koncu, w ostatniej fazie procesora, proces – zamiast przejść do następnej fazy wejścia-wyjścia – wysyła do systemu żądanie zakończenia swojego działania (rys. 5.1).

Dokonano skrupulatnych pomiarów okresów zatrudnień procesora. Choć różnią się one mocno w zależności od procesu i rodzaju komputera, krzywa częstości ich występowania ma kształt zbliżony do wykresu przedstawionego na rys. 5.2. Można ją ogólnie określić jako wykładniczą lub hiperwykładniczą. Okazuje się, że istnieje wiele krótkich faz procesora i mało długich jego faz. Proces ograniczony przez wejście-wyjście ma zazwyczaj wiele krótkich faz procesora. Proces ograniczony przez procesor może mieć mało, lecz bardzo długich faz procesora. Ten rozkład może mieć duże znaczenie przy dobieraniu właściwego algorytmu planowania przydziału procesora.

5.1.2 Planista przydziału procesora

Gdy tylko procesor zaczyna być beczczynny, system operacyjny musi wybrać do wykonywania jakiś proces z kolejki procesów gotowych. Wyboru dokonuje planista krótkoterminowy, czyli planista przydziału procesora. Planista

ten wybiera jeden proces spośród przebywających w pamięci procesów gotowych do wykonania i przydziela mu procesor.

Zauważmy, że kolejka procesów gotowych niekoniecznie musi być kolejką typu „pierwszy na wejściu – pierwszy na wyjściu” (ang. *first-in, first-out* – FIFO). Podczas omawiania różnych algorytmów planowania przekonamy się, że kolejka procesów gotowych może być zrealizowana jako kolejka FIFO, kolejka priorytetowa, drzewo lub nawet nieuporządkowana lista. Jednakże teoretycznie w kolejce procesów gotowych wszystkie procesy oczekują w szeregu na szansę przydzielienia im procesora. Elementami kolejek są na ogół bloki kontrolne procesów.

5.1.3 Planowanie wywłaszczające*

Decyzje o przydiale procesora mogą zapadać w następujących czterech sytuacjach:

1. Proces przeszedł od stanu aktywności do stanu czekania (np. z powodu zamówienia na wejście-wyjście lub rozpoczęcia czekania na zakończenie działania któregoś z procesów potomnych).
2. Proces przeszedł od stanu aktywności do stanu gotowości (np. wskutek wystąpienia przerwania).
3. Proces przeszedł od stanu czekania do stanu gotowości (np. po zakończeniu operacji wejścia-wyjścia).
4. Proces kończy działanie.

W sytuacjach 1 i 4 nie ma możliwości wyboru. Kandydatem do przydiale procesora musi być nowy proces (jeśli taki istnieje w kolejce procesów gotowych). Natomiast w sytuacjach 2 i 3 można dokonać wyboru.

Jeśli planowanie odbywa się tylko w warunkach 1 i 4, to mówimy o *niewywłaszczeniowym* (ang. *nonpreemptive*) schemacie planowania; w przeciwnym razie algorytm planowania jest *wywłaszczeniowy* (ang. *preemptive*). W planowaniu bez wywłaszczeń proces, który otrzyma procesor, zachowuje go dopóty, dopóki nie odda go z powodu swojego zakończenia lub przejścia do stanu czekania. Ta metoda planowania jest używana w środowisku Microsoft Windows. Jest ona jedyną metodą nadającą się do zastosowania w przypadku niektórych rodzajów sprzętu, gdyż nie wymaga specjalnego oprzyrządowania (np. czasomierza), które jest niezbędne do planowania wywłaszczającego.

* W innych książkach spotyka się określenie *szeregowanie z wywłaszczeniem*.
– Przyp. red.

Planowanie wywłaszczające jest, niestety, kosztowne. Rozważmy przypadek dwu procesów korzystających ze wspólnych danych. Jeden z nich może zostać wywłaszczony podczas aktualizowania przez niego danych, po czym nastąpi uaktywnienie drugiego procesu. Drugi proces może usiłować czytać dane, których stan jest obecnie niespójny. Koordynowanie dostępu do danych dzielonych wymaga nowych mechanizmów – zagadnienie to omawiamy w rozdz. 6.

Wywłaszczańe wpływa także na konstrukcję jądra systemu operacyjnego. Podczas wykonywania funkcji systemowej jądro może być zajęte działaniami na rzecz procesu. Czynności te mogą powodować konieczność zmiany ważnych danych jądra (np. kolejek wejścia-wyjścia). Co się stanie, jeśli proces zostanie wywłaszczony w trakcie tych zmian i jądro (albo moduł sterujący urządzeniem) ma przeczytać lub zmienić tę samą strukturę? Dojdzie do chaosu. Niektóre systemy operacyjne, w tym większość wersji systemu UNIX, radzą sobie z tym kłopotem, oczekując z przełączaniem kontekstu do zakończenia wywołania systemowego lub do zablokowania procesu na wejściu-wyjściu. Postępowanie takie zapewnia prostotę struktury jądra, gdyż nie będzie ono wywłaszczało procesu w chwilach, w których struktury danych jądra są niespójne. Niestety, taki model działania jądra nie nadaje się do obliczeń w czasie rzeczywistym ani do wieloprzetwarzania. Zagadnienia te oraz ich rozwiązania są omówione w p. 5.4. i 5.5.

W przypadku systemu UNIX wciąż istnieją ryzykowne fragmenty kodu. Ponieważ na mocy definicji przerwania mogą się pojawiać w dowolnych chwilach i nie zawsze mogą być pominięte przez jądro, więc fragmenty kodu dotyczące przerwań muszą być zabezpieczone przed jednoczesnym użyciem. System operacyjny musi przyjmować przerwania niemal w każdej chwili, w przeciwnym razie dane wejściowe mogą zginąć, a dane wyjściowe mogą zostać zniszczone przez ponowne zapisanie. Aby uniemożliwić wspólnie dostępny dostęp kilku procesów do takich sekcji kodu, stosuje się wyłączenie przerwań na wejściu do nich i ponownie włączanie przerwań przy wychodzeniu z tych sekcji.

5.1.4 Program ekspediujący

Odrębnym elementem zaangażowanym w planowanie przydziału procesora jest *ekspedytor* (ang. *dispatcher*). Ekspedytor jest modelem, który faktycznie przekazuje procesor do dyspozycji procesu wybranego przez planistę krótkoterminowego. Do obowiązków ekspedytora należą:

- przełączanie kontekstu;
- przełączanie do trybu użytkownika;

- wykonanie skoku do odpowiedniej komórki w programie użytkownika w celu wznowienia działania programu.

Ekspedytor powinien być możliwie jak najszybszy, gdyż jest wywoływany podczas każdego przełączania procesu. Czas, który ekspedytor zużywa na wstrzymanie jednego procesu i uaktywnienie innego, nazywa się *opóźnieniem ekspedycji* (ang. *dispatch latency*).

5.2 ■ Kryteria planowania

Różne algorytmy planowania przydziału procesora mają rozmaite właściwości i do pewnych klas procesów mogą się nadawać lepiej niż do innych. Wybierając algorytm w konkretnej sytuacji, należy rozważyć cechy różnych algorytmów.

Do porównywania algorytmów planowania przydziału procesora zaproponowano wiele kryteriów. Wybór metody użytej do porównywania może powodować istotne różnice w określeniu najlepszego algorytmu. W stosowanych kryteriach uwzględnia się między innymi wymienione poniżej właściwości:

- **Wykorzystanie procesora:** Dąży się do tego, aby procesor był nieustannie zajęty pracą. Wykorzystanie procesora może się wahać w granicach od 0 do 100%. W rzeczywistym systemie powinno się ono mieścić w przedziale od 40% (słabe obciążenie systemu) do 90% (intensywna eksploatacja systemu).
- **Przepustowość:** Jeśli procesor jest zajęty wykonywaniem procesów, to praca postępuje naprzód. Jedną z miar pracy jest liczba procesów kończonych w jednostce czasu, zwana *przepustowością* (ang. *throughput*). Dla długich procesów wartość ta może wynosić jeden proces na godzinę, dla krótkich transakcji przepustowość może się kształtować na poziomie 10 procesów na sekundę.
- **Czas cyklu przetwarzania:** Ważnym kryterium dla konkretnego procesu jest czas potrzebny na jego wykonanie. Czas upływający między chwilą nadania procesu do systemu a chwilą zakończenia procesu nazywa się *czasem cyklu przetwarzania* (ang. *turnaround time*). Jest to suma określonych na czasie czekaniu na wejście do pamięci, czekaniu w kolejce procesów gotowych do wykonania, wykonywaniu procesu przez procesor i wykonywaniu operacji wejścia-wyjścia.

* W ujęciu jest także określenie *czas obiegu zadania* (w systemie). – Przyp. tłum.

- **Czas oczekiwania:** Algorytm planowania przydziału procesora nie ma faktycznie wpływu na czas, w którym proces działa lub wykonuje operacje wejścia-wyjścia; dotyczy on tylko czasu, który proces spędza w kolejce procesów gotowych do wykonania. Czas oczekiwania jest sumą okresów, w których proces czeka w kolejce procesów gotowych do działania.
- **Czas odpowiedzi:** W systemach interakcyjnych czas cyklu przetwarzania może nie być najlepszym kryterium. Często bywa tak, że proces produkuje pewne wyniki dość wcześnie i wykonuje następne obliczenia, podczas gdy poprzednie rezultaty są prezentowane użytkownikowi. Toteż kolejną miarą jest czas upływający między wysłaniem żądania (przedłożeniem zamówienia) a pojawieniem się pierwszej odpowiedzi. Ta miara, nosząca nazwę *czasu odpowiedzi* (ang. *response time*), określa, ile czasu upływa do rozpoczęcia odpowiedzi, ale nie obejmuje czasu potrzebnego na wyprowadzenie tej odpowiedzi. Czas odpowiedzi jest na ogół uzależniony od szybkości działania urządzenia wyjściowego.

Dąży się do tego, aby wykorzystanie procesora i przepustowość były maksymalne, natomiast czas cyklu przetwarzania, oczekiwania i odpowiedzi – minimalne. W większości przypadków optymalizuje się miarę średnią. Jednakże w pewnych sytuacjach optymalizacja wartości minimalnych lub maksymalnych może być bardziej pożądana niż troska o wynik średniej. Jeśli na przykład dbamy o dobrą obsługę wszystkich użytkowników, to może nam najbardziej zależeć na zmniejszeniu maksymalnego czasu odpowiedzi.

Istnieje pogląd, że w systemach interakcyjnych (np. z podziałem czasu) ważniejsze jest minimalizowanie *wariancji* czasu odpowiedzi aniżeli minimalizowanie średniego czasu odpowiedzi. System z sensownym i przewidywalnym czasem odpowiedzi może być bardziej pożądany niż system, który ma przeciętnie szybszy, ale zmienny czas reakcji. Jednak duchy czas niewiele powstało algorytmów planowania przydziału procesora minimalizujących wariancję.

Omawiając różne algorytmy planowania przydziału procesora, będziemy chcieli ilustrować ich działanie. W dokładnej ilustracji należałoby uwzględnić wiele procesów, z których każdy powinien się składać z setek zatrudnień procesora i urządzeń zewnętrznych. W celu uproszczenia ilustracji będziemy rozważać w naszych przykładach tylko jedną fazę procesora (w milisekundach) przypadającą na każdy proces. Miarą stosowaną w porównaniach będzie średni czas oczekiwania. Bardziej złożone mechanizmy oszacowań zostaną omówione w p. 5.6.

5.3 ■ Algorytmy planowania

Planowanie przydziału procesora jest związane z zagadnieniem podejmowania decyzji o tym, którym procesem z kolejki procesów gotowych do działania ma być przydzielona jednostka centralna. Istnieje wiele różnych algorytmów planowania przydziału procesora, niektóre z nich przedstawiamy poniżej.

5.3.1 Planowanie metodą FCFS – „pierwszy zgłoszony – pierwszy obsłużony”

Najprostszym ze znanych algorytmów planowania przydziału procesora jest algorytm „pierwszy zgłoszony – pierwszy obsłużony” (ang. *first-come, first-served* – FCFS). Według tego schematu proces, który pierwszy zamówi procesor, pierwszy go otrzyma. Implementację tego algorytmu łatwo się uzyskuje za pomocą kolejki FIFO. Blok kontrolny procesu wchodzącego do kolejki jest dodawany na końcu kolejki. Wolny procesor przydziela się procesowi z czoła kolejki. Algorytm planowania metodą FCFS można łatwo zrozumieć i zaprogramować.

Jeśli jednak przyjmie się metodę FCFS, to średni czas oczekiwania bywa bardzo długi. Rozważmy następujący zbiór procesów nachodzących w chwili 0, których wyrażone w milisekundach długości faz procesora wynoszą:

<u>Proces</u>	<u>Czas trwania fazy</u>
P_1	24
P_2	3
P_3	3

Jeśli procesy nadjejdą w porządku P_1, P_2, P_3 i zostaną obsłużone w porządku FCFS, to otrzymamy wynik pokazany na poniższym diagramie Gantta:



Czas oczekiwania wynosi 0 ms dla procesu P_1 , 24 ms dla procesu P_2 i 27 ms dla procesu P_3 . Zatem średni czas oczekiwania wynosi $(0 + 24 + 27)/3 = 17$ ms. Gdyby procesy nadeszły w kolejności P_2, P_3, P_1 , rezultat przedstawiłby się jak na poniższym diagramie Ganta:



Średni czas czekania wyniósłby teraz $(6 + 0 + 3)/3 = 3$ ms. Nastąpiłoby za-uważalne jego skrócenie. Tak więc średni czas oczekiwania przy zastosowaniu metody FCFS nie jest na ogół minimalny i może wykazywać znaczne wahania, jeśli proces ma rozmaitej długości fazy procesora.

Rozważmy jeszcze działanie algorytmu FCFS w warunkach dynamicznych. Założymy, że mamy jeden proces ograniczony przez procesor i wiele procesów ograniczonych przez wejście-wyjście. Przepływ procesów w systemie może prowadzić do następującego scenariusza. Proces ograniczony przez procesor uzyska przydział procesora i będzie z niego korzystać. W tym czasie wszystkie inne procesy kończą swoje operacje wejścia-wyjścia i przemieszczą się do kolejki procesów gotowych, oczekujących na przydział procesora. Podczas oczekiwania tych procesów w kolejce procesów gotowych urządzenia wejścia-wyjścia są bezczynne. Po jakimś czasie proces ograniczony przez procesor skończy swoją fazę procesora i przejdzie do kolejki oczekującej na wejście-wyjście. Wszystkie procesy ograniczone przez wejście-wyjście, mając bardzo krótkie fazy procesora, zadziałają szybko i powrócą do kolejek związanych z urządzeniami wejścia-wyjścia. W tym momencie procesor jest bezczynny. Proces ograniczony przez procesor powróci już wtedy do kolejki procesów gotowych i otrzyma przydział procesora. I znowu wszystkie pozostałe procesy po zakończeniu operacji wejścia-wyjścia przejdą do kolejki procesów gotowych i będą tam oczekwać, aż proces ograniczony przez procesor wykona swoje obliczenia. Mamy tu do czynienia z tzw. *efektem konwoju* (ang. *convoy effect*), polegającym na tym, że wszystkie pozostałe procesy czekają na zwolnienie procesora przez jeden wielki proces. Efekt ten powoduje mniejsze wykorzystanie procesora i urządzeń, niż byłoby to możliwe, gdyby najpierw pozwolono pracować krótszym procesom.

Algorytm FCFS jest niewywłaszczający. Po objęciu kontroli nad procesorem proces utrzymuje ją do czasu, aż sam zwolni procesor wskutek zakończenia swego działania lub zamówienia operacji wejścia-wyjścia. Algorytm FCFS jest szczególnie kłopotliwy w systemach z podziałem czasu, w których jest ważne, aby każdy użytkownik dostawał swój przydział procesora w regularnych odstępach. Pozwolenie jakiemuś procesowi na zajmowanie procesora przez dłuższy czas byłoby w tych warunkach katastrofalne.

5.3.2 Planowanie metodą „najpierw najkrótsze zadanie”

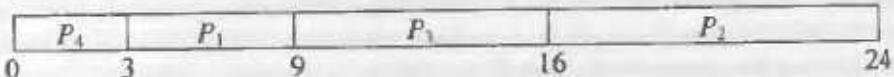
Inne podejście do planowania przydziału procesora umożliwia algorytm „najpierw najkrótsze zadanie” (ang. *shortest-job-first* – SJF). Algorytm ten wiąże z każdym procesem długość jego najbliższej z przyszłych faz procesora. Gdy procesor staje się dostępny, wówczas zostaje przydzielony procesowi mają-

czemu najkrótszą następną fazę procesora. Jeśli dwa procesy mają następujące fazy procesora równej długości, to kłopotu pozbywamy się, stosując algorytm FCFS. Zauważmy, że odpowiedniejszym określeniem byłoby tu: „najkrótsza następna faza procesora”, gdyż planowanie polega na sprawdzaniu w procesie długości jego następnej fazy procesora, a nie całej długości procesu. Nazwy SJF używa się jednak dla tego, iż przez większość osób i w większości podręczników ten sposób planowania określany jest właśnie mianem SJF.

Jako przykład rozważmy następujący zbiór procesów, których długości faz procesora w milisekundach wynoszą odpowiednio:

Proces	Czas trwania fazy
P_1	6
P_2	8
P_3	7
P_4	3

Używając algorytmu SJF, możemy zaplanować te procesy zgodnie z poniższym diagramem Gantta:



Czas oczekiwania dla procesu P_1 wynosi 3 ms, 16 ms dla procesu P_2 , 9 ms dla procesu P_3 i 0 ms dla procesu P_4 . Zatem średni czas oczekiwania wynosi $(3 + 16 + 9 + 0)/4 = 7$ ms. Gdybyśmy uzyli planowania metodą FCFS (pierwszy zgłoszony – pierwszy obsłużony), wtedy średni czas oczekiwania wyniósłby 10,25 ms.

Można udowodnić, że algorytm SJF jest *optimalny*, tzn. daje minimalny średni czas oczekiwania dla danego zbioru procesów. Umieszczenie krótkiego procesu przed długim w większym stopniu zmniejsza czas oczekiwania krótkiego procesu, aniżeli wydłuża czas oczekiwania długiego procesu. W rezultacie zmniejsza się średni czas oczekiwania.

Poważną trudność w algorytmie SJF sprawia określenie długości następującego zamówienia na przydział procesora. W planowaniu długoterminowym (planowaniu zadań) w systemie wsadowym za ową długość możemy przyjąć limit czasu procesu, określany przez użytkownika przedkładającego zadanie. Użytkownicy mogą być zainteresowani w dokładnym oszacowywaniu limitu czasu procesu, ponieważ niższa jego wartość może oznaczać szybszą odpowiedź. (Zbyt niska wartość spowoduje błąd przekroczenia limitu czasu, powiązający za sobą konieczność powtórnego przedłożenia zadania). Algorytm SJF jest często używany w planowaniu długoterminowym.

Algorytm SJF, choć optymalny, nie może być zrealizowany na poziomie krótkoterminowego planowania przydziału procesora. Nie ma sposobu na poznanie długości następnej fazy procesora. Próbuje się zatem przybliżać planowanie SJF. Nie jesteśmy w stanie poznać długości następnej fazy procesora, lecz możemy spróbować oszacować jej wartość. Zakładamy, że długość następnej fazy procesora będzie podobna do długości faz poprzednich. Obliczając przybliżenie długości następnej fazy procesora, można będzie wybrać proces z najkrótszą przewidywaną fazą procesora.

Następną fazę procesora określa się na ogół jako średnią wykładniczą pomiarów długości poprzednich faz procesora. Niech t_n oznacza długość n -tej fazy procesora, a τ_{n+1} symbolizuje przewidywaną przez nas długość następnej fazy procesora. Wówczas dla α z przedziału $0 \leq \alpha \leq 1$ definiujemy

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n.$$

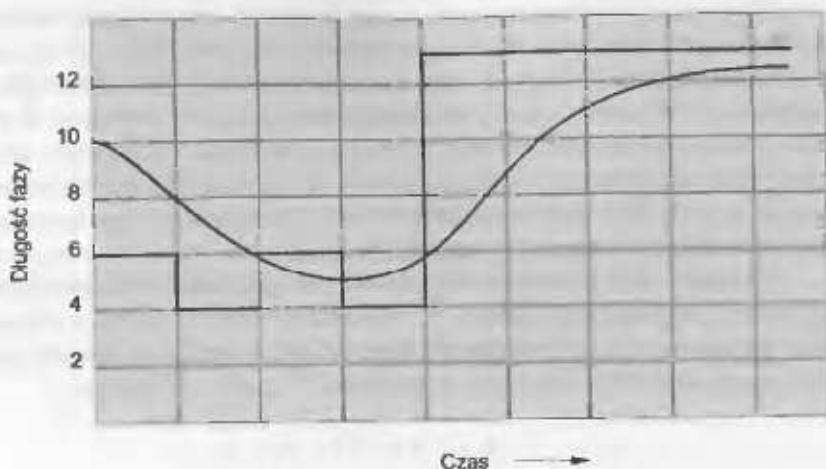
Wzór ten określa średnią wykładniczą. Wartość t_n określa najświeższą informację, natomiast w τ_n przechowuje się dane z minionej historii. Parametr α ustala w naszym oszacowaniu względną wagę między niedawną a wcześniejszą historią. Jeśli $\alpha = 0$, to $\tau_{n+1} = t_n$ i niedawna historia nie ma wpływu na wynik (zakłada się, że niedawne warunki były czymś przejściowym). Jeśli $\alpha = 1$, to $\tau_{n+1} = t_n$ i uwzględnia się tylko najnowsze notowanie długości fazy procesora (o historii zakładając, że jest przebrzmiała i nieistotna). Najczęściej $\alpha = 1/2$, czyli nowej i starej historii nadaje się jednakową wagę. Na rysunku 5.3 widać średnią wykładniczą z $\alpha = 1/2$. Początkową wartość τ_0 można zdefiniować jako stałą lub jako średnią wziętą z całego systemu.

Aby zrozumieć zachowanie się średniej wykładniczej, możemy przez podstawienie za t_n rozwinąć wzór na τ_{n+1} do postaci

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0.$$

Ponieważ zarówno α , jak i $(1 - \alpha)$ są mniejsze niż 1, każdy następny składnik ma mniejszą wagę niż jego poprzednik.

Algorytm SJF może być *wywłaszczający* lub *niewywłaszczający*. Konieczność wyboru powstaje wówczas, gdy w kolejce procesów gotowych pojawia się nowy proces, a poprzedni proces jeszcze używa procesora. Nowy proces może mieć krótszą następną fazę procesora niż to, co jeszcze pozostało do wykonania w procesie bieżącym. Wywłaszczający algorytm SJF usunie w tej sytuacji dorywczo proces z procesora, podczas gdy niewywłaszczający algorytm SJF pozwoli bieżącemu procesowi na zakończenie fazy procesora. Wywłaszczającą metodę SJF przydziału procesora nazywa się czasami planowaniem metodą „najpierw najkrótszy pozostałý czas” (ang. *shortest-remaining-time first*).



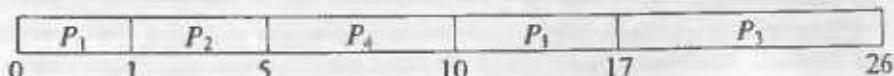
Faza procesora (t_i)	6	4	6	4	13	13	13	...	
Wartość „odgadniętej” (τ_i)	10	8	6	6	5	9	11	12	...

Rys. 5.3 Przewidywanie następnych faz procesora na podstawie średniej wykładniczej

Jako przykład rozważmy następujące cztery procesy, których długości faz procesora w milisekundach wynoszą odpowiednio:

Proces	Czas przybycia	Czas trwania fazy
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Jeśli procesy przybywają do kolejki procesów gotowych w podanych chwilach i potrzebują faz procesora o długościach jak na wykazie, to wynikowe wyłączające zaplanowanie SJF przydziału procesora będzie takie jak na poniższym diagramie Gantta:



Proces P_1 startuje w chwili 0, ponieważ jest jedynym procesem w kolejce. Proces P_2 nadchodzi w chwili 1. Czas pozostały procesowi P_1 (7 ms) jest większy od czasu wymaganego przez proces P_2 (4 ms), toteż proces P_1 jest

wywłaszczały, a proces P_2 przejmuje procesor. Średni czas oczekiwania w tym przykładzie wynosi $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5$ ms. Niewywłaszczające planowanie przydziału procesora metodą SJF dałoby w wyniku średni czas oczekiwania równy 7,75 ms.

5.3.3 Planowanie priorytetowe

Algorytm SJF (najpierw najkrótsze zadanie) jest szczególnym przypadkiem ogólnego algorytmu *planowania priorytetowego*. Każdemu procesowi przypisuje się priorytet, po czym przydziela się procesor temu procesowi, którego priorytet jest najwyższy. Procesy o różnych priorytetach planuje się w porządku FCFS.

Algorytm SJF jest po prostu algorytmem priorytetowym, w którym priorytet (p) jest odwrotnością następnej (przewidywanej) fazy procesora. Im większa jest faza procesora, tym niższy staje się priorytet – i na odwrót.

Wypada zauważyc, że przy tego rodzaju planowaniu używa się terminów: *niski* i *wysoki* priorytet. Priorytety należą zwykle do pewnego, ustalonego przedziału liczb całkowitych, na przykład od 0 do 7 lub od 0 do 4095. Jednakże nie ma ogólnych ustaleń co do tego, czy 0 jest najwyższym czy też najniższym priorytetem. W niektórych systemach używa się małych liczb do reprezentowania niskich priorytetów, w innych zaś małe liczby oznaczają wysokie priorytety. Może to prowadzić do nieporozumień. W naszej książce przyjmujemy, że małe liczby reprezentują wysokie priorytety.

Dla przykładu rozważmy zbiór procesów przybyłych w czasie 0 i w porządku P_1, P_2, \dots, P_5 oraz mających następujące milisekundowe czasy faz procesora:

Proces	Czas trwania fazy	Priorytet
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Przy zastosowaniu planowania priorytetowego otrzymujemy uporządkowanie takie jak na poniższym diagramie Gantta:

P_2	P_5		P_1		P_3	P_4
0	1	6		16	18	19

Średni czas oczekiwania wynosi 8,2 ms.

Priorytety mogą być definiowane wewnętrznie lub zewnętrznie. Do wewnętrznego zdefiniowania priorytetu używa się jakiejś mierzonej właściwości procesu (jednej lub wielu) i na jej podstawie oblicza się priorytet. Mogą to być na przykład: limity czasu, wielkość obszaru wymaganej pamięci, liczba otwartych plików, jak również stosunek średniej fazy wejścia-wyjścia do średniej fazy procesora. Priorytety zewnętrzne są określone na podstawie kryteriów zewnętrznych wobec systemu operacyjnego – takich jak ważność procesu, rodzaj i kwota opłat ponoszonych za użytkowanie komputera, instytucja sponsorująca pracę i inne czynniki, często o znaczeniu politycznym.

Planowanie priorytetowe może być wywłaszczające lub niewywłaszczające. Priorytet procesu dołączanego do kolejki procesów gotowych jest porównywany z priorytetem bieżącą wykonywanego procesu. Wywłaszczający algorytm priorytetowy spowoduje odebranie procesora bieżącemu procesowi, jeśli jego priorytet jest niższy od priorytetu nowo przybyłego procesu. Niewywłaszczający algorytm priorytetowy ustawi po prostu nowy proces na czele kolejki procesów gotowych do wykonania.

Podstawowym problemem w planowaniu priorytetowym jest *nieskończon就业 blokowanie* (ang. *indefinite blocking*), zwane też *grodzeniem* (ang. *starvation*). Proces, który jest gotowy do wykonania, lecz pozbawiony procesora, można traktować jako zablokowany z powodu oczekiwania na przydział procesora. Algorytm planowania priorytetowego może pozostawić niektóre niskopriorytetowe procesy w stanie nie kończącego się czekania na procesor. W mocno obciążonym systemie komputerowym stały napływ procesów o wyższych priorytetach może nigdy nie dopuścić niskopriorytetowego procesu do procesora. Dochodzi wówczas do jednego z dwóch zdarzeń: albo oczekujący proces zostanie w końcu wykonany (o drugiej w nocy z soboty na niedzielę, kiedy obciążenie systemu wreszcie zmaleje), albo w systemie komputerowym wystąpi jakaś awaria i wszystkie niedokończone, niskopriorytetowe procesy zostaną utracone. (Niemałym echem odbija się sprawa wykrycia w 1973 r. w MIT, w wycofywanym z eksploatacji systemie IBM 7094 niskopriorytetowego procesu przedłożonego do wykonania w 1967 r. i wciąż nie uaktywnionego).

Rozwiązaniem problemu nieskończonego blokowania procesów niskopriorytetowych jest ich *postarzanie* (ang. *aging*). Polega ono na stopniowym podwyższaniu priorytetów procesów długo oczekujących w systemie. Jeśli na przykład przedział priorytetów wynosi od 0 (niski) do 127 (wysoki), to można podwyszyszać priorytet procesu o 1 co każde 15 minut. Przy takim postępowaniu nawet proces o początkowym priorytecie 0 uzyska w końcu najwyższy priorytet w systemie i zostanie wykonany. W istocie, postarzanie procesu o priorytecie 0 do priorytetu 127 nie potrwa dłużej niż 32 godziny.

5.3.4 Planowanie rotacyjne

Algorytm *planowania rotacyjnego* (ang. *round-robin* – RR)* zaprojektowano specjalnie dla systemów z podziałem czasu. Jest on podobny do algorytmu FCFS, z tym że w celu przełączania procesów dodano do niego wywłaszczenie. Ustala się małą jednostkę czasu, nazywaną *kwantem czasu* (ang. *time quantum*) lub odcinkiem czasu. Kwant czasu wynosi zwykle od 10 do 100 milisekund. Kolejka procesów gotowych do wykonania jest traktowana jak kolejka cykliczna. Planista przydziału procesora przegląda tę kolejkę i każdemu procesowi przydziela odcinek czasu nie dłuższy od jednego kwantu czasu.

W celu implementacji planowania rotacyjnego utrzymuje się kolejkę procesów gotowych zgodnie z regułą kolejki FIFO, czyli „pierwszy na wejściu – pierwszy na wyjściu”. Nowe procesy są dodawane na końcu kolejki procesów gotowych. Planista przydziału procesora bierze pierwszy proces z kolejki procesów gotowych, ustawia czasomierz na przerwanie po upływie 1 kwantu czasu, po czym ekspediuje ten proces do procesora.

Wypadki mogą się potoczyć dwójako. Proces może mieć fazę procesora krótszą niż 1 kwant czasu. Wówczas proces z własnej inicjatywy zwolni procesor. Planista postąpi wtedy tak jak poprzednio z następnym procesorem z kolejki procesów gotowych. W przeciwnym razie, jeśli wykonywany bieżąco proces ma fazę procesora dłuższą niż 1 kwant czasu, to nastąpi przerwanie zegarowego systemu operacyjnego. Dokona się przełączenie kontekstu i proces zostanie odłożony na koniec kolejki procesów gotowych, planista zaś wybierze następny proces z tej kolejki.

Niestety, średni czas oczekiwania w metodzie rotacyjnej bywa dość długi. Rozważmy zbiór procesów, które nadchodzą w chwili 0 i mają następujące milisekundowe czasy faz procesora:

Proces	Czas trwania fazy
P_1	24
P_2	3
P_3	3

Jeśli zastosujemy kwant czasu równy 4 ms, to proces P_1 otrzyma pierwsze 4 ms. Ponieważ potrzebuje on jeszcze 20 ms, zostanie wywłaszczone po upływie pierwszego kwantu czasu, a procesor będzie przydzielony następnemu procesowi w kolejce – P_2 . Ponieważ faza procesu P_2 trwa krócej niż 4 ms,

* Ze względu na terminologię międzynarodową warto zapamiętać ten skrót na oznaczenie algorytmu rotacyjnego. – Przyp. tłum.

proces zakończy pracę przed upływem kwantu czasu. Wówczas następny proces, czyli P_1 , otrzyma przydział procesora. Kiedy każdy proces otrzyma po jednym kwancie czasu, wtedy procesor zostanie z powrotem przydzielony procesowi P_1 , aby mógł on wykorzystać dodatkowe kwanty czasu. Wynikowe zaplanowanie przydziału procesora metodą rotacyjną jest następujące:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	P_1
0	4	7	10	14	18	22	26	30

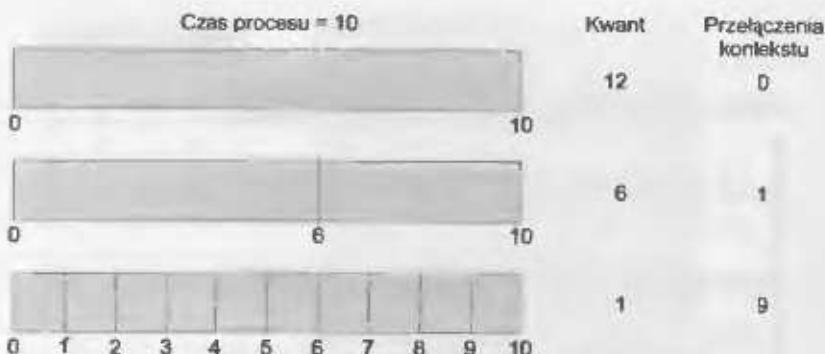
Średni czas oczekiwania wynosi $17/3 = 5,66$ ms.

W algorytmie planowania rotacyjnego żaden proces nie otrzyma więcej niż 1 kwant czasu procesora za jednym razem. Jeżeli faza procesora w danym procesie przekracza 1 kwant czasu, to proces będzie *wywłaszczyony* i wycofany do kolejki procesów gotowych. Algorytm planowania rotacyjnego jest *wywłaszczający*.

Jeśli kolejka procesów gotowych do wykonania składa się z n procesów, a kwant czasu wynosi q , to każdy proces dostaje $1/n$ czasu procesora porcjami, których wielkość nie przekracza q jednostek czasu. Na następny kwant czasu każdy proces musi czekać nie dłużej niż $(n - 1) \times q$ jednostek czasu. Na przykład przy pięciu procesach i kwancie czasu wynoszącym 20 ms każdy proces dostaje po 20 ms co każde 100 ms.

Wydajność algorytmu rotacyjnego w dużym stopniu zależy od rozmiaru kwantu czasu. W jednym z dwu skrajnych przypadków, gdy kwant czasu jest bardzo długi (nieskończony), metoda rotacyjna (RR) sprowadza się do polityki FCFS. Jeśli kwant czasu jest bardzo mały (powiedzmy – 1 μ s), to planowanie rotacyjne nazywa się *dzieleniem procesora* (ang. *processor sharing*), sprawia ono (teoretycznie) na użytkownikach wrażenie, że każdy z n procesów ma własny procesor działający z $1/n$ szybkości rzeczywistego procesora. Pomysł taki wykorzystano w firmie Control Data Corporation (CDC) do budowy w ramach jednego komputera 10 peryferyjnych procesorów i 10 kompletów sprzętowych rejestrów. Sprzęt ten wykonuje jeden rozkaz na jednym zbiorze rejestrów, po czym przechodzi do następnego zbioru rejestrów. Cykl ów powtarza się, dając w efekcie 10 wolniej działających procesorów zamiast jednego szybkiego. (W rzeczywistości, ponieważ procesor był znacznie szybszy od pamięci, a każdy rozkaz odwoływał się do pamięci, procesory peryferyjne nie były dużo wolniejsze od pojedynczego procesora).

W oprogramowaniu należy jednak wziąć również pod uwagę wpływ przełączania kontekstu na zachowanie algorytmu rotacyjnego. Założymy, że mamy tylko jeden proces, którego faza procesora ma długość 10 jednostek czasu. Jeśli kwant czasu wynosi 12 jednostek, to dany proces skończy się w czasie krótszym niż 1 kwant, nie wymagając dodatkowej obsługi. Jeśli



Rys. 5.4 Mniejszy kwant czasu zwiększa przełączanie kontekstu

kwant wyniesie 6 jednostek czasu, to proces będzie już potrzebował dwóch kwantów, a to spowoduje przełączanie kontekstu. Gdyby kwant czasu wynosił 1 jednostkę czasu, nastąpiłoby wówczas aż dziewięć przełączzeń kontekstu, co musiałoby spowolnić wykonanie procesu (rys. 5.4).

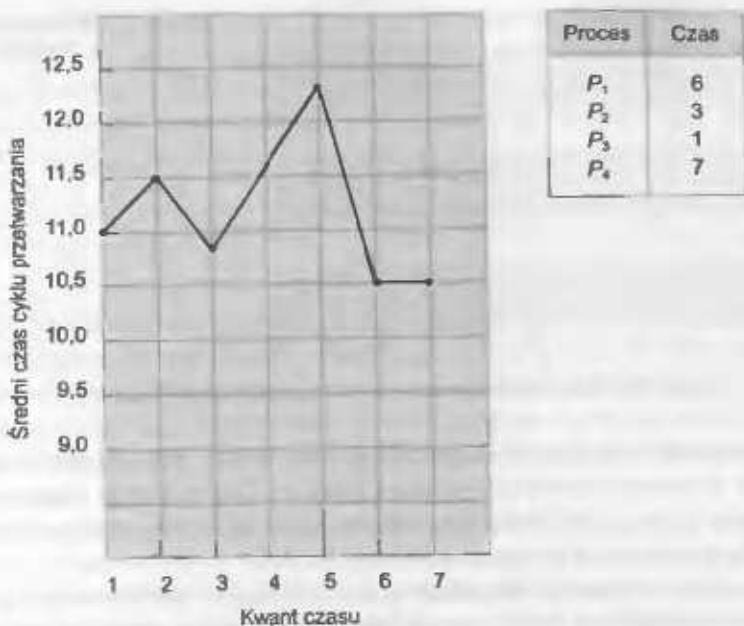
Jest zatem wskazane, aby kwant czasu był długi w porównaniu z czasem przełączania kontekstu. Jeśli czas przełączania kontekstu wynosi w przybliżeniu 10% kwantu czasu, to około 10% czasu procesora traci się na przełączanie kontekstu.

Od rozmiaru kwantu czasu zależy również czas cyklu przetwarzania. Jak można zobaczyć na rys. 5.5, średni czas cyklu przetwarzania zbioru procesów niekoniecznie poprawia się ze wzrostem kwantu czasu. Średni czas cyklu przetwarzania poprawia się na ogół wtedy, kiedy większość procesów kończy swoje kolejne fazy procesora w pojedynczych kwantach czasu. Na przykład przy trzech procesach, z których każdy ma długość 10 jednostek czasu, i kwancie o długości 1 jednostki czasu średni czas obiegu wyniesie 29. Natomiast przy kwancie czasu o długości 10 średni czas cyklu przetwarzania skróci się do 20. Po uwzględnieniu czasu przełączania kontekstu średni czas cyklu przetwarzania będzie wzrastać przy małych kwantach czasu, ponieważ będą potrzebne częstsze przełączania kontekstu.

Jeśli jednakże kwant czasu jest za duży, to planowanie rotacyjne degeneruje się do schematu planowania FCFS. Jako wypróbowaną regułę można przyjąć, że 80% faz procesora powinno być krótszych niż jeden kwant czasu.

5.3.5 Wielopoziomowe planowanie kolejek

Osobną klasę algorytmów planowania wytworzono na okoliczność sytuacji, w których jest możliwe łatwe zaliczenie procesów do kilku różnych grup. Na przykład powszechnie rozgranicza się procesy *pierwszoplanowe* (ang. *fore-*



Rys. 5.5 Średni czas cyku przetwarzania zależy od kwantu czasu

ground), czyli interakcyjne, oraz procesy *drugoplanowe* (ang. *background*), inaczej – wsadowe. Te dwa rodzaje procesów, różniąc się wymaganiami na czasy odpowiedzi, mogą mieć różne wymagania odnośnie do planowania. Ponadto procesy pierwszoplanowe mogą mieć pierwszeństwo (zewnętrznie zdefiniowane) przed procesami drugoplanowymi.

Algorytm wielopoziomowego planowania kolejek rozdziela kolejkę procesów gotowych na osobne kolejki (rys. 5.6). W zależności od pewnych cech, jak rozmiar pamięci, priorytet lub typ procesu, procesy zostają na stałe przydzielone do jednej z tych kolejek. Każda kolejka ma własny algorytm planujący. Na przykład procesy pierwszoplanowe i drugoplanowe mogą być ulokowane w osobnych kolejkach. Do kolejki procesów pierwszoplanowych można zastosować algorytm planowania rotacyjnego, a do kolejki procesów drugoplanowych można zastosować algorytm planowania przydziału procesora metodą FCFS.

Ponadto musi istnieć jakieś planowanie między kolejkami, na ogół realizowane za pomocą stałopriorytetowego planowania z wywłaszczeniami. Kolejka pierwszoplanowa może na przykład mieć bezwzględny priorytet nad kolejką procesów wsadowych.

Przypatrzymy się przykładowi wielopoziomowego algorytmu planowania pięciu kolejek.

Najwyższy priorytet.



Najniższy priorytet.

Rys. 5.6 Wielopoziomowe planowanie kolejek

1. Procesy systemowe
2. Procesy interakcyjne
3. Procesy redagowania interakcyjnego
4. Procesy wsadowe
5. Procesy studenckie

Każda kolejka ma bezwzględne pierwszeństwo przed kolejkami o niższych priorytetach. Na przykład żaden proces z kolejki procesów wsadowych nie może pracować dopóty, dopóki kolejki procesów systemowych, interakcyjnych i interakcyjnego redagowania nie są puste. Gdyby jakiś proces interakcyjnego redagowania nadszedł do kolejki procesów gotowych w chwili, w której procesor wykonywałby proces wsadowy, wtedy proces wsadowy zostałby wywłączony.

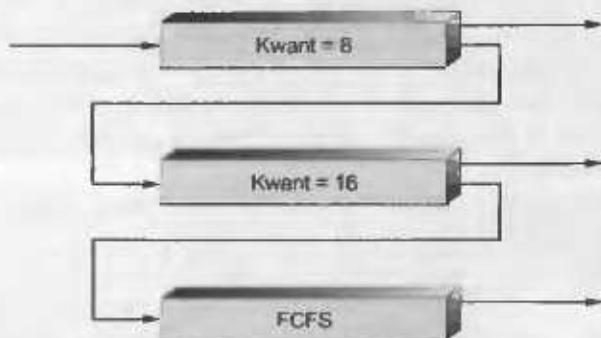
Inną możliwość daje operowanie przedziałami czasu między kolejkami. Każda kolejka dostaje pewną porcję czasu procesora, aby go rozplanować między znajdujące się w niej procesy. Na przykład kolejka procesów pierwszoplanowych może otrzymać 80% czasu procesora na planowanie jej procesów metodą rotacyjną, a kolejka drugoplanowa – pozostałe 20% czasu procesora do rozdysponowania między jej procesy sposobem FCFS.

5.3.6 Planowanie wielopoziomowych kolejek ze sprzężeniem zwrotnym

Zazwyczaj w algorytmie wielopoziomowego planowania kolejek proces jest na stałe przypisywany do kolejki w chwili jego wejścia do systemu. Procesy nie mogą przemieszczać się między kolejkami. Jeśli na przykład istnieją osobne kolejki dla pierwszoplanowych i dla drugoplanowych procesów, to żaden proces nie może zmienić swego przyorządzenia do określonego rodzaju kolejki i przejść z jednej kolejki do drugiej. Zaletą takich ustaleń jest niski koszt planowania, wadą zaś – brak elastyczności.

Planowanie wielopoziomowych kolejek ze sprzężeniem zwrotnym umożliwia w podobnych sytuacjach przemieszczanie procesów między kolejkami. Pomyśl polega na rozdzieleniu procesów o różnych rodzajach faz procesora. Jeśli proces zużywa za dużo czasu procesora, to zostanie przeniesiony do kolejki o niższym priorytecie. Postępowanie to prowadzi do pozostawienia procesów ograniczonych przez wejście-wyjście i procesów interakcyjnych w kolejach o wyższych priorytetach. Podobnie, proces oczekujący zbyt długo w niskopriorytetowej kolejce może zostać przeniesiony do kolejki o wyższym priorytecie. Taki sposób postarzania procesów zapobiega ich głodzeniu.

Rozważmy na przykład działanie planisty wielopoziomowej kolejki ze sprzężeniem zwrotnym, złożonej z trzech kolejek ponumerowanych od 0 do 2 (rys. 5.7). Planista powoduje najpierw wykonanie wszystkich procesów z kolejki 0. Po opróżnieniu kolejki 0 są wykonywane procesy z kolejki 1. Analogicznie, procesy z kolejki 2 nie będą wykonywane dopóty, dopóki kolejki 0 i 1 nie zostaną opróżnione. Proces, który nadjdzie do kolejki 1, wywłaszczy proces z kolejki 2. Z kolei proces należący do kolejki 1 zostanie wywłaszczyony przez proces nadchodzący do kolejki 0.



Rys. 5.7 Kolejki wielopoziomowe ze sprzężeniem zwrotnym

Proces wchodzący do kolejki procesów gotowych trafia do kolejki 0. W kolejce 0 proces dostaje kwant czasu wielkości 8 ms. Jeśli nie zostanie w tym czasie zakończony, to będzie przeniesiony na koniec kolejki 1. Gdy kolejka 0 się opróżni, wówczas proces z czola kolejki 1 dostanie kwant czasu równy 16 ms. Jeśli nie zdąży ukończyć swojej pracy w tym czasie, to zostanie wywłączony i trafi do kolejki 2. Procesy w kolejce 2 są wykonywane w porządku określonym metodą FCFS i tylko wtedy, gdy kolejki 0 i 1 są puste.

Taki algorytm planowania daje najwyższy priorytet procesom, których fazy procesora nie przekraczają 8 ms. Nie czekają one długo na przydział procesora, szybko kończą fazy procesora i powracały do faz wejścia-wyjścia. Procesy potrzebujące więcej niż 8 ms, lecz mniej niż 24 ms są także szybko obsługiwane, choć z niższym priorytetem niż procesy krótsze. Długie procesy automatycznie wpadają do kolejki 2 i są obsługiwane w porządku FCFS w cyklach pracy procesora nie wykorzystanych przez procesy z kolejek 0 i 1.

Mówiąc ogólnie, planista wielopoziomowych kolejek ze sprzężeniem zwotnym jest określony za pomocą następujących parametrów:

- liczby kolejek;
- algorytmu planowania dla każdej kolejki;
- metody użytej do decydowania o awansowaniu procesu do kolejki o wyższym priorytecie;
- metody użytej do decydowania o zdymisjonowaniu procesu do kolejki o niższym priorytecie;
- metody wyznaczającej kolejkę, do której trafia proces potrzebujący obsługi.

Definicja planisty wielopoziomowych kolejek ze sprzężeniem zwotnym jest najogólniejszym algorytmem planowania przydziału procesora. Można ten algorytm modyfikować, dopasowując go do projektu specyficznego systemu. Niestety, algorytm ten wymaga też określenia sposobu wybierania wartości wszystkich parametrów definiujących najlepszego planistę. Wielopoziomowa kolejka ze sprzężeniem zwotnym jest schematem najogólniejszym, lecz także najbardziej złożonym.

5.4 ■ Planowanie wieloprocesorowe

Dotychczas zajmowaliśmy się zagadnieniami planowania przydziału procesora w systemie jednoprocesorowym. Gdy jest dostępnych wiele procesorów, problem planowania ich pracy odpowiednio się komplikuje. Wypróbowano

wiele możliwości i – jak można to było zaobserwować przy planowaniu przydziału jednego procesora – żadna z nich nie okazała się najlepsza. Omówimy teraz pokróćte niektóre z aspektów planowania wieloprocesorowego. (Pełne ujęcie tego zagadnienia wykracza poza ramy naszej książki). Skoncentrujemy się na systemach, których procesory są identyczne (*homogeniczne*) pod względem wykonywanych przez nie funkcji. W systemach takich do wykonania dowolnego procesu z kolejki można użyć dowolnego dostępnego procesora. W rozdziałach 15–18 przedstawimy systemy, w których procesory są różne (systemy *heterogeniczne*); w takich systemach na danym procesorze można wykonać tylko te programy, które zostały przetłumaczone na odpowiadający mu zbiór rozkazów.

Nawet w wieloprocesorze homogenicznym istnieją niekiedy ograniczenia dotyczące przydziału procesorów. Rozważmy system, w którym urządzenie wejścia-wyjścia jest podłączone do jednego z procesorów za pomocą prywatnej szyny. Procesy chcąc korzystać z tego urządzenia muszą trafić do tego procesora, w przeciwnym razie urządzenie byłoby dla nich niedostępne.

Jeśli mamy wiele jednakowych procesorów, to możemy zastosować *dziedzenie obciążenia* (ang. *load sharing*). Z każdym procesorem można by związać oddzielną kolejkę. Jednak w takim przypadku dochodziłoby do tego, że procesor z pustą kolejką byłby bezczynny, podczas gdy inny procesor miałby nadmiar pracy. W celu zapobieżenia tej sytuacji stosuje się wspólną kolejkę procesów gotowych do działania. Wszystkie procesy trafiają do jednej kolejki i są przydzielane do dowolnego z dostępnych procesorów.

W opisanym schemacie można zastosować jedną z dwóch metod planowania. W pierwszej z nich każdy procesor sam planuje swoje działanie. Każdy procesor przegląda kolejkę procesów gotowych, z której wybiera proces do wykonania. Jak zobaczymy w rozdz. 6, jeśli wiele procesorów próbuje korzystać ze wspólnej struktury danych i zmieniać ją, to każdy z nich musi być oprogramowany nader starannie. Należy zadbać, aby dwa procesory nie wybrały tego samego procesu, a także by nie ginęły procesy z kolejki. W drugiej metodzie unika się tych problemów dzięki wybraniu jednego procesora do pełnienia funkcji planisty pozostałych procesorów, czyli utworzeniu struktury typu „pan i sługa” (nadrzędny-podległy; ang. *master-slave*).

W niektórych systemach struktura ta jest rozbudowana o krok dalej. Jeden procesor – serwer główny – podejmuje wszystkie decyzje planistyczne, wykonuje operacje wejścia-wyjścia i inne czynności systemowe. Pozostałe procesory wykonują tylko kod użytkowy. Tego rodzaju *asymetryczne wieloprzetwarzanie* (ang. *asymmetric multiprocessing*) jest znacznie prostsze od wieloprzetwarzania symetrycznego, ponieważ dostęp do systemowych struktur danych ma tylko jeden procesor, zaspakając zapotrzebowanie na dzielenie danych.

5.5 ■ Planowanie w czasie rzeczywistym

W rozdziale 1 dokonaliśmy przeglądu systemów operacyjnych czasu rzeczywistego i omówiliśmy ich wzrastające znaczenie. Kontynuując tutaj to omówienie, przedstawiamy środki służące do planowania w czasie rzeczywistym w ramach uniwersalnego systemu komputerowego.

Programowanie w czasie rzeczywistym dzieli się na dwa rodzaje. Do wy pełniania krytycznych zadań w gwarantowanym czasie są potrzebne *rygorystyczne systemy czasu rzeczywistego* (ang. *hard real-time systems*). Ogólnie mówiąc, proces jest dostarczany wraz z instrukcją określającą ilość czasu, której wymaga on do zakończenia działania lub wykonania operacji wejścia-wyjścia. Na podstawie tych danych planista akceptuje ów proces, zapewniając jego wykonanie na czas, lub odrzuca zlecenie jako niewykonalne. Określa się to terminem *rezerwacji zasobów* (ang. *resource reservation*). Udzielenie takiej gwarancji wymaga, aby planista miał dokładne rozeznanie w czasie zużywanym przez wszelkiego rodzaju funkcje systemu operacyjnego, toteż każdej operacji powinno się zagwarantować maksymalny czas jej wykonania. Jak wykażemy w kilku następnych rozdziałach, zapewnienie takich gwarancji nie jest możliwe w systemie z pamięcią pomocniczą lub wirtualną, gdyż w podsystemach tych występują nieuniknione i nieprzewidywalne odchylenia czasu wykonania poszczególnych procesów. Dlatego w skład rygorystycznych systemów czasu rzeczywistego wchodzi specjalne oprogramowanie, działające na sprycie przypisany na stałe do krytycznych procesów, i nie występują w nich wszystkie właściwości spotykane we współczesnych komputerach i systemach operacyjnych.

Łagodne systemy czasu rzeczywistego (ang. *soft real-time systems*) są mniej restrykcyjne. Wymaga się w nich, aby procesy o decydującym znaczeniu miały priorytet nad słabiej sytuowanymi. Chociaż dodanie do systemu z podziałem czasu możliwości przetwarzania w łagodnie pojmowanym czasie rzeczywistym może spowodować niesprawiedliwy przydział zasobów i zaowocować większymi opóźnieniami lub nawet głodzeniem niektórych procesów, jednak jest przynajmniej możliwe do osiągnięcia. Otrzymujemy w wyniku uniwersalny system, mogący również obsługiwać urządzenia multimedialne, realizować szybkie funkcje graficzne i wykonywać rozmaite inne zadania, które nie działałyby zadowalająco w środowisku nie zapewniającym możliwości wykonywania obliczeń w łagodnym trybie czasu rzeczywistego.

Implementacja łagodnego trybu przetwarzania w czasie rzeczywistym wymaga starannego zaprojektowania planisty i powiązanych z nim elementów systemu operacyjnego. Po pierwsze, system musi mieć planowanie priorytetowe, a procesy działające w czasie rzeczywistym muszą mieć najwyższy

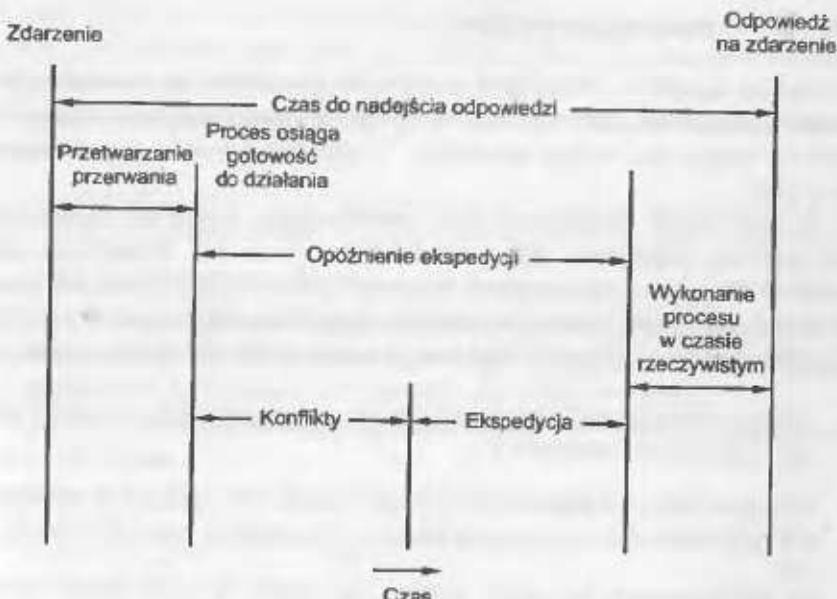
priorytet. Priorytety procesów czasu rzeczywistego nie mogą maleć z upływem czasu, niezależnie od tego, że może to dotyczyć priorytetów procesów wykonywanych poza trybem czasu rzeczywistego. Po drugie, opóźnienie ekspediowania procesów do procesora musi być małe. Im będzie ono mniejsze, tym szybciej proces czasu rzeczywistego będzie mógł rozpoczęć działanie, poczynając od chwili, w której jest do niego gotowy.

Spełnienie pierwszego z tych wymagań jest stosunkowo proste. Możemy na przykład zakazać postarzania procesów w obecności procesów czasu rzeczywistego, zapewniając niezmienność ich priorytetów. Bardziej skomplikowane jest natomiast spełnienie drugiego wymagania. Trudność polega na tym, że w wielu systemach operacyjnych, w tym także w większości wersji systemu UNIX, przed przełączeniem kontekstu należy poczekać do zakończenia funkcji systemowej lub do zablokowania procesu z powodu operacji wejścia-wyjścia. Opóźnienie ekspedycji w takich systemach może być znaczne, gdyż niektóre z funkcji systemowych są złożone, a pewne urządzenia zewnętrzne działają powoli.

Aby utrzymać opóźnienie ekspedycji na niskim poziomie, musimy zzwolić na wywłaszczenie funkcji systemowych. Cel ów można osiągnąć kilkoma sposobami. Jeden z nich polega na wstawianiu do długotrwałych funkcji systemowych tzw. punktów wywłaszczeń (ang. *preemption points*), w których sprawdza się, czy jakiś wysokopriorytetowy proces nie wymaga aktywniania. Jeśli tak jest, to następuje przełączenie kontekstu i działanie przerwanej funkcji systemowej podejmuje się dopiero po wykonaniu procesu o wysokim priorytecie. Punkty wywłaszczeń można umieszczać tylko w „bezpiecznych” miejscach jądra, tj. w takich, w których nie są zmieniane struktury danych jądra. Pomimo zastosowania punktów wywłaszczeń opóźnienia ekspedycji mogą być duże, ponieważ w praktyce do jądra można dodać niewiele takich punktów.

Inna metoda wywłaszczenia polega na spowodowaniu, aby całe jądro było wywłaszczone. W celu zapewnienia poprawności działań wszystkie struktury danych jądra muszą być chronione za pomocą różnorodnych mechanizmów synchronizacji, które są omówione w rozdz. 6. W tej metodzie jądro można wywłaszczyć w dowolnej chwili, gdyż wszystkie aktualizowane dane jądra są chronione przed zmianami za pomocą wysokopriorytetowego procesu. Taką metodę zastosowano w systemie Solaris 2.

Co się jednak stanie, jeśli proces o wyższym priorytecie zechce przeczyć lub zmienić dane jądra w chwili, w której korzysta z nich inny proces o niższym priorytecie? Wysokopriorytetowy proces musiałby czekać na zakończenie pracy procesu o nizszym priorytecie. Sytuacja ta nosi nazwę *odwrócenia priorytetów* (ang. *priority inversion*). W rzeczywistości może powstawać łańcuch procesów korzystających z zasobów potrzebnych procesowi



Rys. 5.8 Opóźnienie ekspedycji

wysokopriorytetowemu. Ten problem można rozwiązać za pomocą *protokołu dziedziczenia priorytetów* (ang. *priority-inheritance protocol*), w którym wszystkie owe procesy (tj. te które używają zasobów potrzebnych procesowi wysokopriorytetowemu) dziedziczą wysoki priorytet dopóty, dopóki nie przestaną korzystać z zasobów będących przedmiotem sporu. Po skończeniu działania tych procesów ich priorytety uzyskują swoje pierwotne wartości.

Na rysunku 5.8 widać, co składa się na opóźnienie ekspedycji. *Faza konfliktowa* (ang. *conflict phase*) w opóźnieniu ekspedycji obejmuje trzy części:

1. Wywłaszczenie procesu działającego w jądrze.
2. Zwolnienie przez niskopriorytetowe procesy zasobów potrzebnych procesowi wysokopriorytetowemu.
3. Przelączanie kontekstu z procesu bieżącego do procesu wysokopriorytetowego.

Jako przykład podamy, że w systemie Solaris 2 opóźnienie ekspedycji przy zakazie wywłaszczenia przekracza 200 ms. Natomiast po zezwoleniu na wywłaszczenie opóźnienie w ekspediowaniu procesów do procesora maleje na całość do 2 ms.

5.6 ■ Ocena algorytmów

Jak wybrać algorytm planowania przydziału procesora dla konkretnego systemu? Pokazaliśmy już w p. 5.3, że istnieje wiele algorytmów planowania, z których każdy ma swoiste parametry. W rezultacie wybór algorytmu może być trudny.

Podstawowym problemem jest zdefiniowanie kryteriów stosowanych przy wyborze algorytmu. Jak powiedzieliśmy w p. 5.2, kryteria są często określane stopniem wykorzystania procesora, czasem odpowiedzi lub przepustowością. Aby móc dokonywać wyboru algorytmu, należy najpierw określić względną ważność tych miar. Pod uwagę można wziąć kilka miar, takich jak:

- maksymalizacja wykorzystania procesora przy założeniu, że maksymalny czas odpowiedzi wyniesie 1 s;
- maksymalizacja przepustowości w taki sposób, aby czas cyklu przetwarzania był (średnio) liniowo proporcjonalny do ogólnego czasu wykonania.

Po zdefiniowaniu kryteriów wyboru zechcemy dokonać oceny różnych algorytmów wziętych pod uwagę. Istnieje kilka różnych metod oszacowań, które opisujemy w p. 5.6.1-5.6.4.

5.6.1 Modelowanie deterministyczne

Jedna z głównych klas metod oceniania algorytmów nosi nazwę *oceny analitycznej* (ang. *analytic evaluation*). Dokonując oceny analitycznej, używa się danego algorytmu i obciążenia systemu pracą w celu wytworzenia wzoru lub wartości oszacowujących zachowanie algorytmu dla danego obciążenia.

Jedną z odmian oceny analitycznej jest *modelowanie deterministyczne* (ang. *deterministic modeling*). W metodzie tej przyjmuje się konkretne, z góry określone obciążenie robocze systemu i definiuje zachowanie każdego algorytmu w warunkach tego obciążenia.

Załóżmy na przykład, że obciążenie pracą przedstawia się tak jak niżej. Wszystkie procesy nadchodzą w chwili 0 oraz mają zadane uporządkowanie i wyrażone w milisekundach fazy procesora:

Proces	Czas trwania fazy
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Dla tego zbioru procesów rozważymy algorytmy planowania: FCFS, SJF i rotacyjny (o kwarcie czasu równym 10 ms). Który algorytm da minimalny średni czas oczekiwania?

Zgodnie z algorymem FCFS procesy zostaną wykonane w następującym porządku:

P_1	P_2	P_3	P_4	P_5
0	10	39	42	49

61

Czas oczekiwania wyniesie 0 ms dla procesu P_1 , 10 ms dla procesu P_2 , 39 ms dla procesu P_3 , 42 ms dla procesu P_4 i 49 ms dla procesu P_5 . Zatem średni czas oczekiwania jest równy $(0 + 10 + 39 + 42 + 49)/5 = 28$ ms.

Zgodnie z niewywłaszczającym planowaniem SJF procesy zostałyby wykonane w porządku:

P_3	P_4	P_1	P_5	P_2
0	3	10	20	32

61

Czas oczekiwania wyniesie 10 ms dla procesu P_1 , 32 ms dla procesu P_2 , 0 ms dla procesu P_3 , 3 ms dla procesu P_4 i 20 ms dla procesu P_5 . Zatem średni czas oczekiwania jest równy $(10 + 32 + 0 + 3 + 20)/5 = 13$ ms.

W algorytmie rotacyjnym proces P_2 , rozpoczęwszy działanie, po 10 ms zostaje wywłaszczony i wędruje na koniec kolejki:

P_1	P_2	P_3	P_4	P_5	P_2	P_5	P_2
0	10	20	23	30	40	50	52

61

Czas oczekiwania wynosi 0 ms dla procesu P_1 , 32 ms dla procesu P_2 , 20 ms dla procesu P_3 , 23 ms dla procesu P_4 i 40 ms dla procesu P_5 . Średni czas oczekiwania jest równy $(0 + 32 + 20 + 23 + 40)/5 = 23$ ms.

Można zauważyć, że w tym przypadku polityka SJF daje wynik czasu oczekiwania o ponad połowę lepszy od planowania FCFS, a za pomocą algorytmu RR uzyskuje się wartość pośrednią.

Modelowanie deterministyczne jest proste i szybkie. Daje dokładne liczby, pozwalając na porównywanie algorytmów. Jednakże wymaga ono dokładnych liczb na wejściu, a uzyskiwane odpowiedzi odnoszą się tylko do zadanego przypadków. Modelowanie deterministyczne znajduje głównie zastosowanie w opisywaniu algorytmów planowania i dostarczaniu przykładów. W przypadkach stale powtarzającego się wykonywania tych samych programów, przy możliwości dokładnego pomiaru wymagań związanych z ich przetwarzaniem, modelowanie deterministyczne może być wykorzystane do

wyboru algorytmu planowania. Po zastosowaniu do zbioru przykładów modelowanie deterministyczne może wykazać tendencje, które możemy osobno analizować i dowodzić. Można na przykład wykazać, że w opisanych warunkach (wszystkie procesy i ich czasy są znane w chwili 0) algorytm SJF da zawsze minimalny czas oczekiwania.

Modelowanie deterministyczne dotyczy jednakże zbyt specyficznych sytuacji i wymaga zbyt wiele dokładnej wiedzy, aby mogło zasługiwać na miano ogólnie użytecznego.

5.6.2 Modele obsługi kolejek

Procesy wykonywane w różnych systemach zmieniają się z dnia na dzień, więc trudno jest mówić o statycznym zbiorze procesów (i pomiarów czasu), który mógłby posłużyć do modelowania deterministycznego. Możliwe jest wszakże ustalenie rozkładów faz procesora i wejścia-wyjścia. Rozkłady te można mierzyć, a potem przybliżać lub po prostu oszacowywać. Wynikiem jest wzór matematyczny opisujący prawdopodobieństwo wystąpienia poszczególnych faz procesora. Na ogół jest to rozkład wykładniczy i takimi środkami jest opisywany. Jest tu również potrzebny rozkład czasów przybywania procesów do systemu. Na podstawie tych dwóch rozkładów można obliczyć średnią przepustowość, wykorzystanie procesora, czas oczekiwania itp. dane dla większości algorytmów.

System komputerowy można opisać jako sieć usługodawców, czyli serwerów. Każdy serwer ma kolejkę czekających procesów. Serwerem jest zarówno procesor z kolejką procesów gotowych do wykonania, jak i system wejścia-wyjścia z kolejkami do urządzeń. Znając tempo nadchodzenia zamówień i czasy wykonywania usług, można obliczyć wykorzystanie procesora, średnie długości kolejek, średnie czasy oczekiwania itd. Badania te należą do *analizy obsługi kolejek w sieciach* (ang. *queuing-network analysis*).

Na przykład niech n będzie średnią długością kolejki (nie licząc procesu aktualnie obsługiwanej), niech W oznacza średni czas oczekiwania w kolejce, a λ niech symbolizuje tempo przybywania nowych procesów do kolejki (np. trzy procesy na sekundę). Można wówczas przypuszczać, że w czasie W , w ciągu którego proces czeka, w kolejce pojawi się $\lambda \times W$ nowych procesów. Jeśli system ma być ustabilizowany, to liczba procesów opuszczających kolejkę musi się równać liczbie procesów przybywających do niej, czyli

$$n = \lambda \times W.$$

Powyzsza równość jest znana pod nazwą *wzoru Little'a*. Wzór Little'a jest wyjątkowo użyteczny, ponieważ obowiązuje dla dowolnego algorytmu planowania i rozkładu przybyć.

Wzoru Little'a można użyć do obliczenia jednej z trzech występujących w nim zmiennych, jeśli są znane dwie pozostałe. Jeżeli na przykład wiadomo, że w każdej sekundzie przybywa (średnio) 7 procesów oraz że w kolejce zwykłej znajduje się 14 procesów, to można obliczyć, że średni czas oczekiwania będzie wynosił 2 s na proces.

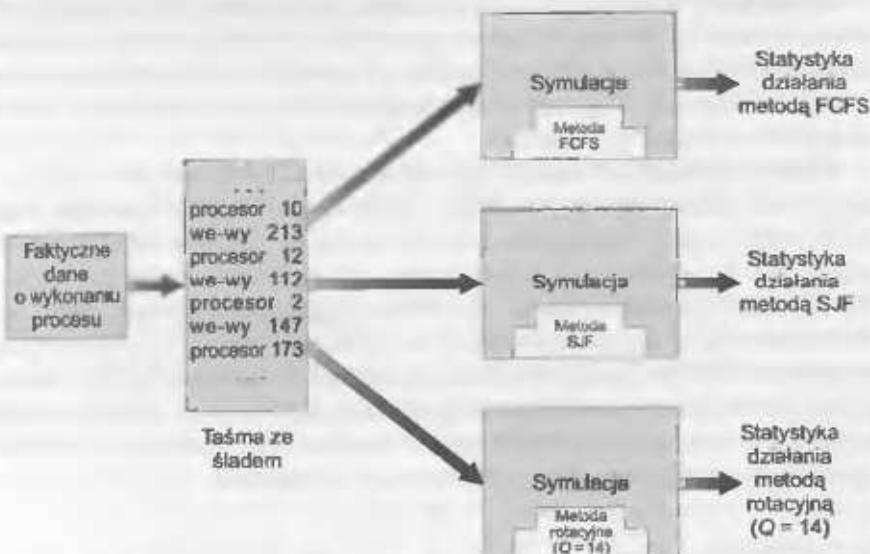
Analizowanie kolejek można z powodzeniem stosować do porównywania algorytmów, ale i ta metoda ma swoje ograniczenia. W chwili obecnej mało jest klas algorytmów i rozkładów, którymi można by się zająć. Matematyczne opracowanie skomplikowanych algorytmów lub rozkładów może być trudne. Toteż rozkłady przybyć i obsługi są często definiowane w sposób, który pozwala opracować je matematycznie, ale jest nirealistyczny. Na ogół jest przy tym potrzebna pewna liczba niezależnych założeń, które mogą być niedokładne. Choć zatem można za ich pomocą obliczyć odpowiedzi, modele obsługi kolejek są często tylko przybliżeniami rzeczywistych systemów. Wskutek tego dokładność obliczanych wyników może być wątpliwa.

5.6.3 Symulacje

Aby otrzymać dokładniejszą ocenę algorytmów planowania, można zastosować *symulację*. W tym celu trzeba zaprogramować model systemu komputerowego. Zaprogramowane struktury danych są głównymi częściami takiego systemu. Symulator ma zmienną reprezentującą zegar; w miarę przyrostu wartości tej zmiennej symulator zmienia stan systemu, aby odzwierciedlać pracę urządzeń, procesów i planisty. Podczas wykonywania programu symulacji gromadzi się i drukuje dane statystyczne obrazujące działanie algorytmu.

Dane do sterowania przebiegiem symulacji można wygenerować kilkoma sposobami. W najczęściej używanej metodzie stosuje się generator liczb losowych, z którego korzysta się w programie symulatora do tworzenia procesów, określania czasów trwania faz procesora, przybyć, odejść itd. – zgodnie z rozkładami prawdopodobieństw. Rozkłady można definiować matematycznie (równomierny, wykładniczy, Poissona) lub doświadczalnie. Jeśli rozkład ma być zdefiniowany doświadczalnie, to dokonuje się pomiarów w badanym systemie. Otrzymywane wyniki są używane do definiowania faktycznych rozkładów zdarzeń w rzeczywistym systemie, a te z kolei służą potem do sterowania symulacją.

Symulacja sterowana rozkładami może być niedokładna z powodu związków zachodzących między kolejnymi zdarzeniami w rzeczywistym systemie. Rozkład częstości pokazuje tylko liczby poszczególnych zdarzeń, nie mówiąc nic o porządku ich występowania. Do poprawienia tej usterki można użyć taśm śladów (ang. *trace tapes*). Taśma śladu powstaje w wyniku nadzorowania rzeczywistego systemu i zapisywania faktycznej kolejności



Rys. 5.9 Ocena planistów przydziału procesora za pomocą symulacji

zdarzeń (rys. 5.9). Sekwencji takiej używa się później do sterowania przebiegiem symulacji. Taśmy śladów świetnie nadają się do porównywania dwóch algorytmów dla takiego samego zbioru rzeczywistych danych. Metoda ta pozwala uzyskać dokładne wyniki dla zadanych danych wejściowych.

Symulacje mogą być kosztowne i wymagać często wielu godzin pracy komputera. Bardziej szczegółowa symulacja dostarcza dokładniejszych wyników, lecz również powoduje zużycie większej ilości czasu komputera. W dodatku taśmy śladów pochłaniają olbrzymie ilości pamięci. Również projektowanie, kodowanie i sprawdzanie poprawności symulatora może być niebagatelnym zadaniem.

5.6.4 Implementacja

Nawet symulacja ma ograniczoną dokładność. Jedynym sposobem na osiągnięcie całkowicie dokładnej oceny algorytmu planowania jest zakodowanie go, włączenie do badanego systemu operacyjnego i przypatrzenie się jego pracy. To podejście zakłada wmontowanie danego algorytmu do rzeczywistego systemu i oszacowanie w rzeczywistych warunkach operacyjnych.

Podstawową trudnością przy zastosowaniu tej metody jest jej koszt. Wydatki są związane nie tylko z zakodowaniem algorytmu i dostosowaniem do niego systemu operacyjnego i jego struktur danych, lecz także z reakcją użytkowników

na ciągle zmieniający się system operacyjny. Większości użytkowników nie zależy na budowie lepszego systemu operacyjnego; potrzeba im po prostu, aby ich procesy były wykonywane i by mogli uzyskiwać wyniki. Stale zmieniający się system operacyjny nie pomaga użytkownikom w osiąganiu ich celów.

Inną trudnością, wystającą przy każdym oszacowaniu algorytmu, jest zmieniające się środowisko, w którym się go używa. Środowisko to zmienia się nie tylko ze zwykłych powodów, takich jak powstawanie nowych programów i zmiany rodzajów stawianych problemów, lecz również wskutek zachowania się planisty. Jeśli będą preferowane krótkie procesy, to użytkownicy mogą zacząć dzielić większe procesy na zbiory mniejszych procesów. Gdy procesy interakcyjne będą miały pierwszeństwo przed procesami nieinterakcyjnymi, wówczas użytkownicy mogą chętniej pracować w trybie interakcyjnym.

Konstruktory pewnego systemu próbowali na przykład zaprojektować automatyczne klasyfikowanie procesów na interakcyjne i nieinterakcyjne, analizując liczbę operacji wejścia-wyjścia z terminali. Jeśli proces nie wykonywał żadnej transmisji na terminalu w ciągu jednej sekundy, to klasyfikowano go jako nieinterakcyjny i przesuwano do kolejki o niższym priorytecie. W odpowiedzi na to jakiś programista tak pozmieniał swoje programy, aby przesyłały na terminal dowolny znak w regularnych odstępach czasu, krótszych niż 1 s. System nadawał jego programom wysoki priorytet, mimo że wyprowadzane na terminal wyniki były całkiem bez sensu.

Najelastyczniejsze algorytmy planowania umożliwiają administratorom systemów dokonywanie w nich zmian. Podczas generowania systemu, jego rozruchu lub normalnej pracy można zmienić wartości zmiennych używanych przez procedury planujące stosownie do przewidywanych, przyszłych zadań systemu. Zapotrzebowanie na elastyczne procedury planowania jest kolejnym przykładem korzyści płynących z oddzielania mechanizmu od polityki jego użycia. Gdy na przykład trzeba natychmiast obliczyć i wydrukować listę plac, którą zazwyczaj sporządza się w trybie niskopriorytetowego zadania wsadowego, wówczas można chwilowo podwyższyć priorytet kolejki wsadowej. Niestety, niewiele systemów operacyjnych pozwala korzystać z tego rodzaju elastycznego planowania*.

5.7 ■ Podsumowanie

Planowanie przydziału procesora to zajęcie polegające na wybraniu procesu oczekującego w kolejce procesów gotowych do działania i przydzieleniu mu procesora. Przydziału procesora do wybranego procesu dokonuje ekspedytor.

* Jednym z owych nielicznych był zapomniany już dziś system operacyjny GEORGE 3 brytyjskiej firmy ICL, używany m.in. w polskich komputerach serii ODRA 1300 – Przyp. tłum.

Najprostszym algorytmem planowania jest metoda „pierwszy zgłoszony – pierwszy obsłużony” (FCFS), może ona jednak powodować opóźnianie wykonywania procesów krótkich przez procesy długie. Dowodzi się, że planowanie według schematu „najpierw najkrótsze zadanie” (SJF) jest optymalne, dając najkrótszy średni czas oczekiwania. Realizacja algorytmu SJF natychmiast zwiększa trudności związane z koniecznością przewidywania długości trwania przyszłych faz procesu. Algorytm SJF jest szczególnym przypadkiem ogólnego algorytmu planowania priorytetowego, który przydziela procesor po prostu temu procesowi, który ma najwyższy priorytet. Zarówno planowanie priorytetowe, jak i planowanie SJF może grozić głodzeniem procesu. Głodzeniu można zapobiegać za pomocą postarzania procesów.

Dla systemu z podziałem czasu (interakcyjnego) najodpowiedniejsze jest planowanie rotacyjne (RR), które pierwszemu procesowi w kolejce procesów gotowych przydziela procesor na q jednostek czasu, przy czym q nazywa się kwantem czasu. Jeśli proces nie zwolni procesora po upływie q jednostek czasu, to jest wywłaszczały i umieszczany na końcu kolejki procesów gotowych. Głównym problemem jest tu dobór kwantu czasu. Jeśli kwant jest za duży, to planowanie rotacyjne degeneruje się do planowania FCFS; gdy kwant czasu jest za mały, wówczas nakład pracy na częste przełączenia kontekstu staje się nadmierny.

Algorytm FCFS jest niewywłaszczający, natomiast algorytm rotacyjny jest wywłaszczający. Algorytm SJF i algorytmy priorytetowe mogą być wywłaszczające lub niewywłaszczające.

Algorytmy wielopoziomowego planowania kolejek pozwalają na użycie różnych algorytmów dla różnych klas procesów. Najczęściej spotyka się pierwszoplanową kolejkę procesów interakcyjnych, planowaną rotacyjnie, oraz drugoplanową kolejkę wsadową, planowaną za pomocą algorytmu FCFS. Wielopoziomowe kolejki ze sprzężeniem zwojnym pozwalają na przemieszczanie procesów z jednych kolejek do drugich.

Różnorodność dostępnych algorytmów planowania powoduje, iż potrzebne są metody pomagające wybierać właściwe algorytmy. W metodach analitycznych do określania wydajności algorytmu stosuje się analizę matematyczną. Metody symulacyjne pozwalają określić zachowanie algorytmu przez naśladowania jego działania na „reprezentatywnych” próbkach procesów i na podstawie zebranych danych obliczyć jego wydajność.

■ Ćwiczenia

- 5.1** Algorytm planowania przydziału procesora wyznacza porządek wykonywania planowanych przez niego procesów. Mając n procesów do za-

planowania dla jednego procesora, określ, ile istnieje możliwych zaplanowań. Podaj wzór zależny od n .

- 5.2 Zdefiniuj różnicę między planowaniem wywłaszczającym a niewywłaszczającym. Uzasadnij, dlaczego w ośrodku obliczeniowym czyste planowanie niewywłaszczające byłoby trudne do przyjęcia.
- 5.3 Rozważmy następujący zbiór procesów, których długości faz procesora w milisekundach wynoszą odpowiednio:

Proces	Czas trwania fazy	Priorytet
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Zakładamy, że procesy nadeszły czasie 0 w kolejności P_1, P_2, P_3, P_4, P_5 .

- (a) Narysuj cztery diagramy Gantta ilustrujące wykonanie tych zadań przy użyciu algorytmu FCFS, SJF, niewywłaszczającego algorytmu priorytetowego (mniejszy numer priorytetu oznacza wyższy priorytet) i algorytmu rotacyjnego (kwant czasu = 1).
- (b) Jaki będzie czas cyku przetwarzania każdego procesu w każdym z algorytmów planowania wymienionych w części (a)?
- (c) Jaki będzie czas oczekiwania każdego procesu w każdym z algorytmów planowania wymienionych w części (a)?
- (d) Który z planistów z części (a) daje minimalny średni czas oczekiwania (dla ogółu procesów)?
- 5.4 Założymy, że poniższe procesy nadchodzą do wykonania w podanych chwilach. Każdy proces będzie działał przez czas podany w tabeli. Odpowiadając na dalsze pytania, zastosuj planowanie niewywłaszczające i podejmuj każdą decyzję na podstawie informacji, którymi będziesz dysponować wtedy, kiedy będzie należało ją podjąć.

Proces	Czas nadania	Czas fazy
P_1	0,0	8
P_2	0,4	4
P_3	1,0	1

- (a) Ile wyniesie średni czas cyku przetwarzania tych procesów przy planowaniu metodą FCFS?

- (b) Ile wyniesie średni czas cyklu przetwarzania tych procesów przy planowaniu według algorytmu SJF?
- (c) Zaktłada się, że algorytm SJF poprawi wydajność, warto jednak zauważyc, że proces P_1 zostanie wybrany w chwili 0, ponieważ nie wiadomo, iż niebawem nadziejają dwa krótsze procesy. Oblicz, ile wyniósłby czas średni cyklu przetwarzania, gdyby przez pierwszą jednostkę czasu pozostawiono procesor bezczynny i dopiero potem zastosowano planowanie SJF. Należy pamiętać, że procesy P_1 i P_2 będą czekały w okresie tej bezczynności, zatem ich czas oczekiwania się zwiększy. Algorytm tego rodzaju mógłby się nazywać planowaniem na podstawie przyszłej wiedzy.
- 5.5 Rozważ wariant algorytmu planowania rotacyjnego, w którym pozycje w kolejce procesów gotowych są wskaźnikami do bloków kontrolnych procesów.
- Co się stanie w wyniku wstawienia do kolejki procesów gotowych dwu wskaźników do tego samego procesu?
 - Jakie będą główne zalety i wady takiego postępowania?
 - Jak można by zmienić podstawowy algorytm rotacyjny, aby uzyskać ten sam efekt bez podwajania wskaźników?
- 5.6 Jakie są zalety dysponowania różnymi kwantami czasu na różnych poziomach kolejek w systemie z wielopoziomowym układem kolejek?
- 5.7 Rozważmy następujący wywlaściżający, priorytetowy algorytm planowania, oparty na dynamicznych zmianach priorytetów. Większe numery priorytetów oznaczają wyższe priorytety. Gdy proces czeka na procesor (jest w kolejce procesów gotowych, ale nie jest wykonywany), wówczas jego priorytet zmienia się w tempie α , kiedy zaś jest wykonywany, wtedy jego priorytet zmienia się w tempie β . Wszystkie procesy w chwilach wchodzenia do kolejki procesów gotowych otrzymują priorytet 0. Wartości parametrów α i β można określić, otrzymując wiele różnych algorytmów planowania.
- Jaki algorytm powstaje, gdy $\beta > \alpha > 0$?
 - Jaki algorytm powstaje, gdy $\alpha < \beta < 0$?
- 5.8 Wiele algorytmów planowania przydziału procesora podlega parametryzacji. Na przykład algorytm rotacyjny wymaga parametru określającego kwant czasu. Wielopoziomowe kolejki ze sprzężeniem zwrotnym wymagają parametrów definiujących liczbę kolejek, algorytm

planowania dla każdej kolejki, kryteria przemieszczania procesów między kolejkami itd.

Algorytmy takie są więc w istocie zbiorami algorytmów (np. zbiór algorytmów rotacyjnych dla wszystkich kwantów czasu itp.) Jeden zbiór algorytmów może zawierać inny (np. algorytm FCFS jest algorytmem rotacyjnym o nieskończonym kwancie czasu). Czy i jakie relacje zachodzą między następującymi parami zbiorów algorytmów:

- (a) priorytetowymi i SJF,
 - (b) wielopoziomowymi kolejkami ze sprzężeniem zwrotnym i FCFS,
 - (c) priorytetowymi i FCFS,
 - (d) rotacyjnymi i SJF.
- 5.9** Założmy, że algorytm planowania (na poziomie krótkoterminowego planowania przydziału procesora) faworyzuje procesy, które zużyły najmniej czasu procesora w przeszłości. Dlaczego algorytm ten będzie faworyzował programy ograniczone przez wejście-wyjście oraz te spośród programów ograniczonych przez procesor, które na razie nie cierpią nieustannie z powodu jego braku?
- 5.10** Wyjaśnij, czy i w jakim stopniu każdy z następujących algorytmów planowania przedłada krótkie procesy ponad inne:
- (a) FCFS,
 - (b) rotacyjny,
 - (c) wielopoziomowe kolejki ze sprzężeniami zwrotnymi.

Uwagi bibliograficzne

Lamson w artykule [234] dostarczył ogólnych rozważań dotyczących planowania. Bardziej formalne potraktowanie teorii planowania podają w swoich książkach: Kleinrock [215], Sauer i Chandy [376] oraz Lazowska i in. [241]. Ujednolicone podejście do planowania zaprezentowali w artykule [362] Russchitzka i Fabry. Haldar i Subramanian [161] omawiają zagadnienie sprawiedliwego planowania w systemach z podziałem czasu.

Kolejki ze sprzężeniami zwrotnymi zostały wprowadzone po raz pierwszy w systemie CTSS opisany przez Corbató i in. [86]. Ten system obsługi kolejek poddał analizie Schrage [380]; odmiany wielopoziomowych kolejek ze sprzężeniami zwrotnymi przeanalizowali Coffman i Kleinrock [76]. Uzu-

pełniające studia zaprezentowali Coffman i Denning [75] oraz Svobodova [411]. Struktury danych do manipulowania kolejkami priorytetowymi przedstawił Vuillemin [436]. Algorytm wywiaszczającego planowania priorytetowego w ćwiczeniu 5.9 jest pomysłu Kleinrocka [215].

Planowanie wątków omówili Anderson i in. [10]. Rozważania na temat planowania wieloprocesorów zaprezentowali: Jones i Schwarz [201], Tucker i Gupta [429], Zahorjan i McCann [447], Feitelson i Rudolph [131] oraz Leutenegger i Vernon [251].

Problemy planowania w systemach czasu rzeczywistego przedstawili: Liu i Layland [261], Abbot [1], Jensen i in. [198], Hong i in. [179] oraz Khanra i in. [210]. Specjalne wydanie periodyku *Operating System Review*, poświęcone systemom czasu rzeczywistego, ukazało się pod redakcją Zhao [449]. Ekyholt i in. [128] opisali elementy czasu rzeczywistego w systemie Solaris 2. Procedury planujące sprawiedliwe przydziały rozpatrywali: Henry [170], Woodside [441] oraz Kay i Lauder [204].

Omówienia zasad planowania zastosowanych w systemach operacyjnych dokonali: Samson [367] dla systemu operacyjnego MVS, Iacobucci [185] dla systemu operacyjnego OS/2, Bach [20] dla systemu operacyjnego UNIX V oraz Black [41] dla systemu operacyjnego Mach.

Rozdział 6

SYNCHRONIZOWANIE PROCESÓW

Proces współpracujący może wpływać na inne procesy w systemie lub podlegać ich oddziaływaniom. Procesy współpracujące mogą bezpośrednio dzielić logiczną przestrzeń adresową (tzn. zarówno kod, jak i dane) albo zezwala się im na dzielenie danych tylko za pośrednictwem plików. Pierwszą możliwość osiąga się za pomocą *procesów lekkich*, czyli *wątków*, które omówiliśmy w p. 4.5. Współbieżny dostęp do danych dzielonych może powodować ich niespójność. W tym rozdziale przedstawiamy różne mechanizmy zachowywania spójności danych przez uporządkowane wykonywanie współpracujących ze sobą procesów użytkujących wspólnie logiczną przestrzeń adresową.

6.1 ■ Podstawy

W rozdziale 4 zaprezentowaliśmy model systemu złożonego z pewnej liczby współpracujących ze sobą procesów sekwencyjnych, wykonywanych asynchronicznie i być może korzystających ze wspólnych danych. Model ten zilustrowaliśmy za pomocą schematu ograniczonego buforowania, który jest typowy dla systemów operacyjnych.

Powróćmy do rozwiązania problemu ograniczonego buforowania z użyciem wspólnej pamięci, które przedstawiliśmy w p. 4.4. Jak zaznaczyliśmy, nasze rozwiązanie pozwalało na pomieszczenie w tym samym czasie w buforze co najwyżej $n - 1$ jednostek. Przypuśćmy, że chcemy usunąć tę wadę, zmieniając algorytm. Można w tym celu dodać zmienną całkowitą *licznik* z nadaną wartością początkową 0. Zmienna *licznik* będzie zwiększana przy

każdym dołączeniu do bufora nowej jednostki i zmniejszana przy każdym usuwaniu jednostki z bufora. Kod procesu producenta można zmodyfikować następująco:

repeat

... produkuj jednostkę w *nastp*

while *licznik* = *n* **do** *nic*;

bufor[we] := *nastp*;

we := *we* + 1 **mod** *n*;

licznik := *licznik* + 1;

until *false*;

Kod procesu konsumenta można zmodyfikować następująco:

repeat

while *licznik* = 0 **do** *nic*;

nastk := *bufor[wy]*;

wy := *wy* + 1 **mod** *n*;

licznik := *licznik* - 1;

... konsumowanie jednostki z *nastk*

until *false*;

Choć obie procedury (producent i konsument) są poprawne, jeśli są rozpatrywane z osobna, mogą nie działać właściwie, gdy będą wykonane wspólnie. Dla zilustrowania tego założymy, że bieżąca wartość zmiennej *licznik* wynosi 5 oraz że producent i konsument wykonują wspólnie instrukcje *licznik* := *licznik* + 1 i *licznik* := *licznik* - 1. Po wykonaniu tych dwu instrukcji wartość zmiennej *licznik* może wynieść 4, 5 albo 6! Poprawny jest tylko wynik *licznik* = 5, który uzyskuje się wówczas, gdy producent i konsument działają oddzielnie.

Oto jak można wykazać, że wartość zmiennej *licznik* może być niepoprawna. Zauważmy, że instrukcja *licznik* := *licznik* + 1 może być przetłumaczona na język maszynowy (typowej maszyny) jako ciąg rozkazów

```
rejestr1 := licznik ;
rejestr1 := rejestr1 + 1 ;
licznik := rejestr1;
```

przy czym *rejestr₁* jest lokalnym rejestrem procesora. Podobnie, przekład na język maszynowy instrukcji *licznik := licznik - 1* może mieć postać

```
rejestr2 := licznik;
rejestr2 := rejestr2 - 1;
licznik := rejestr2;
```

przy czym *rejestr₂* oznacza inny, lokalny rejestr procesora. Nawet jeśli *rejestr₁* i *rejestr₂* byłyby tym samym rejestrem fizycznym (powiedzmy – akumulatorem), to należy pamiętać, że zawartość takiego rejestrów podlega przechowaniu i odtwarzaniu przez procedurę obsługi przerwań (p. 2.1).

Współbieżne wykonanie instrukcji *licznik := licznik + 1* oraz *licznik := licznik - 1* jest równoważne wykonaniu sekwencyjnemu, w którym podane powyżej rozkazy w języku maszynowym mogą się przeplatać w dowolnym porządku (przy zachowaniu porządku wewnętrz instrukcji wyższego poziomu). Jednym z takich przeplotów może być

<i>T₀: producent</i>	wykonaj	<i>rejestr₁ := licznik</i>	{ <i>rejestr₁ = 5</i> }
<i>T₁: producent</i>	wykonaj	<i>rejestr₁ := rejestr₁ + 1</i>	{ <i>rejestr₁ = 6</i> }
<i>T₂: konsument</i>	wykonaj	<i>rejestr₂ := licznik</i>	{ <i>rejestr₂ = 5</i> }
<i>T₃: konsument</i>	wykonaj	<i>rejestr₂ := rejestr₂ - 1</i>	{ <i>rejestr₂ = 4</i> }
<i>T₄: producent</i>	wykonaj	<i>licznik := rejestr₁</i>	{ <i>licznik = 6</i> }
<i>T₅: konsument</i>	wykonaj	<i>licznik := rejestr₂</i>	{ <i>licznik = 4</i> }

Zauważmy, że otrzymujemy niepoprawny stan *licznik = 4*, znamionujący istnienie 4 zajętych pozycji w buforze, podczas gdy faktycznie jest ich pięć. Gdyby odwrócić porządek instrukcji *T₄* i *T₅*, otrzymany stan byłby również niepoprawny i wynosił: *licznik = 6*.

Przyczyną powstawania tych błędów jest zezwolenie na to, aby oba procesy manipulowały zmienną *licznik* współbieżnie. Tego rodzaju sytuacja, w której kilka procesów współbieżnie sięga po te same dane i wykonuje na nich działania, wskutek czego wynik tych działań zależy od porządku, w jakim następował dostęp do danych, nazywa się *szkodliwą rywalizacją* (ang. *race condition*). Aby ustrzec się przed szkodliwą rywalizacją w powyższym przypadku, powinniśmy zapewnić, że tylko jeden proces w danym czasie może operować na zmiennej *licznik*. To z kolei wymaga jakiejś formy synchronizacji obu procesów. Sytuacje takie występują często w systemach operacyjnych, gdy różne części systemu manipulują zasobami, a my nie życzymy sobie, aby wykonywane zmiany zaburzały się wzajemnie. Większa część tego rozdziału jest poświęcona zagadnieniu synchronizacji i koordynacji procesów.

6.2 ■ Problem sekcji krytycznej

Rozważmy system złożony z n procesów $\{P_0, P_1, \dots, P_{n-1}\}$. Każdy proces ma segment kodu zwany *sekcją krytyczną* (ang. *critical section*), w którym może zmieniać wspólne zmienne, aktualizować tablice, pisać do pliku itd. Ważną cechą tego systemu jest to, że gdy jeden proces wykonuje sekcję krytyczną, wówczas żaden inny proces nie jest dopuszczony do wykonywania swojej* sekcji krytycznej. Zatem wykonanie sekcji krytycznych przez procesy podlega *wzajemnemu wykluczaniu* (wzajemnemu wyłączeniu; ang. *mutual exclusion*) w czasie. Problem sekcji krytycznej polega na skonstruowaniu protokołu, który mógłby posłużyć do organizowania współpracy procesów. Każdy proces musi prosić o pozwolenie na wejście do swojej sekcji krytycznej. Fragment kodu realizującego taką prośbę nazywa się *sekcją wejściową* (ang. *entry section*). Po sekcji krytycznej może występować *sekcja wyjściowa* (ang. *exit section*). Pozostały kod nazywa się *resztą* (ang. *remainder section*).

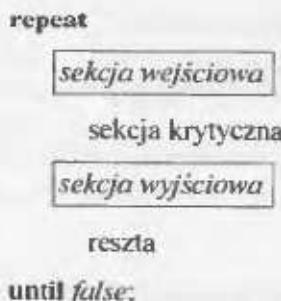
Rozwiązywanie problemu sekcji krytycznej musi spełniać następujące trzy warunki:

- Wzajemne wykluczanie:** Jeśli proces P_i działa w swojej sekcji krytycznej, to żaden inny proces nie działa w sekcji krytycznej.
- Postęp:** Jeśli żaden proces nie działa w sekcji krytycznej oraz istnieją procesy, które chcą wejść do sekcji krytycznych, to tylko procesy nie wykonujące swoich reszt mogą kandydować jako następne do wejścia do sekcji krytycznych i wybór ten nie może być odwlekany w nieskończoność.
- Ograniczone czekanie:** Musi istnieć wartość graniczna liczby wejść innych procesów do ich sekcji krytycznych po tym, gdy dany proces zgłosił chęć wejścia do swojej sekcji krytycznej i zanim uzyskał na to pozwolenie.

Zakładamy, że każdy proces jest wykonywany z szybkością niezerową. Natomiast nie musimy niczego zakładać o względnej szybkości każdego z n procesów.

W punktach 6.2.1 i 6.2.2 dochodzimy do rozwiązań problemu sekcji krytycznej, spełniających wymienione trzy warunki. W rozwiązaniach tych nie przyjmuje się żadnych założeń o rozkazach maszynowych lub liczbie procesorów. Założono jednak, że podstawowe rozkazy maszynowe (takie jak:

* Od samego początku warto podkreślić, że sekcje krytyczne rozpatrywane od strony ich kodu mogą nie mieć ze sobą nic wspólnego w różnych procesach. To, co je łączy, to dostęp do tych samych danych. – Przyp. tłum.



Rys. 6.1 Ogólna struktura typowego procesu P_i

pobierz, przechowaj lub sprawdź są wykonywane niepodzielnie. Oznacza to, że jeżeli dwa takie rozkazy są wykonywane współbieżnie, to wynik jest równowazny wykonaniu ich w sposób sekwencyjny w pewnym nieznanym porządku. Jeśli zatem rozkazy *pobierz* i *przechowaj* będą wykonane współbieżnie, to rozkaz *pobierz* natrafi na starą albo nową wartość w pamięci, ale nie na jakąś kombinację ich obu.

Przy prezentacji algorytmu definiujemy tylko zmienne używane do celów synchronizacji i opisujemy tylko typowy proces P_i , którego ogólną strukturę widać na rys. 6.1. Sekcja wejściowa i sekcja wyjściowa będą na rysunkach ujęte w ramki w celu zaakcentowania tych ważnych segmentów kodu.

6.2.1 Rozwiązania dla dwu procesów

Naszą uwagę skupimy najpierw na algorytmach odnoszących się tylko do dwóch procesów przebiegających w tym samym czasie. Procesom tym nadamy oznaczenia P_0 i P_1 . Dla wygody przy omawianiu P_i będziemy przez P_j oznaczać drugi proces, tzn. $j = 1 - i$.

6.2.1.1 Algorytm 1

W pierwszym podejściu pozwolimy procesom dzielić wspólną zmienną całkowitą *numer*, przyjmującą na początku wartość 0 (lub 1). Jeśli *numer* = i , to w sekcji krytycznej pozwala się pracować procesowi P_i . Struktura procesu P_i jest pokazana na rys. 6.2.

Rozwiązanie to gwarantuje, że tylko jeden proces w danym czasie może znaleźć się w sekcji krytycznej. Jednakże nie spełnia ono warunku postępu, ponieważ narzuca ścisłe naprzemienne wykonywanie przez procesy sekcji krytycznych. Jeśli na przykład *numer* = 0 i proces P_1 jest gotowy do wejścia do swojej sekcji krytycznej, to nie będzie mógł tego uczynić, pomimo że proces P_0 może już wykonywać resztę kodu – poza sekcją krytyczną.

```

repeat
  while numer ≠ i do nic;
    sekcja krytyczna
    numer := j;
  reszta
until false;

```

Rys. 6.2 Struktura procesu P , w algorytmie 1

6.2.1.2 Algorytm 2

Wada algorytmu 1 polega na tym, że nie zachowuje on wystarczającej informacji o stanie każdego procesu; pamięta tylko, któremu procesowi wolno wejść do jego sekcji krytycznej. Aby temu zaradzić, możemy zastąpić zmienią *numer* następującą tablicą:

```
var znacznik: array [0..1] of boolean;
```

Początkowe wartości elementów tej tablicy są równe *false*. Jeżeli *znacznik* [*i*] jest równy *true*, to oznacza, że proces P_i jest gotowy do wejścia do sekcji krytycznej. Struktura procesu P , jest przedstawiona na rys. 6.3.

W tym algorytmie proces P , najpierw nadaje zmiennej *znacznik*[*i*] wartość *true*, sygnalizując, że jest gotowy do wejścia do swojej sekcji krytycznej. Następnie proces P , sprawdza, czy proces P_j nie jest również gotowy do wejścia do swojej sekcji krytycznej. Jeśli proces P_j będzie gotowy do wejścia, to proces P , zaczeka dopóty, dopóki proces P_j nie zasygnalizuje, że nie musi już dłużej przebywać w sekcji krytycznej (nadając zmiennej *znacznik*[*j*] wartość

```

repeat
  znacznik[i] := true;
  while znacznik[j] do nic;
    sekcja krytyczna
    znacznik[i] := false;
  reszta
until false;

```

Rys. 6.3 Struktura procesu P , w algorytmie 2

false). Wówczas proces P , będąc mógł wejść do sekcji krytycznej. Przy opuszczaniu sekcji krytycznej proces P , nada swojemu znacznikowi wartość *false*, pozwalając drugiemu procesowi (jeśliby akurat oczekiwany) na wejście do sekcji krytycznej.

To rozwiązanie spełnia warunek wzajemnego wykluczania. Niestety, nie jest spełniony warunek postępu. Aby to zilustrować, rozważmy następujący ciąg wykonan:

$$\begin{aligned} T_0: & P_0 \text{ ustala } \text{znacznik}[0] = \text{true} \\ T_1: & P_1 \text{ ustala } \text{znacznik}[1] = \text{true} \end{aligned}$$

po których procesy P_0 i P_1 zapętlają się na zawsze w swoich instrukcjach `while`.

Algorytm ten jest bardzo uzależniony od dokładnych następstw czasowych obu procesów. Przedstawiona sekwencja może się zdarzyć w środowisku kilku procesorów działających współbieżnie lub jeśli jakieś przerwanie (np. od czasomierza) wystąpi bezpośrednio po wykonaniu kroku T_0 i procesor zostanie przełączony z jednego procesu do drugiego.

Zauważmy, że zamiana kolejności instrukcji określenia wartości $\text{znacznik}[i]$ i sprawdzania wartości $\text{znacznik}[j]$ nie rozwiąże naszego problemu. Doprowadzi to natomiast do sytuacji, w której stanie się możliwe przebywanie obu procesów w sekcji krytycznej w tym samym czasie, co naruszy warunek wzajemnego wykluczania.

6.2.1.3 Algorytm 3

Lącząc ze sobą pomysły zawarte w algorytmie 1 i algorytmie 2, otrzymujemy poprawne rozwiązanie problemu sekcji krytycznej, spełniające wszystkie trzy wymagania. Procesy mają dwie wspólne zmienne:

```
var znacznik: array [0..1] of boolean;
    numer: 0..1;
```

Na początku $\text{znacznik}[0] = \text{znacznik}[1] = \text{false}$, a wartość zmiennej *numer* jest nieistotna (ale wynosi 0 lub 1). Strukturę procesu P , widać na rys. 6.4.

Aby wejść do sekcji krytycznej, proces P , najpierw nadaje zmiennej $\text{znacznik}[i]$ wartość *true*, po czym zakłada, że również drugi proces chce wejść do sekcji krytycznej, jeśli to będzie możliwe ($\text{numer} = j$). Jeśli oba procesy próbują wejść w tym samym czasie, to zmienna *numer* otrzyma zarówno wartość i , jak i wartość j prawie w tym samym czasie. Tylko jedno z tych przypisów zostanie utrwalone. Drugie pojawi się, lecz zostanie natychmiast zastąpione nowym. Ostateczna wartość zmiennej *numer* zadecyduje o tym,

repeat

```
znacznik[i] := true;
numer := j;
while (znacznik[j] and numer = j) do nic;
```

sekcja krytyczna

```
znacznik[i] := false;
```

reszta

until false;

Rys. 6.4 Struktura procesu P_i w algorytmie 3

któremu z dwóch procesów przypadnie w udziale prawo do wejścia do sekcji krytycznej w pierwszej kolejności.

Udowodnimy teraz, że opisane rozwiązanie jest poprawne. Musimy wykazać, że:

1. Zapewnione jest wzajemne wykluczanie.
2. Spełniony jest warunek postępu.
3. Zachodzi warunek ograniczonego czekania.

Aby dowieść warunku 1, zauważmy, że każdy proces P_i wchodzi do swojej sekcji krytycznej tylko wtedy, gdy $znacznik[j] = \text{false}$ lub $numer = i$. Zauważmy również, że gdyby oba procesy mogły działać jednocześnie w swoich sekcjach krytycznych, to byłaby spełniona równość $znacznik[0] = znacznik[1] = \text{true}$. Z tych dwóch uwag wynika, że procesy P_0 i P_1 nie mogą skutecznie wykonywać swoich instrukcji **while** niemal w tym samym czasie, ponieważ wartość zmiennej $numer$ może być równa 0 albo 1, ale nie jednocześnie. Wobec tego podczas gdy jeden z procesów, powiedzmy P_j , będzie już zajęty wykonywaniem instrukcji **while**, procesowi P_i zostanie jeszcze do wykonania przynajmniej jedna instrukcja ($numer = j$). Ponieważ jednak w tym czasie $znacznik[j] = \text{true}$ i $numer = i$, przy czym warunek ten jest spełniony przez cały czas pobytu procesu P_i w sekcji krytycznej, wniosek brzmi: wzajemne wykluczanie jest zachowane.

Aby dowieść warunków 2 i 3 zauważmy, że proces P_i można powstrzymać przed wejściem do sekcji krytycznej tylko wtedy, gdy utknie w pętli **while** z warunkiem $znacznik[i] = \text{true}$ i $numer = j$; jest to jedyna pętla. Jeśli P_i nie jest gotowy do wejścia do sekcji krytycznej, to $znacznik[j] = \text{false}$ i do-

swojej sekcji krytycznej może wejść proces P_i . Jeśli P_i przypisał zmiennej $značnik[j]$ wartość *true* i też wykonuje swoją pętlę *while*, to albo $numer = i$, albo $numer = j$. Jeśli $numer = i$, to do sekcji krytycznej wejdzie proces P_j . Jeśli $numer = j$, to do sekcji krytycznej wejdzie proces P_i . Jednak z chwilą opuszczenia sekcji krytycznej proces P_i nada zmiennej $značnik[j]$ wartość *false*, co pozwoli procesowi P_j wejść do swojej sekcji krytycznej. Jeśli P_j nada zmiennej $značnik[j]$ wartość *true*, to musi także ustawić $numer = i$. W tej sytuacji proces P_i , który nie zmienia wartości zmiennej $numer$ podczas wykonywania pętli *while*, wejdzie do sekcji krytycznej (postęp) po co najwyżej jednym wejściu P_i (warunek ograniczonego czekania).

6.2.2 Rozwiązania wieloprocesowe

Wykazaliśmy, że algorytm 3 rozwiązuje problem sekcji krytycznej dla dwóch procesów. Teraz zajmiemy się algorymem rozwiązym problemem sekcji krytycznej dla n procesów. Jest to tzw. *algorytm piekarni* (ang. *bakery algorithm*), wzorowany na algorytmie planowania praktykowanym w piekarniach, lodziarniach, sklepach mięsnych, urzędach rejestracji samochodów i innych miejscach, w których chaos wymaga uporządkowania. Algorytm ten opracowano dla środowisk rozproszonych, lecz w tym miejscu skoncentrujemy się tylko na tych jego aspektach, które dotyczą środowiska scentralizowanego.

Przy wejściu do sklepu każdy klient dostaje numerkę. Obsługę rozpoczęnia się od klienta z najmniejszym numerem. Niestety, algorytm piekarni nie gwarantuje, że dwa procesy (klienci) nie dostaną tego samego numeru. W przypadku takiej kolizji jako pierwszy zostanie obsłużony proces o wcześniejszej nazwie. Tak więc, jeśli P_i i P_j otrzymają ten sam numer oraz $i < j$, to P_i będzie obsłużony najpierw. Ponieważ nazwy procesów są jednoznaczne i całkowicie uporządkowane, algorytm jest w pełni deterministyczny.

W algorytmie są używane następujące wspólne struktury danych:

```
var wybrane: array [0..n - 1] of boolean;
numer: array [0..n - 1] of integer;
```

Na początku elementy tych struktur danych przyjmują odpowiednio wartości *false* i 0. Zdefiniujemy, dla wygody, następującą notację:

- $(a, b) < (c, d)$, jeśli $a < c$ lub jeśli $a = c$ i $b < d$;
- $\max(a_0, \dots, a_{n-1})$ jest taką liczbą k , że $k \geq a_i$ dla $i = 0, \dots, n-1$.

Struktura procesu P_n , stosowanego w algorytmie piekarni, jest przedstawiona na rys. 6.5.

repeat

```

wybrane[i] := true;
numer[i] := max(numer[0], numer[1], ..., numer[n - 1]) + 1;
wybrane[i] := false;
for j := 0 to n - 1
  do begin
    while wybrane[j] do nic;
    while numer[j] ≠ 0
      and (numer[j], j) < (numer[i], i) do nic;
  end;

```

sekcja krytyczna

```
numer[i] := 0;
```

reszta

until false;

Rys. 6.5 Struktura procesu P_i w algorytmie piekarni

Aby udowodnić, że algorytm piekarni jest poprawny, trzeba najpierw pokazać, że jeżeli proces P_i znajduje się w swojej sekcji krytycznej, a proces P_k ($k \neq i$) ma już wybrany swój $numer[k] \neq 0$, to $(numer[i], i) < (numer[k], k)$. Przeprowadzenie dowodu pozostawiamy czytelnikom jako cw. 6.2.

Mając ten rezultat, nietrudno teraz wykazać, że jest zachowane wzajemne wykluczanie. Rzeczywiście, załóżmy, że proces P_i jest w sekcji krytycznej, a P_k próbuje wejść do swojej sekcji krytycznej. Gdy proces P_k przejdzie do wykonania drugiej instrukcji **while** przy $j = i$, wówczas sprawdzi, że

- $numer[i] \neq 0$ oraz
- $(numer[i], i) < (numer[k], k)$.

Będzie zatem dopóty wykonywać tę pętlę, dopóki proces P_i nie wyjdzie ze swojej sekcji krytycznej.

Do wykazania, że są spełnione warunki postępu i ograniczonego czekania, czyli algorytm jest w pełni poprawny, wystarczy zauważyc, że procesy wchodzą do sekcji krytycznych na zasadzie „pierwszy zgłoszony – pierwszy obsłużony”.

6.3 ■ Sprzętowe środki synchronizacji

Podobnie jak przy innych aspektach oprogramowania, właściwości sprzętu mogą ułatwić programowanie i polepszyć wydajność systemu. Opiszymy teraz kilka prostych rozkazów sprzętowych, dostępnych na wielu komputerach, i pokażemy, jak można ich użyć przy rozwiązywaniu problemu sekcji krytycznej.

W środowisku jednoprocessorowym problem sekcji krytycznej daje się łatwo rozwiązać, jeśli można zakazać występowania przerwań podczas modyfikowania zmiennej dzielonej. W ten sposób uzyskuje się pewność, że dany ciąg rozkazów zostanie wykonany sekwencyjnie, bez wywłaszczenia. Ponieważ wykonanie żadnego innego rozkazu nie jest możliwe, więc nie nastąpi żadna nieoczekiwana zmiana wspólnej zmiennej.

Niestety, rozwiązanie to jest nieprzydatne w środowisku wieloprocesorowym. Wyłączanie przerwań w wieloprocesorze może być czasochłonne, gdyż wymaga przekazywania komunikatów do wszystkich procesorów. Owo przekazywanie komunikatów opóźnia wejście do każdej sekcji krytycznej, co powoduje spadek wydajności systemu. Należy też zwrócić uwagę na skutki wywoływane w zegarze systemowym, jeśli jest on aktualniany za pomocą przerwań.

Z tych powodów w wielu maszynach są specjalne rozkazy sprzętowe pozwalające w sposób niepodzielny sprawdzić i zmienić zawartość słowa albo zamienić zawartości dwu słów. Takie specjalne rozkazy mogą stosunkowo łatwo posłużyć do rozwiązywania problemu sekcji krytycznej. Zamiast omawiać rozkazy specyficzne dla poszczególnych maszyn, posłużymy się abstrakcyjnymi rozkazami obu typów.

Rozkaz *Testuj-i-Ustal* można określić tak, jak jest to pokazane na rys. 6.6. Ważną cechą tego rozkazu jest jego niepodzielne wykonanie, którego nie można przerwać. Jeśli zatem dwa rozkazy *Testuj-i-Ustal* są wykonywane jednocześnie (każdy w innym procesorze), to będą wykonane sekwencyjnie w dowolnym porządku.

W maszynie, w której istnieje rozkaz *Testuj-i-Ustal*, można uzyskać wzajemne wykluczanie, deklarując zmienną boolowską *zamek* z początkową wartością *false*. Strukturę procesu *P*, widać na rys. 6.7.

```
function Testuj-i-Ustal (var cel: boolean): boolean;
begin
    Testuj-i-Ustal := cel;
    cel := true;
end,
```

Rys. 6.6 Definicja instrukcji *Testuj-i-Ustal*

```

repeat
  while Testuj-i-Ustal(zamek) do nic;
    sekcja krytyczna
      zamek := false;
    reszta
  until false;

```

Rys. 6.7 Wzajemne wykluczanie przy użyciu rozkazu *Testuj-i-Ustal*

```

procedure Zamień(var a, b: boolean);
  var tymcz: boolean;
begin
  tymcz := a;
  a := b;
  b := tymcz;
end;

```

Rys. 6.8 Definicja rozkazu *Zamień*

Rozkaz *Zamień*, zdefiniowany na rys. 6.8, działa na zawartości dwu słów. Tak jak rozkaz *Testuj-i-Ustal*, rozkaz *Zamień* jest niepodzielny.

W maszynie z rozkazem *Zamień* wzajemne wykluczanie można zrealizować w sposób następujący. Deklaruje się globalną zmienną boolowską *zamek* i nadaje jej wartość początkową *false*. Ponadto każdy proces ma jeszcze lokalną zmienną boolowską *klucz*. Struktura procesu *P*, jest pokazana na rys. 6.9.

```

repeat
  klucz := true;
  repeat
    Zamień(zamek, klucz);
  until klucz = false;
  sekcja krytyczna
    zamek := false;
  reszta
  until false;

```

Rys. 6.9 Wzajemne wykluczanie przy użyciu rozkazu *Zamień*

```

var j: 0..n - 1;
    klucz: boolean;
repeat

    czekanie[i] := true;
    klucz := true;
    while czekanie[i] and klucz do klucz := Testuj-i-Ustal(zamek);
        czekanie[i] := false;
```

sekcja krytyczna

```

j := i + 1 mod n;
while (j ≠ i) and (not czekanie[j]) do j := j + 1 mod r;
if j = i then zamek := false
    else czekanie[j] := false;
```

reszta

until *false*;

Rys. 6.10 Wzajemne wykluczanie z ograniczonym czekaniem osiągnięte za pomocą rozkazu *Testuj-i-Ustal*

Podane algorytmy nie spełniają warunku ograniczonego czekania. Na rysunku 6.10 jest zaprezentowany algorytm używający rozkazu *Testuj-i-Ustal*, który spełnia wszystkie wymagania sekcji krytycznej. Korzysta się w nim z następujących wspólnych struktur danych:

```

var czekanie: array [0..n - 1] of boolean;
    zamek: boolean;
```

Początkowe wartości elementów tych struktur danych są równe *false*.

Aby udowodnić, że jest spełniony warunek wzajemnego wykluczania, zauważmy, że proces *P*, może wejść do sekcji krytycznej tylko wtedy, gdy *czekanie*[*i*] = *false* albo *klucz* = *false*. Zmienna *klucz* może przyjąć wartość *false* tylko wskutek wykonania instrukcji *Testuj-i-Ustal*. Pierwszy proces, który będzie wykonywać tę instrukcję, otrzyma zmienną *klucz* = *false*; wszystkie pozostałe procesy będą musiały poczekać. Zmienna *czekanie*[*i*] może stać się równa *false* tylko wtedy, gdy jakiś inny proces opuści swoją sekcję krytyczną. Wartość *false* zostanie nadana tylko jednej zmiennej *czekanie*[*i*]; spełnia to wymóg wzajemnego wykluczania.

Aby udowodnić warunek postępu, zauważmy, że argumentacja użыта przy dowodzeniu wzajemnego wykluczania jest przydatna również i tutaj, ponieważ proces wychodzący z sekcji krytycznej przypisuje wartość *false* albo zmiennej *zamek*, albo zmiennej *czekanie*[*i*]. W obu przypadkach proces czekający na wejście do sekcji krytycznej może kontynuować pracę.

Dowód ograniczonego czekania oprzemy na spostrzeżeniu, że gdy proces opuszcza sekcję krytyczną, wówczas przegląda tablicę *czekanie* w porządku cyklicznym ($i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1$). Zgodnie z tym porządkiem na wejście do sekcji krytycznej zostanie wyznaczony pierwszy proces przebywający w sekcji wejściowej (*czekanie[j] = true*). Tym samym każdy proces czekający na wejście do sekcji krytycznej osiągnie ją po co najwyżej $n - 1$ próbach. Niestety, w przypadku wieloprocesorów zrealizowanie niepodzielnych instrukcji *Testuj-i-Ustal* jest dla konstruktorów sprzętu zadaniem nietrywialnym. Implementacje tego rodzaju są omawiane w książkach dotyczących architektury komputerów.

6.4 ■ Semafora

Rozwiązania problemu sekcji krytycznej zaprezentowane w p. 6.3 nie są łatwe do uogólnienia w bardziej złożonych zagadnieniach. Do omijania tej trudności służy narzędzie synchronizacji nazwane *semforem* (ang. *semaphore*). Semafor S jest zmienną całkowitą, która – oprócz nadania wartości początkowej – jest dostępna tylko za pomocą dwu standardowych, niepodzielnych operacji: *czekaj* (ang. *wait*) i *sygnalizuj* (ang. *signal*). Pierwotnymi nazwami tych operacji były litery P (dla *wait* – od holenderskiego *proberen*: testować) i V (dla *signal* – od *verhogen*: zwiększać). Klasyczne definicje operacji *czekaj* i *sygnalizuj* są następujące:

$$\begin{aligned} \text{czekaj}(S): & \quad \text{while } S \leq 0 \text{ do nic;} \\ & \quad S := S - 1; \\ \text{sygnalizuj}(S): & \quad S := S + 1; \end{aligned}$$

Zmiany wartości całkowitych semafora muszą być wykonywane za pomocą operacji *czekaj* i *sygnalizuj* w sposób niepodzielny. Oznacza to, że gdy jeden proces modyfikuje wartość semafora, wówczas żaden inny proces nie może jednocześnie wartości tej zmieniać. Ponadto w przypadku operacji *czekaj(S)* nie może wystąpić przerwanie podczas sprawdzania wartości zmiennej całkowitej S ($S \leq 0$) i jej ewentualnego zmianiania ($S := S - 1$). W punkcie 6.4.2 pokażemy, w jaki sposób można implementować te operacje, najpierw jednak przypatrzmy się, jak można używać semaforów.

6.4.1 Sposób użycia

Semafora znajdują zastosowanie w rozwiązywaniu problemu sekcji krytycznej z udziałem n procesów. Wspólny semafor *mutex* (ang. *mutual exclusion* – wzajemne wykluczanie) jest dzielony przez n procesów. Na początku *mutex* ma wartość 1. Każdy proces P_i jest zorganizowany tak jak na rys. 6.11.

```

repeat
    czekaj(mutex);
    sekcja krytyczna
    sygnalizuj(mutex);
    reszta
until false;

```

Rys. 6.11 Implementacja wzajemnego wykluczania za pomocą semaforów

Semaforów możemy również używać do rozwiązywania różnych problemów synchronizacji. Rozważmy na przykład dwa współbieżnie wykonywane procesy: P_1 z instrukcją S_1 i P_2 z instrukcją S_2 . Założymy, że chcemy, aby instrukcja S_2 została wykonana dopiero wtedy, gdy zakończy się wykonanie instrukcji S_1 . Schemat ten można czytelnie zrealizować, pozwalając procesom P_1 i P_2 na dzielenie wspólnej zmiennej $synch$, z początkową wartością 0, i dodając do procesu P_1 instrukcję

```

 $S_1;$ 
sygnalizuj(synch);

```

oraz instrukcję

```

czekaj(synch);
 $S_2;$ 

```

do procesu P_2 . Ponieważ początkowa wartość zmiennej $synch$ wynosi 0, więc proces P_2 wykona instrukcję S_2 dopiero po wywołaniu przez proces P_1 instrukcji $sygnalizuj(synch)$, umieszczonej po instrukcji S_1 .

6.4.2 Implementacja

Podstawową wadą rozwiązań problemu wzajemnego wykluczania, zamieszczonych w p. 6.2, i podanej uprzednio definicji semafora jest to, że wszystkie one wymagają aktywnego czekania (ang. *busy waiting*). Podczas gdy jeden proces jest w swojej sekcji krytycznej, pozostałe procesy usiłujące wejść do sekcji krytycznych muszą nieustannie wykonywać instrukcję pętli w sekcji wejściowej. To ciągłe wykonywanie pętli jest w oczywisty sposób kłopotliwe w rzeczywistym systemie wieloprogramowym, w którym jeden procesor jest dzielony między wiele procesów. Aktywne czekanie marnuje cykle procesora, które mogłyby być produktywnie zużytkowane przez inne procesy. Semafor tego rodzaju bywa również nazywany *wirującą blokadą* (ang. *spinlock*), po-

nieważ oczekujący z powodu zamkniętego semafora proces „wiruje” w miejscu. Wirujące blokady są użyteczne w systemach wieloprocesorowych. Zaletą tych blokad jest to, że jeśli proces musi poczekać pod semaforem, to nie trzeba przełączać kontekstu, a przełączanie kontekstu może zabierać sporo czasu. Jeśli więc przewiduje się, że czas oczekiwania procesu będzie bardzo krótki, to wirujące blokady są pozytyczne.

Aby ominąć konieczność aktywnego czekania, można zmodyfikować semaforowe operacje *czekaj* i *sygnalizuj*. Proces, który wykonuje operację *czekaj* i zastaje niedodatnią wartość semafora, musi czekać. Jednakże zamiast czekać aktywnie, proces może się zablokować (ang. *block*). Operacja blokowania umieszcza proces w kolejce związanej z danym semaforem i powoduje przełączenie stanu procesu na czekanie. Następnie sterowanie zostaje przekazane do planisty przydziału procesora, który wybiera do wykonania inny proces.

Działanie procesu zablokowanego pod semaforem S powinno zostać wznowione wskutek wykonania operacji *sygnalizuj* przez jakiś inny proces. Wznowienie procesu następuje za pomocą operacji *budzenia* (ang. *wakeup*), która zmienia stan procesu z oczekiwania na gotowość. Proces przechodzi wówczas do kolejki procesów gotowych do wykonania. (Procesor może, lecz nie musi, zostać przełączony z aktualnie wykonywanego procesu na nowo przygotowany proces; zależy to od algorytmu planowania procesora).

Aby zrealizować semafor według tej definicji, określmy go jako rekord:

```
type semaphore = record
    wartość: integer;
    L: list of proces;
end;
```

Każdy semafor ma wartość* całkowitą i listę procesów. Kiedy proces musi czekać pod semaforem, wtedy dołącza się go do listy procesów. Operacja *sygnalizuj* usuwa jeden proces z listy czekających procesów i budzi go.

Operacje semaforowe można teraz zdefiniować tak:

```
czekaj(S):   S.wartość := S.wartość - 1;
if S.wartość < 0
then begin
    dołącz dany proces do S.L;
    blokuj;
end;
```

* Ponieważ przytaczane fragmenty programów mają charakter tylko poglądowy, w nich nie omijano liter polskiego alfabetu. – Przyp. tłum.

```

sygnalizuj(S): S.wartość := S.wartość + 1;
    if S.wartość ≤ 0
        then begin
            usuń jakiś proces P z S.L;
            obudź(P);
        end;

```

Operacja *blokuj* (ang. *block*) wstrzymuje proces, który ją wywołuje. Operacja *obudź(P)* (ang. *wakeup*) wznowia działanie zablokowanego procesu *P*. Operacje te są dostarczane przez system operacyjny jako elementarne funkcje systemowe.

Zwrócić uwagę, że chociaż w klasycznej definicji semafora z aktywnym czekaniem wartość semafora nie jest nigdy ujemna, w powyższej implementacji semafory mogą przyjmować wartości ujemne. Jeśli wartość semafora jest ujemna, to jej moduł określa liczbę procesów czekających na ten semafor. Wynika to ze zmiany porządku instrukcji zmniejszania i sprawdzania wartości zmiennej w implementacji operacji *czekaj*.

Listę czekających procesów można łatwo zbudować, dodając pole dowiązań (łącznikowe) do bloku kontrolnego każdego procesu. Każdy semafor zawiera wartość całkowitą i wskaźnik do listy bloków kontrolnych procesów. Jednym ze sposobów dodawania i usuwania procesów z tej listy, gwarantującym ograniczone czekanie, może być kolejka „pierwszy na wejściu – pierwszy na wyjściu” (FIFO), przy czym semafor zawiera wskaźniki do początku i końca tej kolejki. Jednak w ogólnym przypadku lista może być obsługiwana według dowolnej strategii tworzenia kolejek. Poprawne użycie semaforów nie zależy od poszczególnych strategii obsługi kolejek list semaforowych.

Decydującym aspektem poprawnego działania semaforów jest ich niepodzielne wykonywanie. Należy zagwarantować, że żadne dwa procesy nie wykonają operacji *czekaj* ani *sygnalizuj* na tym samym semaforze w tym samym czasie. Ten wymóg należy do problemów dotyczących sekcji krytycznej i może być osiągnięty jednym z dwóch sposobów.

W środowisku jednoprocesorowym (tj. gdy istnieje tylko jeden procesor) można po prostu zatrzymać wykrywanie przerwań podczas wykonywania operacji *czekaj* i *sygnalizuj*. Schemat ten działa w środowisku jednoprocesorowym, bo od chwili zablokowania przerwań nie mogą być przepłatane rozkazy z różnych procesów. Do chwili przywrócenia przerwań, kiedy to planista może znów przejąć nadzór nad procesorem, działa tylko proces bieżący.

W środowisku wieloprocesorowym zakaz wykrywania przerwań nie skutkuje. Rozkazy z różnych procesów (wykonywane w różnych procesorach) mogą ulegać dowolnym przeplotom. Jeśli nie ma żadnych specjalnych rozkazów sprzętowych, to można zastosować dowolne z poprawnych rozwiązań

programowych problemu sekcji krytycznej (p. 6.2). W tym przypadku sekcje krytyczne będą się składały z procedur *czekaj* i *sygnalizuj*.

Powinniśmy jednak zdawać sobie sprawę z tego, że za pomocą definicji operacji *czekaj* i *sygnalizuj* nie wyeliminowalismy całkowicie aktywnego czekania. W zamian nastąpiło usunięcie aktywnego czekania z wejścia do sekcji krytycznych w programach użytkowych. Co więcej, udało się nam je ograniczyć wyłącznie do sekcji krytycznych operacji *czekaj* i *sygnalizuj*, które są krótkie (dobrze zakodowane nie powinny zajmować więcej niż około 10 rozkazów). Zatem sekcja krytyczna jest rzadko kiedy zajmowana i aktywne czekanie zdarza się rzadko, a gdy już wystąpi, to trwa krótko. Zupełnie inna sytuacja występuje w programach użytkowych, których sekcje krytyczne mogą być długie (minuty lub nawet godziny) lub prawie zawsze zajęte. W tym przypadku aktywne czekanie byłoby skrajnie nieekonomiczne.

6.4.3 Zakleszczenia^{*} i głodzenie

Implementacja semafora z kolejką oczekujących procesów może prowadzić do sytuacji, w której kilka procesów czeka w nieskończoność na zdarzenie, które może być spowodowane tylko przez jeden z czekających procesów. Zdarzeniem, o którym mowa, jest wykonanie operacji *sygnalizuj*. Kiedy stan taki wystąpi, wtedy o procesach, które się w nim znajdują, mówi się, że ulegają *zakleszczeniu* (ang. *deadlock*).^{**}

Zilustrujemy to na przykładzie systemu złożonego z dwóch procesów, P_0 i P_1 , z których każdy ma dostęp do dwóch semaforów S i Q , ustawionych na 1.

P_0	P_1
<i>czekaj(S);</i>	<i>czekaj(Q);</i>
<i>czekaj(Q);</i>	<i>czekaj(S);</i>
<i>sygnalizuj(S);</i>	<i>sygnalizuj(Q);</i>
<i>sygnalizuj(Q);</i>	<i>sygnalizuj(S);</i>

* Z uwagi na aktualną normę terminologiczną obowiązującą w WNT w tym przekładzie termin „zakleszczenie” będzie stosowany w miejsce terminu „blokada” przyjętego w poprzednich wydaniach tej książki. – Przyp. tłum.

** Inne terminy polskie będące w użyciu na określenie angielskiego słowa *deadlock* to: blokada, zasój, impas. – Przyp. tłum.

Załóżmy, że proces P_0 wykona operację $\text{czekaj}(S)$, a potem proces P_1 wykona operację $\text{czekaj}(Q)$. Kiedy proces P_0 przejdzie do wykonania operacji $\text{czekaj}(Q)$, wtedy będzie musiał czekać, aż proces P_1 wykona operację $\text{sygnalizuj}(Q)$. Podobnie, gdy proces P_1 wykona operację $\text{czekaj}(S)$, rozpoczęcie wówczas czekanie na to, aby proces P_0 wykonał operację $\text{sygnalizuj}(S)$. Ponieważ operacje sygnalizacji nie mogą już być wykonane, więc procesy P_0 i P_1 są zakleszczone.

Mówimy, że zbiór procesów jest w stanie zakleszczenia, gdy każdy proces w tym zbiorze oczekuje na zdarzenie, które może być spowodowane tylko przez inny proces z tego zbioru. Zdarzenia, o które głównie tu chodzi, to pozyskiwanie i zwalnianie zasobów. Ale także i inne rodzaje zdarzeń mogą prowadzić do zakleszczeń, co pokazujemy w rozdz. 7. Tam opisujemy również różne sposoby postępowania w wypadkach zakleszczeń.

Innym problemem związanym z zakleszczeniami jest *blokowanie nieskończone* (ang. *indefinite blocking*), czyli *głodzenie* (ang. *starvation*) – sytuacja, w której procesy czekają w nieskończoność pod semaforem. Blokowanie nieskończone może powstać, jeśli przy dodawaniu i usuwaniu procesów z listy związanej z semaforem używa się porządku LIFO („ostatni na wejściu – pierwszy na wyjściu”).

6.4.4 Semafora binarne

Konstrukcja semafora opisana w poprzednich punktach jest powszechnie znana pod nazwą *semafora zliczającego* (ang. *counting semaphore*), ponieważ jego wartości całkowite mogą przebiegać dowolny przedział. *Semafor binarny* (ang. *binary semaphore*) to taki, którego wartość całkowita może wynosić tylko 0 lub 1. Zależnie od rodzaju sprzętu realizacja semafora binarnego może być prostsza niż semafora zliczającego. Pokażemy teraz, w jaki sposób za pomocą semaforów binarnych można utworzyć semafor zliczający.

Niech S oznacza semafor zliczający. Aby zaimplementować go przy użyciu semaforów binarnych, będziemy potrzebować następujących struktur danych:

```
var S1: semafor-binarny;
    S2: semafor-binarny;
    C: integer;
```

Początkowo $S1 = 1$, $S2 = 0$, a wartość zmiennej całkowitej C jest określana według początkowej wartości semafora zliczającego S .

Operację *czekaj* semafora zliczającego S można zrealizować następująco:

```

czekaj(S1);
 $C := C - 1;$ 
if  $C < 0$ 
    then begin
        sygnalizuj(S1);
        czekaj(S2);
    end
sygnalizuj(S1);

```

Operacja *sygnalizuj* dotycząca semafora zliczającego S może być wykonyana następująco:

```

czekaj(S1);
 $C := C + 1;$ 
if  $C \leq 0$ 
    then sygnalizuj(S2);
    else sygnalizuj(S1);

```

6.5 ■ Klasyczne problemy synchronizacji

Omówimy teraz kilka różnych problemów synchronizacji, ważnych zwłaszcza dla tego, że są one przykładami obszernych klas problemów sterowania współbieżnością. Niemal każdy nowo zaproponowany schemat synchronizacji testuje się pod względem rozwiązania tych problemów. W naszych rozwiązaniach do synchronizacji posługują się semafory.

6.5.1 Problem ograniczonego buforowania

Zagadnienie ograniczonego buforowania zostało wprowadzone w p. 6.1. Jest ono powszechnie używane do ilustrowania możliwości elementarnych operacji synchronizacji. Przedstawiamy tu ogólną strukturę tego schematu, bez zagłębiania się w jakkolwiek konkretną implementację. Zakładamy, że operujemy na puli n buforów, z których każdy mieści jedną jednostkę. Semafor *mutex* umożliwia wzajemne wykluczanie dostępu do puli buforów i ma wartość początkową równą 1. Semafor *pusty* i *pełny* zawierają odpowiednio liczbę pustych i pełnych buforów. Semafor *pusty* ma wartość początkową n ; semafor *pełny* ma wartość początkową 0.

```

repeat
  ...
    produkowanie jednostki w nastp
    ...
    czekaj(pusty);
    czekaj(mutex);
    ...
    dodanie jednostki nastp do bufora bufor
    ...
    sygnalizuj(mutex);
    sygnalizuj(pelny);
until false;

```

Rys. 6.12 Struktura procesu producenta

```

repeat
  ...
    czekaj(pelny);
    czekaj(mutex);
    ...
    wyjmowanie jednostki z bufora bufor do nastk
    ...
    sygnalizuj(mutex);
    sygnalizuj(pusty);
    ...
    konsumowanie jednostki z nastk
until false;

```

Rys. 6.13 Struktura procesu konsumenta

Kod procesu producenta jest pokazany na rys. 6.12, a kod procesu konsumenta – na rys. 6.13. Warta odnotowania jest symetria między producentem a konsumentem. Kod ten można interpretować jako produkowanie przez producenta pełnych buforów dla konsumenta albo jako wytwarzanie przez konsumenta pustych buforów dla producenta.

6.5.2 Problem czytelników i pisarzy

Obiekt danych (np. plik lub rekord) ma podlegać dzieleniu między kilka procesów współbieżnych. Niektóre z tych procesów będą tylko czytać zawartość obiektu dzielonego, natomiast inne mają go aktualniąć (tj. zarówno czytać, jak i zapisywać). Rozróżniamy oba typy procesów, nazywając te,

które są zainteresowane tylko czytaniem – czytelnikami, pozostałe zaś – pisarzami. Oczywiście jednocześnie uzyskanie dostępu przez dwu czytelników do dzielonego obiektu danych nie powoduje żadnych szkodliwych skutków. Jednak gdyby pisarz i jakiś inny proces (obojętnie który – czytelnik czy pisarz) miały jednocześnie dostęp do dzielonego obiektu, mogłyby to spowodować chaos.

Aby uniknąć trudności, należy zagwarantować wyłączność dostępu pisarzy do obiektu dzielonego. Ten problem synchronizacyjny nosi nazwę *problemu czytelników i pisarzy* (ang. *readers-writers problem*). Odkąd został sformułowany, wykorzystano go do testowania niemal wszystkich nowych elementów synchronizacji. Problem czytelników i pisarzy ma kilka odmian z zastosowaniem priorytetów. W najprostszej z nich, nazywanej pierwszym problemem czytelników i pisarzy, zaktąda się, iż żaden czytelnik nie powinien czekać, chyba że właśnie pisarz otrzymał pozwolenie na używanie obiektu dzielonego. Innymi słowy, żaden czytelnik nie powinien czekać na zakończenie pracy innych czytelników tylko z tego powodu, że czeka pisarz. W drugim problemie czytelników i pisarzy zaktąda się, że jeśli pisarz jest gotowy, to rozpoczyna wykonanie swojej pracy tak wcześnie, jak tylko to jest możliwe. Mówiąc inaczej, jeśli pisarz czeka na dostęp do obiektu, to żaden nowy czytelnik nie rozpoczęnie czytania.

Zauważmy, że rozwiązanie każdego z tych problemów może powodować głodzenie. W pierwszym przypadku może to dotyczyć pisarzy, w drugim – czytelników. Z tego powodu zaproponowano inne warianty problemu. W tym punkcie przedstawiamy rozwiązanie pierwszego problemu czytelników i pisarzy. Stosowne odesłania do rozwiązań problemu czytelników i pisarzy nie zagrożonych głodzeniem można znaleźć w uwagach bibliograficznych.

W rozwiązyaniu pierwszego problemu czytelników i pisarzy procesy czytelników dzielą poniższe zmienne:

```
var mutex, pis: semaphore;
liczba-czyt: integer;
```

Semafor *mutex* i *pis* przyjmują na początku wartość 1; *liczba-czyt* ma wartość początkową 0. Semafor *pis* jest wspólny dla procesów czytelników i pisarzy. Semafor *mutex* służy do zagwarantowania wzajemnego wykluczania przy aktualizacji zmiennej *liczba-czyt*. Zmienna *liczba-czyt* przechowuje liczbę procesów czytających obiekt. Semafor *pis* organizuje wzajemne wykluczanie pracy pisarzy. Jest on również używany przez pierwszego wchodzącego lub ostatniego opuszczającego sekcję krytyczną czytelnika. Nie używają go czytelnicy wchodzący lub wychodzący wówczas, gdy inni czytelnicy są w sekcjach krytycznych.

czekaj(pis);
 ...
 tu następuje pisanie
 ...
sygnalizuj(pis);

Rys. 6.14 Struktura procesu pisarza

czekaj(mutex);
liczba-czyt := liczba-czyt + 1;
if liczba-czyt = 1 then czekaj(pis);
sygnalizuj(mutex);
 ...
 tu następuje czytanie
 ...
czekaj(mutex);
liczba-czyt := liczba-czyt - 1;
if liczba-czyt = 0 then sygnalizuj(pis);
sygnalizuj(mutex);

Rys. 6.15 Struktura procesu czytelnika

Kod procesu pisarza jest przedstawiony na rys. 6.14; kod procesu czytelnika jest pokazany na rys. 6.15. Zauważmy, że jeśli na pisarza przebywającego w sekcji krytycznej oczekuje n czytelników, to jeden czytelnik stoi w kolejce do semafora *pis* oraz $n - 1$ czytelników ustawia się w kolejce do semafora *mutex*. Oznacza to, że gdy pisarz wykona operację *sygnalizuj(pis)*, to można wznowić działanie czekających czytelników lub pojedynczego pisarza. Wybór należy do planisty.

6.5.3 Problem obiadujących filozofów

Wyobraźmy sobie pięciu filozofów, którzy spędzają życie na myśleniu i jedzeniu. Filozofowie dzielą wspólny okrągły stół, wokół którego ustawiono pięć krzeseł – po jednym dla każdego filozofa. Na środku stołu stoi miska ryżu, a naokoło leży pięć pałeczek (rys. 6.16). Kiedy filozof myśli, wtedy nie kontaktuje się ze swoimi kolegami. Od czasu do czasu filozof zaczyna odczuwać głód. Wówczas próbuje ująć w dłonie dwie pałeczki leżące najbliżej jego miejsca przy stole (pałeczki znajdujące się pomiędzy nim a sąsiadami z lewej i prawej). Za każdym razem filozof może podnieść tylko jedną pałeczkę. Jest oczywiste, że nie będzie wyrywać pałeczek z ręki sąsiada. Kiedy



Rys. 6.16 Sytuacja obiadujących filozofów

głodny filozof zdobędzie cbie pałeczki, wtedy rozpoczyna jedzenie, nie rozmawiając się z pałeczkami ani na chwilę. Po spożyciu posiłku filozof odkłada obie pałeczki na stół i ponownie zatapia się w rozmyślaniach.

Problem pięciu obiadujących filozofów jest uznawany za klasyczne zadanie synchronizacji nie ze względu na jego praktyczne znaczenie ani też nie dlatego, iżby informatycy nie lubili filozofów, lecz z uwagi na to, że stanowi przykład szerokiej klasy problemów sterowania współbieżnością. Jest prostym odzwierciedleniem konieczności przydzielania wielu zasobów do wielu procesów w sposób grożący zakleszczeniami i głodzeniem.

W jednym z prostych rozwiązań przyjmuje się, że każda pałeczka jest semaforem. Filozof próbuje wziąć pałeczkę, wykonując operację *czekaj* odnoszącą się do danego semafora. Odkłada pałeczki za pomocą operacji *sygnalizuj* kierowanych do odpowiednich semaforów. Zatem dane dzielone przedstawiają się następująco:

```
var paleczka: array[0..4] of semaphore;
```

przy czym wszystkie elementy tablicy *paleczka* mają na początku wartość 1. Struktura *i*-tego filozofa jest przedstawiona na rys. 6.17.

Chociaż rozwiązanie to zapewnia, że żadni dwaj sąsiedzi nie będą jedli jednocześnie, musi być odrzucone, ponieważ kryje w sobie możliwość powstania zakleszczenia. Założmy, że cała piątka filozofów poczuła głód jednocześnie i każdy podniósł jedną pałeczkę, leżącą po swojej lewej stronie. Wszystkie elementy tablicy *paleczka* stają się równe 0. Usiłowania któregokolwiek z filozofów podniesienia prawej pałeczki będą skazane na nieustanne niepowodzenie.

```

repeat
    czekaj(paleczka[i]);
    czekaj(paleczka[i + 1 mod 5]);
    ...
    jedzenie
    ...
    sygnalizuj(paleczka[i]);
    sygnalizuj(paleczka[i + 1 mod 5]);
    ...
    myślenie
    ...
until false;

```

Rys. 6.17 Struktura i-tego filozofa

Poniżej wymieniamy kilka możliwych sposobów rozwiązywania problemu zakleszczenia. Rozwiążanie problemu obiadujących filozofów gwarantujące unikanie zakleszczeń przedstawimy w p. 6.7.

- Pozwolić zasiadać do stołu co najwyżej czterem filozofom naraz.
- Pozwolić filozofowi na podnoszenie pałeczek tylko wtedy, gdy są obie dostępne (tego zabiegu należy dokonać w sekcji krytycznej).
- Zastosować rozwiązanie asymetryczne, tzn. filozof o numerze nieparzystym podnosi najpierw pałeczkę po lewej stronie, a potem sięga na prawo, podczas gdy filozof parzysty rozpoczyna od pałeczki z prawej strony, a potem zwraca się ku lewej.

Dodajmy na koniec, że każde satysfakcjonujące rozwiązanie problemu obiadujących filozofów musi zapewniać, iż żaden z filozofów nie zostanie zgłodzony na śmierć. Rozwiążanie wolne od zakleszczeń nie eliminuje automatycznie możliwości blokowania nieskończonego.

6.6 ■ Regiony krytyczne

Pomimo że semafory stanowią wygodny i efektywny mechanizm synchronizowania procesów, niewłaściwe ich użycie może powodować błędy koordynacji czasowej, trudne do wykrycia z tego powodu, że zależą od pewnych szczególnych – nie zawsze występujących – ciągów wykonywanych instrukcji.

Przykład tego rodzaju błędów można było zaobserwować przy użyciu liczników w omówionym w p. 6.1 rozwiązaniu problemu producenta-konsumenta.

W tamtym przykładzie błąd synchronizacji powstawał rzadko, a zaburzenie wartości licznika nie było znaczące, odchylając się o 1. Niemniej jednak jest oczywiste, że rozwiązanie takie jest nie do przyjęcia. To z tego powodu zaczęto się najpierw interesować semaforami.

Niestety, błędy tego rodzaju mogą się zdarzać również przy zastosowaniu semaforów. Dla zilustrowania, jak może do nich dochodzić, przeanalizujemy rozwiązanie problemu sekcji krytycznej przy użyciu semaforów. Wszystkie procesy dzielą zmienną semaforową *mutex*, początkowo równą 1. Przed wejściem do sekcji krytycznej każdy proces musi wykonać operację *czekaj(mutex)*, a przy wyjściu – operację *sygnalizuj(mutex)*. Jeśli nie doszłoby do takiej sekwencji, dwa procesy mogłyby znaleźć się w sekcji krytycznej jednocześnie.

Dokonamy teraz przeglądu możliwych trudności. Dodajmy, że trudności mogą występować nawet wtedy, gdy pojedyńczy proces zachowuje się właściwie. Sytuacja taka może powstać wskutek ewidentnego błędu w programowaniu lub braku porozumienia między programistami.

- Założmy, że proces zamienia kolejność wykonywania operacji *czekaj i sygnalizuj na semaforze mutex*, doprowadzając do następującego ciągu działań:

sygnalizuj(mutex);

...
sekcja krytyczna

...
czekaj(mutex);

W tej sytuacji sekcje krytyczne może wykonywać jednocześnie wiele procesów, naruszając zasadę wzajemnego wykluczania. Wykrycie błędu może nastąpić tylko wtedy, gdy kilka procesów rzeczywiście będzie pracowało jednocześnie w sekcjach krytycznych. Weźmy pod uwagę, że sytuacja taka nie zawsze musi być odtwarzalna.

- Założmy, że w procesie operację *sygnalizuj(mutex)* zastąpiono operacją *czekaj(mutex)*; to znaczy, wykonuje się sekwencja

czekaj(mutex);

...
sekcja krytyczna

...
czekaj(mutex);

W tym przypadku powstanie zakleszczenie.

- Założmy, że proces pomija wykonanie *czekaj(mutex)* lub *sygnalizuj(mutex)* albo obu operacji. Spowoduje to naruszenie wzajemnego wykluczenia albo powstanie zakleszczenia.

Powyższe przykłady ilustrują, że niewłaściwe uzycie semaforów w rozwiązywaniu problemu sekcji krytycznej może prowadzić do powstawania różnych błędów. Podobne trudności mogą powstawać w innych modelach synchronizacji omówionych w p. 6.5.

Do radzenia sobie z naszkicowanymi tu błędami zaproponowano wiele konstrukcji językowych. W tym punkcie opiszymy jedną z podstawowych konstrukcji synchronizujących wysokiego poziomu – *region krytyczny* (ang. *critical region*), czasami nazywaną też *warunkowym regionem krytycznym* (ang. *conditional critical region*). W punkcie 6.7 opiszymy inną podstawową konstrukcję synchronizacji – *monitor*. Przedstawiając obie konstrukcje założymy, że proces składa się z zestawu danych lokalnych i sekwencyjnego programu, który na nich operuje. Do danych lokalnych można mieć dostęp tylko za pomocą programu sekwencyjnego należącego do tego procesu, do którego należą one same. Żaden proces nie może więc bezpośrednio sięgać po dane lokalne innego procesu. Niemniej jednak procesy mogą dzielić dane globalne.

W konstrukcji regionu krytycznego, służącej do organizowania synchronizacji w języku wysokiego poziomu, wymaga się, aby zmienną *v* typu *T*, która będzie używana wspólnie przez wiele procesów, deklarowano, jak niżej:

var *v*: shared *T*;

Zmienna *v* będzie dostępna tylko w obrębie instrukcji *region* o następującej postaci:

region *v* when *B* do *S*;

Konstrukcja ta oznacza, że podczas wykonywania instrukcji *S* żaden inny proces nie ma dostępu do zmiennej *v*. Dostęp do regionu krytycznego zależy od wyrażenia boolowskiego *B*. Gdy proces próbuje wejść w obszar sekcji krytycznej, wówczas oblicza się wyrażenie boolowskie *B*. Jeśli wyrażenie jest prawdziwe, to instrukcja *S* będzie wykonana. Jeśli jest fałszywe, to proces nie ubiega się o wyłączny dostęp i ulega opóźnianiu do czasu, aż wyrażenie *B* stanie się prawdziwe oraz żaden inny proces nie będzie przebywał w regionie związanym ze zmienną *v*. Jeśli więc dwie instrukcje:

region *v* when *true* do *S1*;
region *v* when *true* do *S2*;

są wykonywane współbieżnie w różnych procesach sekwencyjnych, to wynik będzie równoważny sekwencyjnemu wykonaniu „*S1 przed S2*” lub „*S2 przed S1*”.

Konstrukcja regionu krytycznego strzeże przed pewnymi prostymi błędami związanymi z rozwiązywaniem problemu sekcji krytycznej za pomocą semaforów, które mogą być popełniane przez programistę. Wypada dodać, że nie wyklucza ona z całą pewnością wszystkich błędów synchronizacji, niemniej jednak zmniejsza ich liczbę. Jeśli wystąpi błąd logiczny w programie, to odtworzenie konkretnego ciągu zdarzeń może nie być łatwe.

Konstrukcję regionu krytycznego można z powodzeniem stosować do rozwiązywania niektórych ogólnych problemów synchronizacji. Zilustrujemy to za pomocą zakodowania schematu ograniczonego buforowania. Obszar bufora i jego wskaźniki są zawarte w strukturze

var bufor: shared record

```

    pula: array [0..n - 1] of jednostka;
    licznik, we, wy: integer;
  end;

```

Proces produkujący umieszcza nową jednostkę *nastp* w buforze dzielnym, wykonując instrukcję

```

region bufor when licznik < n
  do begin
    pula[we] := nastp;
    we := we + 1 mod n;
    licznik := licznik + 1;
  end;

```

Proces konsumujący usuwa jednostkę z bufora dzielnego i zapamiętuje ją w *nastk* za pomocą instrukcji

```

region bufor when licznik > 0
  do begin
    nastk := pula[wy];
    wy := wy + 1 mod n;
    licznik := licznik - 1;
  end;

```

Pokażemy teraz, jak warunkowy region krytyczny może zostać zaimplementowany przy użyciu kompilatora. Z każdą zmienną dzieloną kojarzy się następujące zmienne:

```

var mutex, pierwsza-zwloka, druga-zwloka: semaphore;
pierwszy-licznik, drugi-licznik: integer;

```

Semafor *mutex* przyjmuje początkową wartość 1. Początkowe wartości semaforów *pierwsza-zwłoka* i *druga-zwłoka* są równe 0.

Za wyłączność dostępu do sekcji krytycznej odpowiada zmienna *mutex*. Jeśli proces nie może wejść do sekcji krytycznej z powodu fałszywości warunku boolowskiego *B*, to najpierw czeka pod semaforem *pierwsza-zwłoka*. Proces czekający pod semaforem *pierwsza-zwłoka* jest w końcu przenoszony do semaforu *druga-zwłoka*, co poprzedza udzielenie mu zezwolenia na ponowne obliczenie warunku boolowskiego *B*. Za pomocą zmiennych *pierwszy-licznik* i *drugi-licznik* pamięta się odpowiednio, ile procesów oczekuje pod semaforami *pierwsza-zwłoka* i *druga-zwłoka*.

Jeśli jakiś proces opuszcza sekcję krytyczną, to może zmienić wartość jakiegoś warunku boolowskiego *B*, wzbraniającego innemu procesowi wejścia do sekcji krytycznej. Biorąc to pod uwagę, należy przejrzeć kolejkę procesów czekających pod semaforami *pierwsza-zwłoka* i *druga-zwłoka* (w tej kolejności) i pozwolić, by każdy proces przetestował wyrażenie boolowskie. Proces testujący to wyrażenie może odkryć, że jego obecna wartość jest równa *true*. Wówczas proces wechodzi do sekcji krytycznej. W przeciwnym razie proces musi

```

czekaj(mutex);
while not B
    do begin
        pierwszy-licznik := pierwszy-licznik + 1;
        if drugi-licznik > 0
            then sygnalizuj(druga-zwłoka)
            else sygnalizuj(mutex);
        czekaj(pierwsza-zwłoka);
        pierwszy-licznik := pierwszy-licznik - 1;
        drugi-licznik := drugi-licznik + 1;
        if pierwszy-licznik > 0
            then sygnalizuj(pierwsza-zwłoka)
            else sygnalizuj(drugi-licznik);
        czekaj(druga-zwłoka);
        drugi-licznik := drugi-licznik - 1;
    end;
S;
if pierwszy-licznik > 0
    then sygnalizuj(pierwsza-zwłoka)
else if drugi-licznik > 0
    then sygnalizuj(druga-zwłoka)
    else sygnalizuj(mutex);

```

Rys. 6.18 Implementacja konstrukcji regionu warunkowego

nadal czekać pod semaforami *pierwsza-zwłoka* i *druga-zwłoka*, jak opisaliśmy poprzednio. W związku z powyższym, mając daną zmienną *x*, instrukcję

region *x* when *B* do *S*:

można zrealizować tak, jak to jest pokazane na rys. 6.18. Zauważmy, że przytoczona implementacja wymaga, aby wyrażenie *B* było obliczane dla każdego z oczekujących procesów za każdym razem, gdy jakiś proces opuszcza sekcję krytyczną. Jeśli opóźnianiu podlega wiele procesów oczekujących, aby ich odpowiednie wyrażenia boolowskie przyjęły wartość *true*, to kosztowych ponawianych obliczeń może uczynić kod algorytmu niewydajny. Istnieją różne metody optymalizacji, za pomocą których koszt ten może być zmniejszony. Odpowiednie odesłania można znaleźć w uwagach bibliograficznych zamieszczonych na końcu rozdziału.

6.7 ■ Monitory

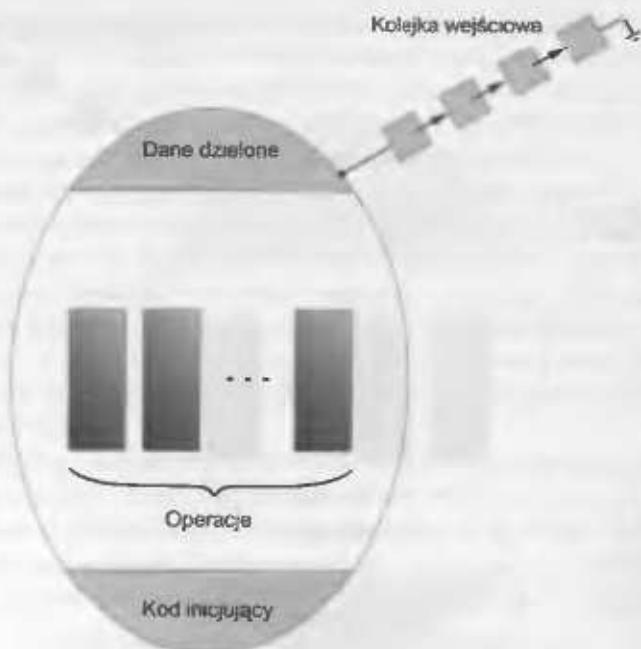
Innego rodzaju konstrukcją stosowaną w językach wysokiego poziomu do synchronizacji jest typ^{*} *monitor*. Charakterystyczną cechą monitora jest zbiór operacji zdefiniowanych przez programistę. Reprezentacja typu monitor zawiera deklaracje zmiennych, których wartości określają stan obiektu tego typu, oraz treści procedur lub funkcji realizujących działania na tym obiekcie. Składnia monitora przybiera postać

```

type nazwa-monitora = monitor
    deklaracje zmiennych
    procedure entry P1 (...);
        begin ... end;
    procedure entry P2 (...);
        begin ... end;
    .
    .
    .
    procedure entry Pn (...);
        begin ... end;
    begin
        kod inicjujący
    end.

```

^{*} Chodzi tu o typ danych w rozumieniu programowania obiektowego, czyli zarówno o statycznej strukturze, jak i zbiór określonych na niej działań. – Przyp. tłum.



Rys. 6.19 Schematyczny obraz monitora

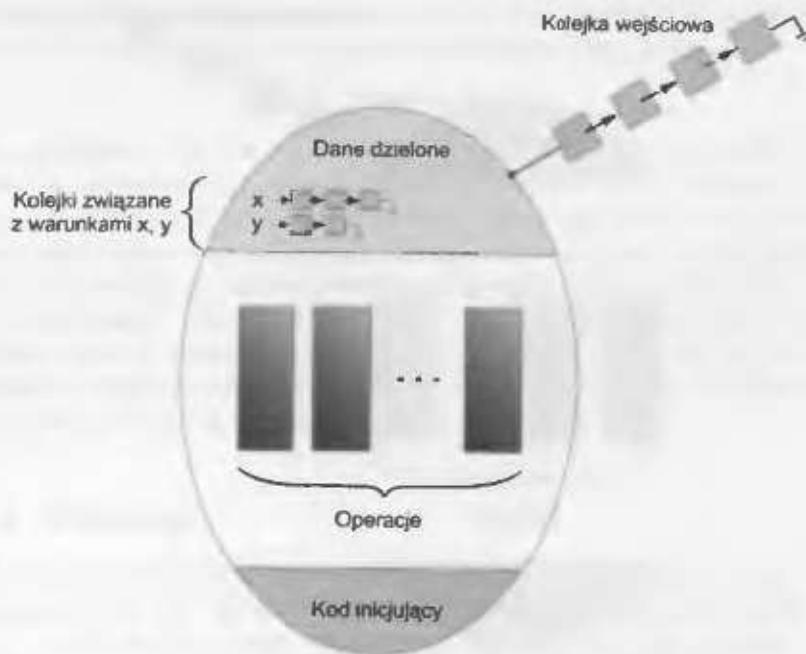
Reprezentacji typu monitor nie mogą używać bezpośrednio dowolne procesy. Procedura zdefiniowana wewnątrz monitora może korzystać tylko ze zmiennych zadeklarowanych w nim lokalnie i ze swoich parametrów formalnych. I podobnie, zmienne lokalne w monitorze są dostępne tylko za pośrednictwem tych lokalnych procedur.

Konstrukcja monitora gwarantuje, że w jego wnętrzu w danym czasie może być aktywny tylko jeden proces. W związku z tym programista nie musi kodować barier synchronizacyjnych w sposób jawny (rys. 6.19). Jednakże konstrukcja monitora w postaci zdefiniowanej do tej pory nie jest wystarczająco ogólna do modelowania pewnych schematów synchronizacji. Aby tak się stało, potrzebujemy dodatkowego mechanizmu – jest nim konstrukcja pod nazwą *warunek* (ang. *condition*). Jeśli programista chce zapisać przykrojony na miarę własnych potrzeb schemat synchronizacji, to może zdefiniować jedną lub kilka zmiennych typu *condition*:

var x, y: condition;

Jedynymi opercjami, które mogą dotyczyć zmiennej typu *condition*, są *czekaj* i *sygnalizuj*. Operacja

x.czekaj;



Rys. 6.20 Monitor ze zmiennymi warunkowymi

oznacza, że proces ją wywołujący zostaje zawieszony do czasu, aż inny proces wywoła operację

x.sygnalizuj;

Operacja *x.sygnalizuj* wznowia dokładnie jeden z zawieszonych procesów. Jeśli żaden proces nie jest zawieszony, to operacja *sygnalizuj* nie ma żadnych skutków, tzn. stan zmiennej *x* jest taki, jak gdyby operacji tej nie wykonano wcale (rys. 6.20). Warto porównać to z operacją *sygnalizuj* dotyczącą semaforów, która zawsze oddziałuje na stan semafora.

Załóżmy teraz, że w chwili, w której proces *P* wywołuje operację *x.sygnalizuj*, istnieje zawieszony proces *Q*, związany z warunkiem *x*. W tej sytuacji, jeśli zawieszonemu procesowi *Q* udzieli się zezwolenia na wznowienie działania, to sygnalizujący to proces *P* będzie musiał poczekać. W przeciwnym razie zarówno *P*, jak i *Q* stałyby się jednocześnie aktywne wewnętrz monitora. Zauważmy jednak, że teoretycznie każdy z dwóch procesów mógłby kontynuować działanie. Istnieją dwie możliwości:

1. *P* zaczeka, aż *Q* opuści monitor, albo zaczeka na inny warunek.
2. *Q* zaczeka, aż *P* opuści monitor, albo zaczeka na inny warunek.

Można znaleźć argumenty przemawiające zarówno za przypadkiem 1, jak i 2. Ponieważ proces P działa już w obrębie monitora, wybór przypadku 2 wydaje się rozsądniejszy. Jeśli jednakże pozwolimy procesowi P kontynuować pracę, to warunek „logiczny”, w oczekiwaniu na który Q pozostawał wstrzymany, może nie być już spełniony wtedy, gdy Q zostanie wznowiony.

Za wybraniem przypadku 1 opowiedział się Hoare, głównie z tego powodu, że przypadek ten można przełożyć na prostsze i bardziej eleganckie reguły przy dowodzeniu. W języku Concurrent Pascal przyjęto rozwiązanie kompromisowe. Gdy proces P wykonuje operację *sygnalizuj*, wówczas opuszcza monitor natychmiast, dzięki czemu proces Q jest niezwłocznie wznowiany. Taki schemat jest mniej uniwersalny niż zaproponowany przez Hoare'a, ponieważ proces nie może zasygnalizować więcej niż jeden raz podczas jednego wywołania procedury.

Zobrazujemy te pomysły za pomocą nie grożącego zakleszczeniami rozwiązania problemu obiadujących filozofów. Przypomnijmy, że filozofowi wolno podnosić pałeczki tylko wtedy, gdy obie są dostępne. Aby zaprogramować takie rozwiązanie, należy rozróżnić trzy stany, w których może się znajdować filozof. W tym celu wprowadzamy następującą strukturę danych:

```
var stan: array [0..4] of (myśli, głodny, je);
```

Filozof o numerze i może nadać wartość zmiennej $stan[i] = je$ tylko wtedy, gdy jego dwu sąsiadów nie pozywa się (tzn. $stan[i + 4 \bmod 5] \neq je$ and $stan[i + 1 \bmod 5] \neq je$).

Należy również zadeklarować tablicę warunków

```
var sam: array [0..4] of condition;
```

za pomocą których nękanie głodem filozofowie będą mogli opóźniać swoje zamiary, jeśli nie będzie możliwości otrzymania kompletu pałeczek.

Mogliśmy już przystąpić do opisu naszego rozwiązania problemu obiadujących filozofów. Dystrybucja pałeczek odbywa się pod nadzorem monitora *obiad-f* będącego obiektem typu *obiadujący-filozofowie*, którego definicję zawiera rys. 6.21. Przed rozpoczęciem jedzenia każdy filozof musi wywołać operację *podnieś*. Może to spowodować zawieszenie procesu filozofa. Po pomyślnym zakończeniu tej operacji filozof może jeść. Następnie filozof wywołuje operację *położ* i może rozpocząć myślenie. Tak więc i -ty filozof musi wywoływać operacje *podnieś* i *położ* w następującej kolejności:

```
obiad-f.podnieś(i);
```

↑
jedzenie

```
...  
obiad-f.położ(i);
```

```

type obiadujący-filozofowie = monitor
  var stan: array [0..4] of (myśl, głodny, je);
  var sam: array [0..4] of condition;

  procedure entry podnieś(i: 0..4);
    begin
      stan[i] := głodny;
      test(i);
      if stan[i] ≠ je then sam[i].czekaj;
    end:

  procedure entry polóż(i: 0..4);
    begin
      stan[i] := myśl;
      test(i + 4 mod 5);
      test(i + 1 mod 5);
    end:

  procedure entry test(k: 0..4);
    begin
      if stan[k + 4 mod 5] ≠ je
        and stan[k] = głodny
        and stan[k + 1 mod 5] ≠ je
        then begin
          stan[k] := je;
          sam[k].sygnalizuj;
        end:
    end:

    begin
      for i := 0 to 4
        do stan[i] := myśl;
    end.
  
```

Rys. 6.21 Zastosowanie monitora do rozwiązywania problemu obiadujących filozofów

Latwo pokazać, że to rozwiązanie zapewnia, iż żadni dwaj sąsiedzi nie będą jedli jednocześnie oraz że nie dojdzie do zakleszczenia. Zauważmy jednak, że istnieje możliwość zagłodzenia filozofa na śmierć. Nie przedstawimy rozwiązań tego problemu, pozostawiając to jako ćwiczenie dla czytelnika.

Przeanalizujemy obecnie możliwą realizację monitora, używającą semaforów. Dla każdego monitora zakłada się semafor *mutex* (z wartością początkową 1). Przed wejściem do monitora proces musi wykonać operację *czekaj(mutex)*, a przy jego opuszczaniu – operację *sygnalizuj(mutex)*.

Ponieważ proces sygnalizujący musi czekać na wyjście lub rozpoczęcie czekania przez proces wznowiony, wprowadza się dodatkowy semafor *nast*, z wartością początkową 0, za pomocą którego procesy sygnalizujące mogą same wstrzymywać swoje wykonanie. Dodatkowa zmienna całkowita *nast-licznik* posłuży do liczenia procesów wstrzymanych pod semaforem *nast*. Zatem każda zewnętrzna procedura *F* zostanie zastąpiona przez konstrukcję

czekaj(mutex);

...
treść procedury *F*

...
if *nast-licznik* > 0
 then *sygnalizuj(nast)*
 else *sygnalizuj(mutex);*

Wzajemne wykluczanie wewnętrz monitora jest zapewnione.

Opiszemy teraz, jak implementuje się zmienne w warunkach. Dla każdego warunku *x* wprowadzamy semafor *x-sem* i zmienną całkowitą *x-licznik*; nadamy im obu wartości początkowe 0. Operację *x.czekaj* można wtedy wyrazić następująco:

x-licznik := x-licznik + 1;
if *nast-licznik* > 0
 then *sygnalizuj(nast)*
 else *sygnalizuj(mutex);*
czekaj(x-sem);
x-licznik := x-licznik - 1;

Implementacja operacji *x.sygnalizuj* będzie mieć postać

if *x-licznik* > 0
 then begin
 nast-licznik := nast-licznik + 1;
 sygnalizuj(x-sem);
 czekaj(nast);
 nast-licznik := nast-licznik - 1;
 end;

Podana implementacja stosuje się do definicji monitora podanej zarówno przez Hoare'a, jak przez Brincha Hansena. Jednakże w pewnych przypadkach jest ona za mało ogólna, jak również jest możliwe znaczące polepszenie jej wydajności. Pozostawiamy ten problem czytelnikowi w ćw. 6.12.

Teraz przejdziemy do sprawy kolejności wznowiania procesów w obrębie monitora. Jeśli kilka procesów jest wstrzymanych przez warunek x i jakiś proces wykona operację $x.sygnalizuj$, to powstaje pytanie: jak rozstrzygnąć, który z wstrzymanych procesów powinien być wznowiony jako pierwszy? Najprostszym rozwiązaniem jest zastosowanie schematu FCFS, według którego jako pierwszy zostanie wznowiony proces czekający najdłużej. Jest jednak wiele sytuacji, w których ten prosty schemat planowania okazuje się nieadekwatny. W takich wypadkach można zastosować konstrukcję czekania warunkowego; ma ona postać

$x.czekaj(c);$

w której c jest wyrażeniem całkowitym obliczanym przy wykonywaniu operacji *czekaj*. Wartość c , nazywana *numerem priorytetu*, jest następnie przechowywana wraz z nazwą zawieszanego procesu. Kiedy dochodzi do wykonania $x.sygnalizuj$, jako pierwszy jest wznowiany proces z najmniejszym numerem priorytetu.

Aby zilustrować ten mechanizm, rozważmy monitor pokazany na rys. 6.22, nadzorujący przydział pojedynczego zasobu rywalizującym procesom. Każdy proces, ubiegając się o przydział zasobu, podaje maksymalny czas, przez który zamierza go używać. Monitor przydziela zasób temu spośród procesów, który zamawia go na najkrótszy czas.

Omawiany proces, ubiegający się o dostęp do zasobu, musi działać w następującym porządku:

$Z.przydziel(t);$

...
dostęp do zasobu

...
 $Z.zwolnij;$

przy czym Z jest obiektem typu *przydzielanie-zasobu*.

Niestety, pomysł z monitorem nie zapewnia, że nie dojdzie do następujących ciągów dostępów. W szczególności może się zdarzyć, że:

- proces uzyska dostęp do zasobu bez uprzedniego otrzymania pozwolenia na jego używanie;
- po uzyskaniu dostępu do zasobu proces może nigdy nie zwolnić zasobu;

```

type przydzielanie-zasobu = monitor
  var zajęty: boolean;
    x: condition;

procedure entry przydziel(czas: integer);
  begin
    if zajęty then x.czekaj(czas);
    zajęty := true;
  end;

procedure entry zwolnij;
  begin
    zajęty := false;
    x.sygnalizuj;
  end;

  begin
    zajęty := false;
  end.

```

Rys. 6.22 Monitor do przydziału pojedynczego zasobu

- proces może usiłować zwolnić zasób, którego nigdy nie zamawiał;
- proces może zamówić ten sam zasób dwukrotnie (nie zwalniając go uprzednio).

Zwrócmy uwagę na to, że te same trudności pojawiły się w konstrukcji sekcji krytycznej i są one podobne do tych, które uprzednio zachęciły nas do zbudowania regionu krytycznego i monitora. Przedtem musielismy się martwić o poprawne użycie semaforów. Teraz przedmiotem naszej troski jest poprawne użycie definiowanych przez programistę operacji wyższego poziomu, przy którym kompilator nie może już asystować.

Jednym z możliwych rozwiązań powyższego problemu jest umieszczenie operacji dostępu do zasobu wewnętrz monitora *przydzielanie-zasobu*. Jednak spowodowałoby to, że planowanie dostępu odbywałoby się według algorytmu wbudowanego w monitor, a nie przez algorytm zaprogramowany przez nas.

Aby zapewnić działanie procesów we właściwym porządku, należy prze-
glądać wszystkie programy korzystające z monitora *przydzielanie-zasobu*
i zarządzanego przez niego zasobu. W celu zapewnienia poprawności systemu
powinny być sprawdzone dwa warunki. Po pierwsze, w procesach użytkownika
musi być zawsze zachowana poprawna kolejność wywołań monitora. Po

drugie, należy mieć pewność, że żaden niezależny proces nie pominie funkcji do wzajemnego wykluczania organizowanej przez monitor i nie spróbuje uzyskać bezpośredniego dostępu do zasobu dzielnego, bez użycia protokołów dostępu. Tylko wtedy, gdy oba te warunki będą spełnione, można gwarantować, że nie wystąpią żadne błędy synchronizacji i że algorytm planowania nie ulegnie załamaniu.

Sprawdzenie takie, choć możliwe w małym, statycznym systemie, byłoby trudne do pomyślenia w systemie dużym lub dynamicznym. Do rozwiązania tego problemu kontroli dostępu są niezbędne dodatkowe mechanizmy, które omówimy w rozdz. 19.

6.8 ■ Synchronizacja w systemie Solaris 2

Aby osadzić przedstawiony materiał w realiach, powróćmy do systemu Solaris 2. Przed pojawiением się systemu Solaris 2 ważne struktury danych jego poprzednika – systemu SunOS – były strzezone za pomocą sekcji krytycznych. Sekcje krytyczne implementowano w systemie w ten sposób, że ustalano poziom przerwań jako równy lub wyższy niż poziom dowolnego przerwania, które mogłyby zmienić te same dane. Zabraniano więc występowania wszelkich przerwań, których obsługa mogłyby spowodować zmiany we wspólnych danych.

W punkcie 5.5 opisaliśmy zmiany potrzebne do umożliwienia obliczeń w czasie rzeczywistym w systemie z podziałem czasu. System Solaris 2 wyposaźono w możliwość działań w czasie rzeczywistym, działania wielowątkowe i zdolność obsługi wieloprocesorów. Pozostanie przy sekcjach krytycznych spowodowałoby w tych warunkach znaczny spadek wydajności wskutek zakorkowania jądra oczekiwaniemi na wejścia do tych sekcji. Co więcej, nie można już było implementować sekcji krytycznych za pomocą podwyższania poziomu przerwań, gdyż w systemie wieloprocesorowym przerwania mogą występować na różnych procesorach. Aby uniknąć tych problemów, w systemie Solaris 2 zastosowano do ochrony wszelkich krytycznych obiektów danych zamki adaptacyjne (ang. *adaptive mutexes*).

Zamek adaptacyjny w systemie wieloprocesorowym rozpoczyna działanie niczym standardowy semafor zrealizowany w trybie aktywnego czekania. Jeśli dane są już zamknięte, tzn. znajdują się w użyciu, to zamek adaptacyjny wykonuje jedną z dwóch czynności. Jeśli zamek jest utrzymywany przez wątek aktualnie wykonywany, to wątek ubiegający się o zamek zaczeka, ponieważ wątek utrzymujący zamek zapewne niedługo zakończy działanie. Jeżeli wątek utrzymujący zamek nie jest w stanie aktywności, to wątek pretendujący do nabycia zamka blokuje się, czyli usypia do czasu, aż zostanie obudzony

z chwilą zwolnienia zamka. Usłanie wątku pozwala uniknąć jego wirowania w przypadku, gdy zamek nie może być zwolniony dość szybko – zamek należący do uspionego wątku zapewne jest jednym z takich. W systemie jednoprocesorowym wątek utrzymujący zamek nie jest nigdy wykonywany, jeżeli zamek jest sprawdzany przez inny wątek, ponieważ w danej chwili może działać tylko jeden wątek. Dlatego w systemie jednoprocesorowym wątki, które napotkają zamknięcie, zawsze usypiają, zamiast wirować.

W sytuacjach wymagających bardziej złożonej synchronizacji system Solaris 2 stosuje zmienne warunkowe (ang. *condition variables*) oraz blokowanie zasobów w celu pisania lub czytania (ang. *readers-writers locks*). Z opisanej wyżej metody zamka adaptacyjnego korzysta się tylko do ochrony tych danych, do których dostęp odbywa się za pomocą krótkich segmentów kodu. Tak więc zamka używa się wówczas, gdy zamknięcie będzie utrzymywane przez czas wykonania co najwyżej kilkuset rozkazów. Jeżeli segment kodu jest dłuższy, to czekanie aktywne jest za mało wydajne. W przypadku dłuższych segmentów kodu stosuje się zmienne warunkowe. Jeżeli potrzebny zamek jest zablokowany, to wątek wykonuje operację *czekaj* i usypia. Kiedy zamek zostanie zwolniony przez wątek, to sygnalizuje to następnemu wątkowi z kolejki uspionych. Dodatkowe koszty usypiania wątku, jego budzenia oraz związanego z tym przełączania kontekstu są mniejsze niż marnowanie setek rozkazów podczas oczekiwania za pomocą wirującej blokady.

Blokowanie zasobów w celu pisania lub czytania stosuje się do ochrony danych, do których dostęp jest częsty, ale zazwyczaj jest to tylko czytanie. W tej sytuacji blokowanie zasobów do pisania lub czytania jest wydajniejsze niż używanie semaforów, ponieważ dane mogą być czytane współbieżnie przez wiele wątków, podczas gdy semafory zawsze szeregują dostęp do danych. Blokowanie w celu pisania lub czytania jest drogie w implementacji, toteż – jak poprzednio – stosuje się je tylko do długich sekcji kodu.

6.9 ■ Transakcje niepodzielne

Wzajemne wykluczanie się sekcji krytycznych zapewnia, że są one wykonywane w sposób niepodzielny. Oznacza to, że jeśli dwie sekcje krytyczne są wykonywane współbieżnie, to wynik jest równoważny wykonaniu ich po kolei w nieznanym porządku. Choć cecha ta jest przydatna w wielu dziedzinach zastosowań, istnieje wiele przypadków, w których chciałibyśmy mieć pewność, że sekcja krytyczna tworzy logiczną jednostkę pracy, która zostaje wykonana od początku do końca albo nie wykonuje się jej wcale. Przykładem są przelewki pieniężne, w których z jednego konta pobiera się określoną kwotę i przenosi ją na drugie konto. Mówiąc jaśniej, ze względem na spójność danych

jest istotne, aby wystąpiła zarówno wpłata, jak i potranie lub aby nie wystąpiła żadna z tych operacji.

Pozostała część tego punktu dotyczy dziedziny systemów baz danych. W *bazach danych* (ang. *databases*) uwaga skupia się na przechowywaniu i odzyskiwaniu danych oraz na ich spójności. Ostatnio obserwuje się nagły wzrost zainteresowania zastosowaniami technik baz danych w systemach operacyjnych. Systemy operacyjne można rozpatrywać w kategoriach manipulatorów danych, co umożliwia skorzystanie w nich z zaawansowanych technik i modeli wynikających z badań nad bazami danych. Na przykład wiele technik zarządzania plikami, stosowanych w systemach operacyjnych *ad hoc*, dały się uelastycznić i uogólnić, gdyby potraktowano je z uwzględnieniem bardziej formalnych metod baz danych. W punktach 6.9.2-6.9.4 opisujemy owe znamienne dla baz danych techniki oraz pokazujemy sposób ich użycia w systemach operacyjnych.

6.9.1 Model systemu

Zbiór instrukcji (operacji), które wykonują logicznie spójną funkcję, nazywa się *transakcją* (ang. *transaction*). Podstawową kwestią rozważaną w przetwarzaniu transakcji jest zachowanie ich niepodzielności pomimo ewentualnych awarii systemu komputerowego. W tym punkcie opisujemy różne mechanizmy zapewniania niepodzielności transakcji. Najpierw rozważamy środowisko, w którym w danej chwili może być wykonywana tylko jedna transakcja. Następnie bierzemy pod uwagę przypadek, w którym wiele transakcji przebiega jednocześnie.

Transakcja jest fragmentem programu, w którym dokonuje się dostępu do różnorodnych obiektów danych przechowywanych w różnych plikach na dysku. Z naszego punktu widzenia transakcja jest po prostu ciągiem operacji czytania (ang. *read*) i pisania (ang. *write*), zakończonym operacją zatwierdzenia (ang. *commit*) lub zaniechania (ang. *abort*). Operacja zatwierdzenia oznacza, że transakcja zakończyła się pomyślnie, natomiast operacja zaniechania oznacza, że wykonanie transakcji nie dobiegło do końca z powodu jakiegoś błędu logicznego. Pomyślnie zakończona transakcja nazywa się *zatwierdzoną* (ang. *committed*); w przeciwnym razie mówimy, że transakcja została *zaniechana* (ang. *aborted*). Skutków transakcji zatwierzonej nie można cofnąć przez zaniechanie transakcji.

Transakcja może również nie dobiec do końca z powodu awarii systemu. W każdym przypadku stan danych dostępnych w transakcji może nie być wówczas taki jak po niepodzielnym jej wykonaniu, ponieważ zaniechana transakcja mogła już zdążyć pozmieniać niektóre dane. Zapewnienie niepodzielności transakcji wymaga, aby transakcja zaniechana nie pozostawała

śladów w danych, które zdążyła już zmienić. Należy wobec tego dane zmienione przez zaniechaną transakcję odtworzyć do stanu, jaki miały bezpośrednio przed rozpoczęciem wykonywania transakcji. Mówimy o takiej transakcji, że została *wycofana* (ang. *rolled back*). Jednym z obowiązków systemu jest zapewnianie tej właściwości.

Aby rozstrzygnąć, w jaki sposób system powinien zapewniać niepodzielność, należy najpierw określić cechy urządzeń stosowanych do przechowywania różnych danych, z których korzystają transakcje. Biorąc pod uwagę względną szybkość środków magazynowania danych, ich pojemność oraz odporność na uszkodzenia, możemy je podzielić na kilka typów.

- **Pamięć ulotna (ang. *volatile storage*):** Informacje przechowywane w pamięci ulotnej na ogół nie są w stanie przetrwać awarii systemu. Przykładami tego rodzaju pamięci są: pamięć operacyjna (główna) oraz pamięć podrzczna. Dostęp do pamięci ulotnych jest najszybszy, zarówno z powodu swoistych ich szybkości, jak i dlatego, że dowolną jednostkę danych uzyskuje się z pamięci ulotnej w sposób bezpośredni.
- **Pamięć nieulotna (ang. *nonvolatile storage*):** Informacje przechowywane w pamięci nieulotnej zwykle potrafią przetrwać awarie systemu. Przykładami nośników takiej pamięci są dyski i taśmy magnetyczne. Dyski są bardziej niezawodne niż pamięć operacyjna, lecz mniej niezawodne niż taśmy magnetyczne. Jednak zarówno dyski, jak i taśmy są narażone na uszkodzenia, które mogą powodować utratę informacji. Obecnie wytwarzane pamięci nieulotne są wolniejsze od pamięci ulotnych o kilka rzędów wielkości, ponieważ dyski i taśmy magnetyczne są urządzeniami elektromechanicznymi, w których dostęp do danych wymaga ruchu.
- **Pamięć trwała (ang. *stable storage*):** Informacje przechowywane w pamięci trwałej nie giną nigdy („nigdy” należy tu potraktować z pewnym dystansem, gdyż teoretycznie taka stuprocentowa gwarancja nie jest możliwa). Do implementacji przybliżenia takiej pamięci należy zastosować zwielokrotnienie informacji w kilku nieulotnych pamięciach podrzcznych* (zazwyczaj dyskach), niezależnych od siebie na wypadek awarii, oraz aktualniac te informacje w sposób kontrolowany (zob. p. 13.6).

* Nie ma tu sprzeczności z uprzednim stwierdzeniem o ulotności pamięci podrzcznych, gdyż pojęcie „pamięć podrzczna” dotyczy głównie sposobu jej wykorzystania, a niezawodność zależy od technologii wykonania pamięci; poprzednia uwaga o nietrwałości pamięci podrzcznych odnosi się do pamięci elektronicznych, wymagających stałego zasilania prądem elektrycznym. – Przyp. tłum.

Dalej skoncentrujemy się wyłącznie na zapewnianiu niepodzielności transakcji w środowisku, w którym awarie powodują utratę informacji w pamięci ulotnej.

6.9.2 Odtwarzanie za pomocą rejestru

Jednym ze sposobów zapewniania niepodzielności jest zapisywanie w pamięci trwałej informacji określających wszystkie zmiany wykonywane przez transakcję w danych, do których ma ona dostęp. Najczęściej stosuje się w tym celu metodę *rejestrowania z wypredzeniem* (ang. *write-ahead logging*). System utrzymuje w pamięci trwałej strukturę danych nazywaną *rejestrem** (ang. *log*). Każdy rekord rejestru opisuje jedną operację pisania w transakcji i ma następujące pola:

- nazwa transakcji – jednoznaczna nazwa transakcji wykonującej operację pisania;
- nazwa jednostki danych – jednoznaczna nazwa zapisywanej jednostki danych;
- stara wartość – wartość jednostki danych przed zapisem;
- nowa wartość – wartość jednostki danych po zapisie.

Rejestr zawiera też inne, specjalne rekordy odnotowujące istotne zdarzenia występujące podczas przetwarzania transakcji, takie jak początek transakcji oraz jej zatwierdzenie lub zaniczanie.

Zanim rozpocznie się wykonywanie transakcji T_i , w rejestrze zapisuje się rekord $\langle T_i, \text{rozpoczęcie} \rangle$. Podczas wykonywania transakcji każda należąca do niej operacja pisz jest poprzedzana zapisaniem odpowiedniego rekordu w rejestrze. Gdy dochodzi do zatwierdzenia transakcji, wówczas w rejestrze zapisuje się rekord $\langle T_i, \text{zatwierdzenie} \rangle$.

Ponieważ informacje w rejestrze są używane do rekonstrukcji stanu danych przetwarzanych przez różne transakcje, nie możemy pozwolić na faktyczne uaktualnienie jednostki danych, zanim nie zostanie zapisany odpowiedni rekord w rejestrze przechowywanym w pamięci trwałej. Zadamy więc, aby przez wykonaniem operacji pisz(X), w trwałym rejestrze zapisano rekord dotyczący X .

Zwróciły uwagę na straty, jakie przychodzi w takim systemie ponosić na wydajności. Na każde logiczne zamówienie pisania przypadają dwie fizyczne operacje zapisu. Potrzeba też więcej pamięci – na same dane oraz na rejestr wykonanych w nich zmian. Jednak w przypadkach niezwykle ważnych da-

* W użyciu jest też nazwa dziennik. – Przyp. tłum.

nych i konieczności szybkiego odtwarzania stanu systemu po awarii cena ta jest opłacalna.

Na podstawie rejestru system potrafi poradzić sobie z każdą awarią, która nie spowodowała zaginięcia danych w pamięci nieulotnej. W algorytmie rekonstrukcji korzysta się z dwu procedur:

- **wycofaj** (ang. *undo*) – ta procedura odtwarza wszystkie dane uaktualnione przez transakcję T_i , nadając im stare wartości;
- **przywróć** (ang. *redo*) – ta procedura nadaje nowe wartości wszystkim danym uaktualnionym przez transakcję T_i .

Zbiór danych uaktualnionych przez transakcję T_i oraz odpowiadających im starych i nowych wartości można odnaleźć w rejestrze.

Aby zagwarantować poprawność działania nawet w przypadku awarii występującej podczas odtwarzania, operacje **wycofaj** i **przywróć** muszą być idempotentne (tzn. ich wielokrotne wykonywanie wywołuje takie same skutki jak wykonanie jednorazowe).

W przypadku zaniechania transakcji T_i odtworzenie stanu zmienionych przez nią danych odbywa się po prostu przez wykonanie operacji **wycofaj**(T_i). Jeśli nastąpi awaria systemu, to stan wszystkich zmienionych danych jest odtwarzany na podstawie analizowania rejestru w celu ustalenia, które transakcje należy przywrócić, a które wycofać. Owa klasyfikacja transakcji przebiega następująco:

- transakcja T_i musi być wycofana, jeżeli w rejestrze znajduje się rekord $\langle T_i, \text{rozpoczęcie} \rangle$, lecz nie ma w nim rekordu $\langle T_i, \text{zatwierdzenie} \rangle$;
- transakcja T_i musi być przywrócona, jeżeli rejestr zawiera zarówno rekord $\langle T_i, \text{rozpoczęcie} \rangle$, jak i rekord $\langle T_i, \text{zatwierdzenie} \rangle$.

6.9.3 Punkty kontrolne

Po awarii systemu należy przeanalizować rejestr transakcji, aby określić, które transakcje muszą być przywrócone, a które wycofane. Aby to rozstrzygnąć, w zasadzie należałoby przeglądać cały rejestr. Postępowanie takie ma dwie poważne wady:

1. Proces przeglądania jest czasochłonny.
2. Większość transakcji, których skutki, zgodnie z naszym algorytmem, powinny być przywrócone, spowodowało już zaktualizowanie danych.

o których na podstawie rejestru można by wnieść, że wymagają modyfikacji. Aczkolwiek powtórne wykonanie tych zmian w danych nie spowoduje żadnej szkody (dzięki idempotentności działań), jednak wydłuży czas rekonstrukcji.

Aby zmniejszyć tego rodzaju koszty, wprowadza się pojęcie *punktów kontrolnych* (ang. *checkpoints*). System podczas działania rejestruje z wyprzedzeniem operacje pisania, a ponacto co pewien czas organizuje punkty kontrolne, w których należy wykonać następujący ciąg czynności:

1. Wszystkie rekordy aktualnie pozostające w pamięci ulotnej (zazwyczaj w pamięci operacyjnej) mają być zapisane w pamięci trwałej.
2. Wszystkie zmienione dane pozostające w pamięci ulotnej mają być umieszczone w pamięci trwałej.
3. W przechowywanym w pamięci trwałej rejestrze transakcji należy zapisać rekord **<punkt kontrolny>**.

Obecność rekordu **<punkt kontrolny>** w rejestrze transakcji pozwala systemowi upłynnić procedurę rekonstrukcji. Rozważmy transakcję T_i , której zatwierdzenie nastąpiło przed punktem kontrolnym. Rekord **< T_i , zatwierdzenie>** pojawia się w rejestrze przed rekordem **<punkt kontrolny>**. Wszystkie zmiany dokonane przez transakcję T_i musiały być zapisane w pamięci trwałej albo przed punktem kontrolnym, albo w trakcie wykonywania jego operacji. Wobec tego w czasie rekonstrukcji wykonywanie operacji **przywrócić** w odniesieniu do transakcji T_i jest zbyteczne.

To spostrzeżenie pozwala ulepszyć nasz poprzedni algorytm rekonstrukcji stanu danych. Po wystąpieniu awarii procedura rekonstrukcji przegląda rejestr w celu odnalezienia ostatniej transakcji T_i , której wykonywanie rozpoczęto przed wykonaniem ostatniego punktu kontrolnego. Transakcja taka zostaje odnaleziona przez wsteczne przeszukanie rejestru w celu napotkania pierwszego rekordu **<punkt kontrolny>**, a następnie odnalezienie pierwszego rekordu **< T_i , rozpoczęcie>** występującego po nim.

Po zidentyfikowaniu transakcji T_i operacje **przywrócić i wycosaj** wystarczy wykonać tylko w odniesieniu do transakcji T_i i wszystkich transakcji, których wykonywanie rozpoczęło się po niej. Oznaczmy zbiór tych transakcji jako T . Resztę rejestru można więc pominąć. Oto wymagane operacje odtwarzania:

- Dla wszystkich transakcji T_k ze zbioru T , dla których w rejestrze występuje rekord **< T_k , zatwierdzenie>**, należy wykonać operację **przywrócić(T_k)**.

- Dla wszystkich transakcji T_k ze zbioru T , dla których w rejestrze rekord $\langle T_k, \text{zatwierdzenie} \rangle$ nie występuje, należy wykonać operację $\text{wycofaj}(T_k)$.

6.9.4 Współbieżne transakcje niepodzielne

Ponieważ każda transakcja jest niepodzielna, współbieżne wykonanie transakcji musi być równoważne wykonaniu ich jedna po drugiej w pewnym dowolnym porządku. Ta cecha, zwana *szeregowalnością* (ang. *serializability*), może być urzeczywistniona przez zwykłe wykonywanie każdej transakcji w sekcji krytycznej. Należy przez to rozumieć, że wszystkie transakcje korzystają ze wspólnego semafora *mutex*, który na początku jest równy 1. Transakcja rozpoczętająca działanie wykonuje na samym początku operację *czekaj(mutex)*. Gdy transakcja zostanie zatwierdzona, lub po jej zaniechaniu, wykonuje się operację *sygnalizuj(mutex)*.

Ten schemat, choć zapewnia niepodzielność wszystkich współbieżnie wykonywanych transakcji, jest nazbyt ograniczający. Jak się przekonamy, istnieje wiele sytuacji, w których można by dopuścić do zachodzenia w tym samym czasie działań wykonywanych w transakcjach przy zachowaniu szeregowalności. Istnieje kilka różnych algorytmów *sterowania współbieżnością* zapewniających szeregowalność. Opisujemy je poniżej.

6.9.4.1 Szeregowalność

Weźmy pod uwagę system z dwoma obiektami danych A i B , które są czytane i zapisywane przez transakcje T_0 i T_1 . Założymy, że transakcje te są wykonywane niepodzielnie w ten sposób, że transakcja T_0 poprzedza transakcję T_1 . Ten ciąg wykonan, nazywany *planem* (ang. *schedule*), jest przedstawiony na rys. 6.23. W pokazanym na rys. 6.23 planie 1 instrukcje występują w porządku chronologicznym od góry do dołu, przy czym instrukcje transakcji T_0 są umieszczone w lewej kolumnie, a instrukcje transakcji T_1 – w prawej.

T_0	T_1
czytaj(A)	
pisz(A)	
czytaj(B)	
pisz(B)	
	czytaj(A)
	pisz(A)
	czytaj(B)
	pisz(B)

Rys. 6.23 Plan 1, czyli plan szeregowy, w którym transakcja T_0 poprzedza transakcję T_1 .

Plan, w którym każda transakcja jest wykonywana niepodzielnie, nazywa się *planem szeregowym* (ang. *serial schedule*). Każdy plan szeregowy składa się z ciągu instrukcji należących do różnych transakcji, przy czym instrukcje jednej transakcji występują w nim obok siebie. Wobec tego dla zbioru n transakcji istnieje $n!$ różnych, poprawnych planów szeregowych. Poprawność każdego planu szeregowego wynika stąd, że jest on równoważny niepodzielnemu wykonaniu poszczególnych składowych transakcji w pewnym dowolnym porządku.

Jeżeli zezwolimy na przeplatanie się instrukcji wchodzących w skład dwóch transakcji, to wynikowy plan przestaje być szeregowy. Nieszeregowy plan niekoniecznie musi pociągać za sobą błąd w ostatecznym wyniku wykonania (tzn. że jest on równoważny planowi szeregowemu). Aby się przekonać, czy tak jest, musimy zdefiniować pojęcie *operacji konfliktowych* (ang. *conflicting operations*). Rozważmy plan P , w którym występują po sobie dwie operacje O_1 i O_2 , należące odpowiednio do transakcji T_0 i T_1 . Powiemy, że operacje O_1 i O_2 pozostają w konflikcie, jeżeli mają dostęp do tych samych obiektów danych i co najmniej jedna z nich jest operacją pisania. Aby zilustrować pojęcie operacji konfliktowych, przeanalizujemy nieszeregowy plan 2 z rys. 6.24. Operacja *pisz(A)* należąca do transakcji T_0 pozostaje w konflikcie z operacją *czytaj(A)* należącą do transakcji T_1 . Natomiast operacja *pisz(A)* z transakcji T_1 nie pozostaje w konflikcie z operacją *czytaj(B)* z transakcji T_0 , gdyż każda z tych operacji ma dostęp do innych danych.

Niech O_1 i O_2 będą sąsiednimi operacjami w planie P . Jeśli O_1 i O_2 są operacjami w różnych transakcjach oraz O_1 i O_2 nie pozostają w konflikcie, to wolno nam zmienić porządek operacji O_1 i O_2 , co daje nowy plan P' . Można oczekwać, że plan P będzie równoważny planowi P' , jako że wszystkie operacje występują w obu planach w tym samym porządku z wyjątkiem operacji O_1 i O_2 , których kolejność jest bez znaczenia.

T_0	T_1
czytaj(A)	
pisz(A)	
	czytaj(A)
	pisz(A)
czytaj(B)	
pisz(B)	
	czytaj(B)
	pisz(B)

Rys. 6.24 Plan 2: szeregowalne zaplanowanie współbieżne

Aby zilustrować pomysł z zamianą, rozważmy raz jeszcze plan 2 z rys. 6.24. Ponieważ operacja `pisz(A)` należąca do transakcji T_1 nie pozostaje w konflikcie z operacją `czytaj(B)` z transakcji T_0 , możemy zamienić te operacje w celu utworzenia równoważnego planu. Niezależnie od początkowego stanu systemu oba plany doprowadzą system do jednakowych stanów. Kontynuując zamianianie bezkonfliktowych operacji, wykorujemy, co następuje:

- zamianę operacji `czytaj(B)` z transakcji T_0 z operacją `czytaj(A)` z transakcji T_1 ;
- zamianę operacji `pisz(B)` z transakcji T_0 z operacją `pisz(A)` z transakcji T_1 ;
- zamianę operacji `pisz(B)` z transakcji T_0 z operacją `czytaj(A)` z transakcji T_1 .

Końcowym efektem tych zamian jest plan 1 z rys. 6.23, czyli plan szeregowy. Wykaźaliśmy więc, że plan 2 jest równoważny planowi szeregowemu, a to oznacza, że niezależnie od początkowego stanu systemu plan 2 doprowadzi do takiego samego stanu końcowego jak któryś z planów szeregowych.

Jeśli dany plan P można przekształcić w plan szeregowy P' za pomocą ciągu zamian bezkonfliktowych operacji, to mówimy, że plan P jest *szeregowalny z uwzględnieniem konfliktów* (ang. *conflict serializable*). Zatem plan 2 jest szeregowalny z uwzględnieniem konfliktów, gdyż można go przekształcić w plan szeregowy 1.

6.9.4.2 Protokół blokowania

Jeden ze sposobów zapewnienia szeregowalności polega na skojarzeniu z każdym obiektem danych zamka* i wymaganiu, aby w każdej transakcji przestrzegano *protokołu blokowania* (ang. *locking protocol*), rządzącego blokowaniem i zwalnianiem zasobów. Obiekt danych może być zablokowany na różne sposoby. W tym punkcie ograniczymy naszą uwagę do dwu trybów blokowania:

- **Tryb wspólny** (ang. *shared mode*): Jeśli transakcja T_i zajmuje obiekt danych w trybie wspólnym (oznaczanym przez S), to może ona czytać ten obiekt, lecz nie może go zapisywać.
- **Tryb wyłączny** (ang. *exclusive mode*): Jeśli transakcja T_i zajmuje obiekt danych w trybie wyłącznym (oznaczanym przez X), to wolno jej zarówno czytać, jak i zapisywać ten obiekt.

Wymaga się, aby każda transakcja zamawiała zamek blokujący obiekt danych Q w odpowiednim trybie, stosownie do rodzaju operacji, które będzie ona na obiekcie Q wykonywać.

* Przez zamk (ang. *lock*) rozumie się tu logiczny atrybut zasobu wskazujący, czy zasób jest zablokowany (zajęty) czy wolny do wynajęcia. Według obowiązujących obecnie norm angielski termin *lock (locking)* tłumaczy się jako „blokada” („blokowanie”), choć w literaturze jest również stosowany w tym kontekście termin „zajmowanie” (zasobu). – Przyp. tłum.

Aby uzyskać dostęp do obiektu Q , transakcja T , musi go wpierw zablokować w odpowiednim trybie. Jeśli obiekt Q nie jest w danej chwili zablokowany, to zezwolenie na jego zablokowanie zostaje udzielone i transakcja T , może z obiektem skorzystać. Jeśli jednak obiekt Q jest w danej chwili zablokowany przez inną transakcję, to transakcja T , musi zaczekać. Uściślając, założymy, że transakcja T , zamawia obiekt Q na wyłączny użytkę. W tym przypadku transakcja T , musi czekać dopóty, dopóki obiekt Q nie zostanie zwolniony. Jeżeli transakcja T , zamawia dostęp do obiektu Q w trybie wspólnym, to musi czekać wówczas, gdy obiekt Q jest zablokowany w trybie wyłącznym. W przeciwnym razie wolno jej zablokować obiekt Q i korzystać z niego. Zauważmy, że schemat ten jest bardzo podobny do algorytmu czytelników i pisarzy, omówionego w p. 6.5.2.

Transakcja może zwolnić obiekt danych uprzednio przez nią zablokowany. Niemniej jednak musi ona blokować obiekt danych dopóty, dopóki ma do niego dostęp. Ponadto natychmiastowe zwalnianie obiektu danych przez transakcję, tuż po wykonaniu przez nią ostatniego dostępu do obiektu, nie zawsze jest pożądane, gdyż może to naruszyć szeregowalność jej operacji.

Jednym z protokołów zapewniających szeregowalność jest *protokół blokowania dwufazowego*^{*} (ang. *two-phase locking protocol*). Protokół ten wymaga, aby każda transakcja blokowała zasób i zwalniała go w dwu fazach. Są to:

- **Faza wzrostu (ang. growing phase):** Transakcja może zablokować zasób, lecz nie wolno jej zwolnić żadnego z już zablokowanych zasobów.
- **Faza zmniejszania (ang. shrinking phase):** Transakcja może zwolnić zasób, lecz nie wolno jej już blokować nowych zasobów.

Początkowo transakcja jest w fazie wzrostu, zajmując potrzebne zasoby. Z chwilą zwolnienia zasobu transakcja wchodzi w fazę zmniejszania, w której nie wolno jej już blokować nowych zasobów.

Protokół blokowania dwufazowego zapewnia szeregowalność pod względem konfliktów (ćw. 6.21). Nie chroni jednak przed zakleszczeniem. Zaznaczmy, że dla zbioru transakcji mogą istnieć uszeregowania pod względem konfliktów, których nie da się otrzymać za pomocą protokołu blokowania dwufazowego. Jednak w celu uzyskania lepszej wydajności od tej, którą osiąga się przy zastosowaniu protokołu blokowania dwufazowego, należy dysponować dodatkowymi informacjami o transakcjach albo narzucić zbiorowi danych pewną strukturę lub uporządkowanie.

* Inna nazwa tego protokołu to *protokół zajmowania dwufazowego*. – Przyp. tłum.

6.9.4.3 Protokoły korzystające ze znaczników czasu

W opisanych wyżej protokołach blokowania porządek w każdej parze konfliktowych transakcji jest określony podczas ich wykonywania, kiedy dochodzi do zablokowania pierwszego zasobu, o który obie ubiegają się w niezgodnych trybach. Inna metoda określania kolejności transakcji polega na ich uporządkowaniu z góry. W jednej z najpopularniejszych metod tego rodzaju stosuje się schemat *uporządkowania według znaczników czasu* (ang. *time-stamp ordering*).

Z każdą transakcją T_i w systemie kojarzymy niepowtarzalny i stały znacznik czasu: określmy go symbolem $ZC(T_i)$. Transakcja otrzymuje znacznik czasu od systemu, zanim rozpoczęci się jej wykonywanie. Jeśli transakcji T_i przypisano znacznik czasu $ZC(T_i)$ i w późniejszym czasie w systemie pojawi się nowa transakcja T_j , to $ZC(T_i) < ZC(T_j)$. Schemat ten można zrealizować dwoma prostymi sposobami:

- Należy użyć w roli znaczników czasu wartości zegara systemowego. Tak więc znacznik czasu transakcji będzie równy wartości zegara systemowego w chwili pojawienia się transakcji w systemie. Metody tej nie można zastosować do transakcji występujących w oddzielnym systemach ani w przypadku procesorów nie korzystających ze wspólnego zegara.
- Należy użyć w roli znaczników czasu wskaźników licznika logicznego. W tym przypadku znacznik czasu transakcji jest równy wartości tego licznika w chwili nadania transakcji do systemu. Po przypisaniu nowego znacznika czasu licznik ulega zwiększeniu.

Znaczniki czasu transakcji określają porządek szeregowania transakcji. Jeśli więc $ZC(T_i) < ZC(T_j)$, to system musi zapewnić, że wytworzony plan jest równoważny planowi szeregowemu, w którym transakcja T_i występuje przed transakcją T_j .

W celu implementacji tego schematu z każdym obiektem danych Q kojarzymy dwa znaczniki czasu:

- **znacznik-czasu-P(Q)**, określający największy znacznik czasu transakcji, która pomyślnie wykonała operację $\text{pisz}(Q)$;
- **znacznik-czasu-C(Q)**, określający największy znacznik czasu transakcji, która pomyślnie wykonała operację $\text{czytaj}(Q)$.

Owe znaczniki czasu są aktualniane podczas każdego wykonywania instrukcji $\text{czytaj}(Q)$ lub $\text{pisz}(Q)$.

Protokół uporządkowania według znaczników czasu zapewnia, że wszelkie konfliktowe operacje **czytaj** i **pisz** będą wykonywane w porządku znaczników czasu. Protokół ten działa następująco:

- Założmy, że transakcja T_1 wydaje polecenie **czytaj**(Q).
 - Jeśli $ZC(T_1) < \text{znacznik-czasu-P}(Q)$, to z tego wynika, że transakcja T_1 chce czytać wartość Q , która już została zmieniona. Dlatego operacja **czytaj** zostaje odrzucona, a transakcja T_1 – wycofana.
 - Jeśli $ZC(T_1) \geq \text{znacznik-czasu-P}(Q)$, to wykonuje się operację **czytaj** oraz określa znacznik-czasu-C(Q) jako maksimum z wartości znacznik-czasu-C(Q) i $ZC(T_1)$.
- Założmy, że transakcja T_1 wydaje polecenie **pisz**(Q).
 - Jeśli $ZC(T_1) < \text{znacznik-czasu-C}(Q)$, to z tego wynika, że wytwarzana przez transakcję T_1 wartość Q była potrzebna wcześniej i przyjmuje się, że transakcja T_1 nigdy nie wytworzyła tej wartości. Wobec tego rezygnuje się z operacji **pisz** i wycofuje transakcję T_1 .
 - Jeśli $ZC(T_1) < \text{znacznik-czasu-P}(Q)$, to z tego wynika, że transakcja T_1 usiłuje zapisać przestarzałą wartość Q . Dlatego odrzuca się tę operację **pisz** i wycofuje transakcję T_1 .
 - W przeciwnym razie wykonuje się operację **pisz**.

Transakcja T_1 , która w wyniku żądania wykonania operacji **czytaj** lub **pisz** została przez algorytm sterowania współbieżnością wycofana, otrzymuje nowy znacznik czasu i jest wznowiana.

W celu zilustrowania tego protokołu rozpatrzmy plan 3 z rys. 6.25, zawierający transakcje T_2 i T_3 . Zakładamy, że transakcji przypisuje się znacznik czasu tuż przed jej pierwszą instrukcją, zatem w planie 3 $ZC(T_2) < ZC(T_3)$ i plan ten jest możliwy do przyjęcia w protokole znaczników czasu.

T_2	T_3
czytaj (B)	czytaj (B)
czytaj (A)	pisz (B) czytaj (A) pisz (A)

Rys. 6.25 Plan 3: zaplanowanie możliwe przy użyciu protokołu ze znacznikami czasu

Zauważmy, że taki ciąg wykonań może być też wytworzony przez protokół blokowania dwufazowego. Istnieją jednakże plany akceptowalne w protokole blokowania dwufazowego, które są nie do przyjęcia w protokole z użyciem znaczników czasu – i na odwrót (ćw. 6.22).

Protokół porządkowania według znaczników czasu zapewnia szeregowalność z uwzględnieniem konfliktów. Wynika to z tego, że konfliktowe operacje są wykonywane w porządku znaczników czasu. Protokół jest także wolny od zakleszczeń, ponieważ żadna z transakcji nigdy na nic nie czeka.

6.10 ■ Podsumowanie

Mając dany zbiór współpracujących procesów sekwencyjnych, dzielących wspólne dane, należy zadbać o zachowanie wzajemnego wykluczania niektórych z ich działań. Jedno z rozwiązań polega na zapewnieniu, że sekcja krytyczna kodu jest używana w danym czasie tylko przez jeden proces lub wątek. Istnieje wiele algorytmów rozwiązywania problemu sekcji krytycznej przy założeniu, że przeplatanie dostępu może dotyczyć tylko pamięci.

Główną wadą rozwiązań kodowanych przez użytkownika jest wymagane przez nie aktywne czekanie. Trudność tę można pokonać dzięki użyciu semaforów. Semafora mogą być stosowane do rozwiązywania różnorodnych problemów synchronizacji, przy czym mają one efektywne implementacje, zwłaszcza wówczas, gdy sprzęt dostarcza niepodzielnych operacji.

Różnorodne problemy synchronizacji (takie jak problem ograniczonego buforowania, problem czytelników i pisarzy oraz problem obiadujących filozofów) są ważne głównie z tego powodu, że są przykładami szerokich klas problemów sterowania współbieżnością. Problemy te są użyteczne przy testowaniu prawie wszystkich nowych schematów synchronizacji.

System operacyjny musi dostarczać środków do ochrony przed błędami synchronizacji. W tym celu zaproponowano kilka konstrukcji w językach wysokiego poziomu. Bezpieczną i wydajną implementację wzajemnego wykluczania i rozwiązań dowolnych problemów synchronizacji można osiągnąć za pomocą schematu regionów krytycznych. Monitory dostarczają mechanizmów synchronizacji umożliwiających dzielenie abstrakcyjnych typów danych. Za pomocą zmieronych warunkowych można blokować wykonywanie procedury monitora dopóty, dopóki nie zostanie zasygnalizowana konieczność jej dalszego działania.

System Solaris 2 jest przykładem nowoczesnego systemu operacyjnego, w którym wprowadzono różnego rodzaju zabezpieczenia wspomagające wielozadaniowość i wielowątkowość (włącznie z wątkami czasu rzeczywistego) oraz wieloprocesorowość. Do ochrony danych, z których korzystają krótkie

segmenty kodu, stosuje się w systemie Solaris 2 zareki adaptacyjne. W przypadku dostępu do danych w dłuższych sekcjach kodu używa się zmiennych warunkowych i blokowania na czas czytania lub pisania.

Transakcja jest elementem programu, który należy wykonać w sposób niepodzielny, tzn. jej operacje są wykonywane aż do ostatniej albo nie wykonuje się żadnej z nich. Aby zapewnić niepodzielność transakcji pomimo awarii systemu, można zastosować *rejestr zapisów wyprzedzających*. Wszystkie aktualnienia są zapamiętywane w rejestrze, a on sam jest przechowywany w pamięci trwałe. W przypadku awarii systemu zapamiętane w rejestrze informacje wykorzystuje się do rekonstrukcji stanu zaktualizowanych obiektów danych, w czym są pomocne operacje *wycosaj* i *przywróć*. Aby zmniejszyć koszt przeszukiwania rejestru po awarii systemu, można posłużyć się *punktami kontrolnymi*.

Jeśli operacje kilku transakcji przeplatają się ze sobą, to efekt takiego wykonania może nie być równoważny niepodzielnemu wykonaniu transakcji. W celu zagwarantowania poprawnego wykonania należy zastosować schemat sterowania współbieżnością, aby zapewnić szeregowalność*. Istnieją rozmaite schematy sterowania współbieżnością zapewniające szeregowalność przez odwlekanie operacji lub przez zaniechanie transakcji, w której polecono wykonać operację**. Do najpopularniejszych należą algorytmy blokowania dwufazowego i porządkowania według znaczników czasu.

■ Ćwiczenia

- 6.1 Co oznacza termin *czekanie aktywne*? Jaki inne rodzaje czekania występują w systemie operacyjnym? Czy można uniknąć czekania aktywnego w zupełności? Odpowiedź uzasadnij.
- 6.2 Udowodnij, że algorytm piekarni (p. 6.2.2) ma następującą właściwość: jeśli proces P_i jest w sekcji krytycznej oraz proces P_k ($k \neq i$) ma już wybrany $numer[k] \neq 0$, to $(numer[i], i) < (numer[k], k)$.
- 6.3 Pierwsze poprawne, programowe rozwiązanie problemu sekcji krytycznej jest autorstwa Dekkera. Dwa procesy, P_0 i P_1 , dzielą następujące zmienne:

```
var znacznik: array [0..1] of boolean; (* początkowo fałszywe *)
    numer: 0..1;
```

Struktura procesu P_i ($i = 0$ lub 1), przy czym P_j ($j = 1$ lub 0) oznacza drugi proces, jest taka jak na rys. 6.26.

* Uporządkowane wykonanie działań transakcji. – Przyp. tłum.

** W niewłaściwym porządku. – Przyp. tłum.

repeat

```

znacznik[i] := true;
while znacznik[j]
    do if numer = j
        then begin
            znacznik[i] := false;
            while numer = j do nic;
            znacznik[i] := true;
        end:

```

sekcja krytyczna

```

numer := j;
znacznik[i] := false;

```

reszta

until false;

Rys. 6.26 Struktura procesu P , w algorytmie Dekkera

Udowodnij, że algorytm ten spełnia wszystkie trzy wymagania dotyczące do sekcji krytycznej.

- 6.4 Pierwsze poprawne, programowe rozwiązanie problemu sekcji krytycznej dla n procesów, w którym dolna granica oczekiwania wynosi $n - 1$ prób, zaprezentowali Eisenberg i McGuire. Proces dzieli następujące zmienne:

```

var znacznik: array [0..n - 1] of (bezczyyny, gotowy, w-sekcji);
numer: 0..n - 1;

```

Wszystkie elementy tablicy *znacznik* mają początkowo stan *bezczyyny*, początkowa wartość zmiennej *numer* nie ma znaczenia (równa się którejś z liczb od 0 do $n - 1$). Struktura procesu P , jest przedstawiona na rys. 6.27.

Udowodnij, że algorytm ten spełnia wszystkie trzy wymagania dotyczące do sekcji krytycznej.

- 6.5 W punkcie 6.3 wspominaliśmy, że zakazywanie przerwań może mieć wpływ na zegar systemowy. Wyjaśnij, z jakiego powodu mogłyby tak być, i określ, w jaki sposób możliwe owe skutki minimalizować.
- 6.6 Pokaż, że jeśli operacje *czekaj* i *sygnalizuj* nie są wykonywane niepozzielnie, to wzajemne wykluczanie może zostać naruszone.

var $j : 0..n$:

repeat

repeat

znacznik[i] := gotowy;

j := numer;

while $j \neq i$

do if *znacznik[j] ≠ bezczynny*

then *j := numer*

else *j := j + 1 mod n;*

znacznik[i] := w-sekcji;

j := 0;

while ($j < n$) **and** ($j = i$ **or** *znacznik[j] ≠ w-sekcji*) **do**

j := j + 1;

until ($j \geq n$) **and** (*numer = i* **or** *znacznik[numer] = bezczynny*);

numer := i;

sekcja krytyczna

j := numer + 1 mod n;

while (*znacznik[j] = bezczynny*) **do** *j := j + 1 mod n;*

numer := j;

znacznik[i] := bezczynny;

reszta

until *false;*

Rys. 6.27 Struktura procesu P , w algorytmie Eisenberga-McGuire'a

- 6.7 Problem śpiącego golibrody (ang. *The Sleeping-Barber Problem*). W zakładzie fryzjerskim jest poczekalnia z n krzesłami i salon z jednym tylko fotelem. Jeśli brakuje klientów, to fryzjer po prostu zasypia. Jeżeli w poczekalni nie ma wolnych miejsc, to nowy klient opuszcza zakład. Gdy fryzjer jest zajęty, ale są wolne miejsca, wówczas klient siada na jednym z nich. Jeśli fryzjer śpi, to klient go budzi. Napisz program koordynujący zachowanie fryzjera i klientów.
- 6.8 Problem palaczy tytoniu (ang. *The Cigarette-Smokers Problem*). Rozważmy system z trzema procesami *palacz* i jednym procesem *dostawcy*. Każdy palacz nieustannie skręca i wypala papierosa. Aby jednak skręcić i zapalić papierosa, palacz potrzebuje trzech rzeczy: tytoniu, papieru i zapalnika. Jeden z procesów-palaczy ma papier, drugi ma tytoni, a trzeci – zapalki. Dostawca ma nieograniczone ilości wszystkiego. Do-

- stawca kładzie dwa różne składniki na stole. Palacz, który ma pozostały składnik, robi wówczas skręta, wypala go i sygnalizuje to dostawcy. Wówczas dostawca znów kładzie dwa składniki na stole i cykl się powtarza. Napisz program koordynujący działania dostawcy i palaczy.
- 6.9 Wykaż, że monitory, warunkowe regiony krytyczne oraz semafory są równoważne w sensie możliwości rozwiązywania za ich pomocą tych samych typów problemów synchronizacji.
- 6.10 Napisz monitor obsługujący ograniczone buforowanie, w którym bufory (porcje) są umieszczone w obrębie samego monitora.
- 6.11 Ścisłe wzajemne wykluczanie wewnętrz monitora powoduje, że monitor ograniczonego buforowania z ćw. 6.10 jest przydatny głównie do małych porcji.
- (a) Wyjaśnij, dlaczego to stwierdzenie jest prawdziwe?
 - (b) Zaprojektuj nowy schemat, odpowiedni dla większych porcji.
- 6.12 Założmy, że instrukcja *sygnalizuj* może wystąpić tylko jako ostatnia instrukcja w procedurze monitora. W jaki sposób może to wpłynąć na uproszczenie implementacji omówionej w p. 6.7?
- 6.13 Rozważmy system złożony z procesów P_1, P_2, \dots, P_n , z których każdy ma jednoznaczny numer priorytetu. Napisz monitor, który przydziela tym procesom trzy identyczne drukarki wierszowe, używając numerów priorytetów przy decydowaniu o kolejności przydziału.
- 6.14 Plik ma być dzielony przez różne procesy, z których każdy ma jednoznaczny numer. Do tego pliku może mieć dostęp jednocześnie kilka procesów przy następującym ograniczeniu: suma wszystkich jednoznacznych numerów przypisanych procesom, które mają w danej chwili dostęp do pliku, musi być mniejsza od n . Napisz monitor koordynujący dostęp do tego pliku.
- 6.15 Założmy, że monitorowe operacje *czekaj* i *sygnalizuj* zastępujemy pojedynczą konstrukcją *oczekuj(B)*, w której B jest ogólnym wyrażeniem boolowskim; operacja ta powoduje, że wykonujący ją proces rozpoczęta czekanie na spełnienie warunku B .
- (a) Napisz monitor implementujący za pomocą tej konstrukcji problem czytelników i pisarzy.
 - (b) Wyjaśnij, dlaczego w ogólnym przypadku konstrukcji tej nie da się wydajnie implementować.

- (c) Jakie ograniczenia należałoby nałożyć na instrukcję *oczekuj*, aby można ją było zrealizować efektywnie? (Wskazówka: należy ograniczyć ogólność wyrażenia *B*; Kessels [209]).
- 6.16 Napisz monitor implementujący *budzik*, który pozwala wywołującemu go programowi opóźniać się o określony liczbę jednostek czasu (tyknięcie). Możesz założyć istnienie rzeczywistego zegara sprzętowego, który w regularnych odstępach czasu wywołuje procedurę *tik-tak* w Twoim monitorze.
- 6.17 Dlaczego w systemie Solaris 2 wprowadzono wiele mechanizmów blokowania? W jakich sytuacjach stosuje się czekanie aktywne, blokowanie za pomocą semaforów, zmienne warunkowe oraz blokowanie na czas czytania lub pisania? W jakim celu system korzysta z każdego z tych mechanizmów?
- 6.18 Omów różnice między trzema rodzajami pamięci: ulotną, nieulotną i trwałą, biorąc pod uwagę ich koszt.
- 6.19 Wyjaśnij mechanizm punktu kontrolnego. Jak często należy wykonywać działania w punktach kontrolnych? W jaki sposób częstotliwość występowania punktów kontrolnych wpływa na:
- wydajność systemu podczas jego bezawaryjnej pracy;
 - czas zużywany na odtwarzanie stanu systemu po awarii;
 - czas zużywany na rekonstrukcję systemu po awarii dysku.
- 6.20 Wyjaśnij zasadę niepodzielności transakcji.
- 6.21 Wykaż, że protokół blokowania dwufazowego zapewnia szeregowalność pod względem konfliktów.
- 6.22 Wykaż, że istnieją plany możliwe do zrealizowania z użyciem protokołu blokowania dwufazowego, lecz nie nadające się do protokołu ze znacznikami czasu – i na odwrót.

Uwagi bibliograficzne

Algorytmy 1 i 2 wzajemnego wykluczania dla dwu procesów zostały przedstawione po raz pierwszy w klasycznym artykule Dijkstry [110]. Algorytm Dekkera (ćw. 6.3) – pierwsze poprawne rozwiązanie programowe problemu wzajemnego wykluczania dla dwu procesów – został opracowany przez holenderskiego matematyka T. Dekkera. Algorytm ten był także omawiany

przez Dijkstrę [110]. Prostsze rozwiązanie problemu wzajemnego wykluczania dla dwóch procesów zostało od tamtej pory zaprezentowane przez Petersona [329] (algorytm 3).

Pierwsze rozwiązanie problemu wzajemnego wykluczania dla n procesów zaprezentował Dijkstra [111]. Rozwiązywanie to nie miało jednak górnego ograniczenia czasu oczekiwania procesu na zezwolenie wejścia do sekcji krytycznej. Knuth w artykule [217] przedstawił pierwszy algorytm z takim ograniczeniem; wynosiło ono 2^n cykli. Ulepszenie algorytmu Knutha dokonane przez deBruijna [95] zmniejszyło czas czekania do n^2 cykli, po czym Eisenbergowi i McGuire'owi [124] (ćw. 6.4) udało się skrócić ten czas do dolnej granicy $n - 1$ cykli. Algorytm piekarni (algorytm 5) jest dziełem Lamporta [225]; również ten algorytm wymaga $n - 1$ cykli, ale jest łatwiejszy do zaprogramowania i zrozumienia. Algorytm rozwiązania sprzętowego, które spełnia wymóg ograniczonego czekania opracował Burns [60].

Ogólne omówienie dotyczące problemu wzajemnego wykluczania oferuje Lamport w artykułach [231, 232]. Zbiór algorytmów realizujących wzajemne wykluczanie podał Raynal [344].

Pomysł semafora pochodzi od Dijkstry [110]. Patil w pracy [324] analizuje pytanie, czy semafory mogą posłużyć do rozwiązania wszystkich możliwych problemów synchronizacji. Parnas [322] omawia pewne niedociągnięcia w argumentacji Patila. Kosaraju podążył śladem Patila, aby w pracy [222] przedstawić problem, którego nie można rozwiązać za pomocą operacji *czekaj* i *sygnalizuj*. Lipton w dysertacji [258] przeanalizował ograniczenia różnych elementarnych konstrukcji synchronizujących.

Opisane przez nas klasyczne problemy koordynacji procesów są wzorcowymi przykładami obszernej klasy problemów sterowania współbieżnością. Problemy ograniczonego buforowania, obiadujących filozofów oraz śpiącego golibrody (ćw. 6.7) zaproponował Dijkstra [110], [113]. Problem palaczy tytoniu (ćw. 6.8) sformułował Patil [324]. Problem czytelników i pisarzy zaproponowali Courtois i in. [87]. Lamport w artykule [227] omówił zagadnienie współbieżnego czytania i pisania, a w artykule [226] zajął się problemem synchronizacji procesów niezależnych.

Koncepcję regionu krytycznego zaproponowali Hoare [172] i Brinch Hansen [55]. Pomysł monitora jest autorstwa Brincha Hansena [56]. Pełny opis monitora podał Hoare [173]. Kessels [209] zaproponował rozszerzenie monitora o możliwość automatycznego sygnalizowania. Ben-Ari zaważył ogólne omówienie zagadnień programowania współbieżnego w książce [29].

Szczegóły dotyczące mechanizmów blokowania zasobów w systemie Solaris 2 przedstawiono w pracach: Khanna i in. [210], Powell i in. [334] oraz Ekholt i in. [128] (przede wszystkim). Zauważmy, że mechanizmy blokowania stosowane przez jądro są również dostępne dla wątków poziomu użytko-

wego, tak więc zarówno w jądrze, jak i poza nim można blokować zasoby w ten sam sposób.

Metodę rejestracji z wyprzedzeniem zastosowano po raz pierwszy w Systemie R (Gray i in. [156]). Ideę szeregowalności sformułowali Eswaran i in. [127] w związku z prowadzonymi przez siebie pracami nad sterowaniem współbieżnością w Systemie R. Protokół blokowania dwufazowego pochodzi od Eswarana i in. [127]. Schemat sterowania współbieżnością za pomocą znaczników czasu zaproponował Reed [347]. Wybór różnych algorytmów sterowania współbieżnością, korzystających ze znaczników czasu, zaprocentowali Bernstein i Goodman [31].

Rozdział 7

ZAKLESZCZENIA*

W środowisku wieloprogramowym kilka procesów może rywalizować o skończoną liczbę zasobów. Proces zamawia zasoby i jeśli nie są one dostępne w danym czasie, wchodzi w stan oczekiwania. Może się zdarzyć, że oczekujące procesy nigdy już nie zmieniają swego stanu, ponieważ zamawiane przez nie zasoby są przetrzymywane przez inne procesy. Sytuację taką nazywa się *zakleszczeniem** (ang. *deadlock*). Zagadnienie to omawialiśmy już pokrótko w rozdz. 6 przy okazji semaforów.

Być może najlepszą ilustrację zakleszczenia można zaczerpnąć z prawa ustanowionego przez legislaturę stanu Kansas w początku tego wieku. Glosiło ono w szczególności, że: „Jeśli dwa pociągi zbliżają się do siebie po kryzysujących się torach, to każdy z nich powinien się zatrzymać i nie ruszać ponownie do czasu, aż drugi z nich odjedzie”.

W tym rozdziale opisujemy metody, za pomocą których system operacyjny może pokonywać problem zakleszczenia. Zwrócić jednak uwagę, że większość współczesnych systemów operacyjnych nie ma środków zapobiegania zakleszczeniom. We właściwości takie zostaną one prawdopodobnie wyposażone z upływem czasu, gdy problemy zakleszczeń zaczną pojawiać się częściej. Przyczyną takiego przebiegu zdarzeń będzie kilka tendencji, w tym rosnące liczby procesów, znaczny przyrost ilości zasobów (w tym procesorów) i większy nacisk kładziony na długowieczne serwery plików i baz danych anizeli na systemy wsadowe.

* Przypominamy, że z uwagi na aktualną normę terminologiczną w tym przekładzie termin „zakleszczenie” będzie stosowany w miejscu terminu „blokada” przyjętego w poprzednich wydaniach tej książki. – Przyp. tłum.

** Inaczej: *blokada, zastój, impas.* – Przyp. tłum.

7.1 ■ Model systemu

System składa się ze skończonej liczby zasobów, które są rozdzielane między pewną liczbę rywalizujących ze sobą procesów. Zasoby dzieli się na typy, z których każdy zawiera pewną liczbę identycznych egzemplarzy. Przykładami typów zasobów są: obszar pamięci, cykle procesora, pliki i urządzenia wejścia-wyjścia (takie jak drukarki czy przewijaki taśmy). Jeśli system ma dwa procesory, to zasób typu *procesor* ma dwa egzemplarze. Podobnie, zasób typu *drukarka* może mieć pięć egzemplarzy.

Jeśli proces zamawia egzemplarz zasobu jakiegoś typu, to jego zapotrzebowanie spełni przydział *dowolnego* egzemplarza danego typu. Gdyby tak się nie stało, oznaczałoby to, że egzemplarze nie są identyczne i klasy zasobów nie zostały właściwie zdefiniowane. Na przykład system może mieć dwie drukarki. Obie mogą być zaliczone do tej samej klasy zasobów, jeśli nikomu nie zależy na tym, która drukarka będzie wyprowadzała czyste wyniki. Jednakże, gdy jedna drukarka znajduje się na ósmym piętrze, a druga w piwnicy, wówczas ludzie z ósmego piętra mogą nie uważać obu drukarek za równoważne; w tej sytuacji byłoby pożąданie zaliczyć je do oddzielnych klas.

Proces powinien zamówić zasób przed jego użyciem i zwolnić go po wykorzystaniu. Proces może żądać tylu zasobów, ile ich potrzebuje do wykonania zadania. Oczywiście liczba zamawianych zasobów nie może przekroczyć ogólnej liczby zasobów dostępnych w systemie. Innymi słowy, proces nie może zamówić trzech drukarek, jeśli system ma tylko dwie drukarki.

W normalnych warunkach działania proces może użyć zasobu tylko w następującym porządku:

- Zamówienie:** Jeśli zamówienie nie może być spełnione natychmiast (np. zasób jest użytkowany przez inny proces), to proces zamawiający musi czekać do chwili otrzymania zasobu.
- Użycie:** Proces może korzystać z zasobu (np. jeśli zasobem jest drukarka, to proces może drukować na tej drukarce).
- Zwolnienie:** Proces oddaje zasób.

Zasoby można zamawiać i zwalniać za pomocą funkcji systemowych, jak było wyjaśnione w rozdz. 3. Przykładami są funkcje systemowe przydzielania i zwalniania urządzenia, otwarcia i zamknięcia pliku oraz przydziału i zwolnienia pamięci. Zamawianie i zwalnianie innych zasobów może być dokonywane za pomocą semaforowych operacji *czekaj* i *sygnalizuj*. Dzięki temu przed każdym użyciem zasobu system operacyjny sprawdza, czy proces zamówił dany zasób i czy mu go przydzielono. W tablicy systemowej zapisuje się, czy zasób jest wolny czy też przydzielony, a jeśli jest przydzielony, to do

którego procesu. Jeśli proces zamawia zasób, który jest aktualnie przydzielony innemu procesowi, to zamawiający proces dołącza się do kolejki procesów oczekujących na dany zasób.

Zbiór procesów jest w stanie zakleszczenia, gdy każdy proces z tego zbioru czeka na zdarzenie, które może być spowodowane tylko przez inny proces z tego samego zbioru. Zdarzenia, z którymi najczęściej mamy tu do czynienia, dotyczą przydziału i zwalniania zasobów. Może tu chodzić o zasoby fizyczne (np. drukarki, przewijaki taśmy, miejsce w pamięci i cykle procesora) lub logiczne (takie jak pliki, semafory i monitory). Niemniej jednak także i inne typy zdarzeń mogą prowadzić do zakleszczeń (np. schemat komunikacji międzyprocesowej omawiany w rozdz. 4).

Aby zobrazować zakleszczenie, rozważmy system z trzema przewijakami taśmy. Założymy, że istnieją trzy procesy, z których każdy przechodzi przez jeden z tych przewijaków. Jeśli każdy proces zamówi teraz dodatkowy przewijak taśmy, to omawiane trzy procesy znajdą się w stanie zakleszczenia. Każdy będzie czekał na zdarzenie „zwolniono przewijak taśmy”, które mogłyby być spowodowane tylko przez któryś z pozostałych czekających procesów. Ten przykład ilustruje zakleszczenie dotyczące procesów rywalizujących o zasób tego samego typu.

Przyczyną zakleszczeń mogą być także zasoby różnych typów. Rozważmy na przykład system z jedną drukarką i jednym przewijakiem taśmy. Założymy, że proces P_1 ma przydzielony przewijak taśmy, a proces P_2 przechodzi przez drukarkę. Jeśli P_1 zamówi teraz drukarkę, a P_2 zamówi przewijak taśmy, to wystąpi zakleszczenie.

7.2 ■ Charakterystyka zakleszczenia

Jest zrozumiałe, że zakleszczenia nie są zjawiskiem pożądanym. Będące w stanie zakleszczenia procesy nigdy nie skończą swoich działań, wiążąc zasoby systemowe i uniemożliwiając rozpoczęcie wykonywania innych zadań. Zanim omówimy różne metody postępowania z problemem zakleszczeń, opiszemy cechy, którymi zakleszczenia się charakteryzują.

7.2.1 Warunki konieczne

Do zakleszczeń może dochodzić wtedy, kiedy w systemie zachodzą jednocześnie cztery warunki:

1. **Wzajemne wykluczanie:** Przynajmniej jeden zasób musi być niepodzielny; to znaczy, że zasobu tego może używać w danym czasie tylko

jeden proces. Jeśli inny proces zamawia dany zasób, to musi być opóźniany do czasu, aż zasób zostanie zwolniony.

2. **Przetrzymywanie i oczekiwanie:** Musi istnieć proces, któremu przydzielono co najmniej jeden zasób i który oczekuje na przydział dodatkowego zasobu, przetrzymywanejgo właśnie przez inny proces.
3. **Brak wywłaszczeń:** Zasoby nie podlegają wywłaszczeniu, co oznacza, że zasób może zostać zwolniony tylko z inicjatywy przetrzymującego go procesu, po zakończeniu pracy tego procesu.
4. **Czekanie cykliczne:** Musi istnieć zbiór $\{P_0, P_1, \dots, P_n\}$ czekających procesów, takich że P_0 czeka na zasób przetrzymywany przez proces P_1 , P_1 czeka na zasób przetrzymywany przez proces P_2, \dots, P_{n-1} , P_{n-1} czeka na zasób przetrzymywany przez proces P_n , a P_n czeka na zasób przetrzymywany przez proces P_0 .

Podkreślimy, że do wystąpienia zakleszczenia jest niezbędne, aby były spełnione wszystkie cztery warunki. Warunek czekania cyklicznego implikuje warunek przetrzymywania i oczekiwania, więc wymienione cztery warunki nie są zupełnie niezależne. Wykażemy jednak w p. 7.4, że rozważanie każdego z nich z osobna jest wygodne.

7.2.2 Graf przydziału zasobów

Zakleszczenia można dokładniej opisać za pomocą grafu skierowanego, zwanego tu *grafem przydziału zasobów systemu* (ang. *system resource-allocation graph*). Graf ten składa się ze zbioru wierzchołków W i zbioru krawędzi K . Zbiór wierzchołków W jest podzielony na dwa rodzaje węzłów: $P = \{P_1, P_2, \dots, P_n\}$, czyli zbiór wszystkich procesów systemu, oraz $Z = \{Z_1, Z_2, \dots, Z_m\}$ – oznaczający zbiór wszystkich typów zasobów systemowych.

Krawędź skierowaną od procesu P_i do zasobu typu Z_j zapisuje się w postaci $P_i \rightarrow Z_j$; oznacza ona, że proces P_i zamówił egzemplarz zasobu typu Z_j i obecnie czeka na ten zasób. Krawędź skierowana od zasobu typu Z_j do procesu P_i , co zapisujemy w postaci $Z_j \rightarrow P_i$, oznacza, że egzemplarz zasobu typu Z_j został przydzielony do procesu P_i . Krawędź skierowaną $P_i \rightarrow Z_j$ nazywa się *krawędzią zamówienia* (ang. *request edge*), a krawędź skierowaną $Z_j \rightarrow P_i$ nazywa się *krawędzią przydziału* (ang. *assignment edge*).

Każdy proces P_i jest przedstawiany na rysunku w postaci kółka, a każdy typ zasobu Z_j – w postaci prostokąta. Ponieważ zasób typu Z_j może mieć więcej niż jeden egzemplarz, każdy egzemplarz zasobu będziemy oznaczać kropką umieszoną w prostokącie. Zauważmy, że krawędź zamówienia sięga

tylko do brzegu prostokąta Z_j , podczas gdy krawędź przydziału musi także wskazywać na jedną z kropek w prostokącie.

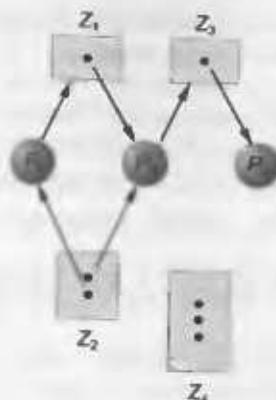
Kiedy proces P_i zamawia egzemplarz zasobu typu Z_j , wtedy w grafie przydziału zasobów umieszcza się krawędź zamówienia. Gdy zamówienie to zostaje spełnione, wówczas krawędź zamówienia natychmiast zamienia się na krawędź przydziału. Gdy zasób przestaje być procesowi potrzebny, wówczas proces zwalnia go, wskutek czego krawędź przydziału zostanie usunięta.

Graf przydziału zasobów na rys. 7.1 odzwierciedla następującą sytuację:

- Zbiory P , Z i K :
 - $P = \{P_1, P_2, P_3\}$;
 - $Z = \{Z_1, Z_2, Z_3, Z_4\}$;
 - $K = \{P_1 \rightarrow Z_1, P_2 \rightarrow Z_1, Z_1 \rightarrow P_2, Z_2 \rightarrow P_2, Z_2 \rightarrow P_1, Z_3 \rightarrow P_3\}$.
- Egzemplarze zasobów:
 - jeden egzemplarz zasobu typu Z_1 ;
 - dwa egzemplarze zasobu typu Z_2 ;
 - jeden egzemplarz zasobu typu Z_3 ;
 - trzy egzemplarze zasobu typu Z_4 .
- Stany procesów:
 - proces P_1 utrzymuje egzemplarz zasobu typu Z_2 i oczekuje na egzemplarz zasobu typu Z_1 ;
 - proces P_2 ma po egzemplarzu Z_1 i Z_2 i czeka na egzemplarz zasobu typu Z_3 ;
 - proces P_3 ma egzemplarz zasobu Z_3 .

Mając definicję grafu przydziału zasobów, można wykazać, że jeśli graf nie zawiera cykli, to w systemie nie ma zakleszczonych procesów. Jeśli natomiast graf zawiera cykl, to może dojść do zakleszczenia.

Jeżeli zasób każdego typu ma tylko jeden egzemplarz, to cykl implikuje, że doszło do zakleszczenia. Jeśli cykl zawiera zbiór tylko takich typów zasobów, z których każdy ma tylko pojedynczy egzemplarz, to zakleszczenie jest faktem. Każdy proces występujący w cyklu tkwi w miejscu. W tym wypadku istnienie cyklu w grafie jest warunkiem koniecznym i wystarczającym do wystąpienia zakleszczenia.

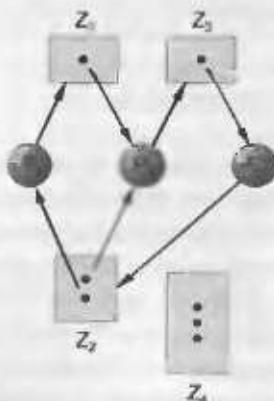


Rys. 7.1 Graf przydziału zasobów

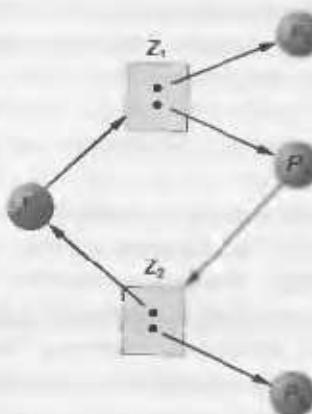
Jeśli istnieje po kilka egzemplarzy zasobu każdego typu, to obecność cyklu nie oznacza jeszcze, że wystąpiło zakleszczenie. W tym wypadku cykl w grafie jest warunkiem koniecznym, lecz nie wystarczającym do istnienia zakleszczenia.

Aby zilustrować tę koncepcję, powróćmy do grafu przydziału zasobów przedstawionego na rys. 7.1. Założmy, że proces P_3 zamawia egzemplarz zasobu typu Z_1 . Ponieważ w danej chwili nie ma żadnego wolnego egzemplarza tego zasobu, do grafu dodaje się krawędź zamówienia $P_3 \rightarrow Z_1$ (rys. 7.2). Od tej chwili w systemie istnieją dwa cykle minimalne:

$$\begin{aligned} P_1 &\rightarrow Z_1 \rightarrow P_2 \rightarrow Z_3 \rightarrow P_3 \rightarrow Z_1 \rightarrow P_1 \\ P_2 &\rightarrow Z_3 \rightarrow P_3 \rightarrow Z_1 \rightarrow P_1 \end{aligned}$$



Rys. 7.2 Graf przydziału zasobów z zakleszczeniem



Rys. 7.3 Graf przydziału zasobów z cyklem, lecz bez zakleszczenia

Procesy P_1 , P_2 i P_3 są zakleszczone. Proces P_2 czeka na zasób Z_1 , przetrzymywany przez proces P_3 . Z kolei proces P_3 czeka na to, aby albo proces P_1 , albo proces P_2 zwolnił zasób Z_2 . Wreszcie, proces P_1 czeka na zwolnienie przez proces P_2 zasobu Z_1 .

Rozważmy teraz rys. 7.3. W tym przykładzie również występuje cykl:

$$P_1 \rightarrow Z_1 \rightarrow P_3 \rightarrow Z_2 \rightarrow P_1$$

Niemniej jednak nie ma tu zakleszczenia. Zauważmy, że proces P_4 może zwolnić egzemplarz zasobu typu Z_2 . Egzemplarz ów może zostać wtedy przydzielony procesowi P_3 , co spowoduje rozerwanie cyklu.

Podsumowując, jeśli graf przydziału zasobów nie ma cyklu, to system *nie* jest w stanie zakleszczenia. W przeciwnym razie – w przypadku istnienia cyklu – system może być w stanie zakleszczenia lub nie. Jest to ważne spostrzeżenie, mające znaczenie w postępowaniu z zakleszczeniami.

7.3 ■ Metody postępowania z zakleszczeniami

Zasadniczo są trzy różne metody radzenia sobie z problemem zakleszczeń:

- Można stosować protokół gwarantujący, że system *nigdy* nie wejdzie w stan zakleszczenia.
- Pozwala się systemowi na zakleszczenia, po czym podejmuje się działania zmierzające do ich usunięcia.

- Można też zlekceważyć ten problem zupełnie, uważając, że zakleszczenia nigdy nie pojawią się w systemie. Takie rozwiązanie stosuje się w większości systemów, m.in. w systemie UNIX.

Omówimy krótko każdą z metod, a następnie w p. 7.4-7.8 przedstawimy szczegółowe algorytmy.

Aby zapewnić, że zakleszczenia nigdy się nie pojawią, system może stosować metody zapobiegawcze lub schemat unikania zakleszczeń. Przez *zapobieganie zakleszczeniom* (ang. *deadlock prevention*) rozumie się zbiór metod zapewniających, że co najmniej jeden z warunków koniecznych do wystąpienia zakleszczenia (p. 7.2.1) nie będzie spełniony. Metody te zapobiegają zakleszczeniom przez nakładanie ograniczeń na sposób zamawiania zasobów; omawiamy je w p. 7.4.

Unikanie zakleszczeń (ang. *deadlock avoidance*) prowadzi z kolei do ządania, aby system operacyjny zarazem dysponował dodatkowymi informacjami o zasobach, które proces będzie zamawiał i używał podczas swojego działania. Mając te dodatkowe informacje, możemy dla każdego zamówienia rozstrzygać, czy proces powinien zaczekać, czy nie. Każde zamówienie wymaga, aby system, podejmując decyzję o tym, czy można je zrealizować, czy też należy je odłożyć, wziął pod uwagę aktualnie dostępne zasoby, zasoby w danej chwili przydzielone do procesów oraz przyszłe zamówienia i zwolnienia zasobów w odniesieniu do każdego procesu. Schematy takie omawiamy w p. 7.5.

Jeżeli system nie korzysta ani z algorytmu zapobiegania zakleszczeniom, ani z algorytmu ich unikania, to zakleszczenie może się pojawić. W takich warunkach system powinien umożliwiać wykonanie algorytmu, który sprawdzi jego stan, aby określić, czy doszło do zakleszczenia, oraz algorytmu likwidowania zakleszczenia (jeśli zakleszczenie naprawdę wystąpiło). Zagadnienia te omawiamy w p. 7.6 i 7.7.

Jeśli system nie zapewnia, że zakleszczenia nigdy nie wystąpią ani też nie zawiera mechanizmu ich wykrywania i usuwania, to może dojść do sytuacji, w której system jest w stanie zakleszczenia, lecz nie ma sposobu zorientowania się w tym, co zaszło. W tym przypadku zakleszczenie pogorszy działanie systemu z powodu utrzymywania zasobów przez procesy nie będące w stanie działać, jak również dlatego, że coraz więcej procesów wykonujących zamówienia zasobów będzie ulegać zablokowaniu. Po pewnym czasie dojdzie do zatrzymania pracy systemu, który trzeba będzie ręcznie uruchomić od początku.

Choć ta metoda nie wygląda zachęcająco, jeśli idzie o postępowanie z problemem zakleszczenia, jest stosowana w niektórych systemach operacyjnych. W wielu systemach do zakleszczeń dochodzi rzadko (dajmy na to – raz do roku), więc opłaca się postępować w ten sposób, zamiast korzystać

z kosztownego zapobiegania zakleszczeniom lub ich unikania, albo też stosowania metod wykrywania i usuwania zakleszczeń, wymagających ciągłych zabiegów. Istnieją też sytuacje, w których system znajduje się w stanie zamrożenia, nie będąc zakleszczonym. Przykładem takiej sytuacji może być proces czasu rzeczywistego wykonywany z najwyższym priorytetem (lub dowolny proces działający w systemie bez wywłaszczeń), który nie oddaje sterowania systemowi operacyjnemu.

7.4 ■ Zapobieganie zakleszczeniom

W punkcie 7.2.1 zauważaliśmy, że aby doszło do zakleszczenia, musi być spełniony każdy z czterech niezbędnych warunków. Zapewniając, iż przynajmniej jeden z tych warunków nie będzie mógł być spełniony, możemy zapobiegać występowaniu zakleszczeń. Przeanalizujmy to podejście, przyglądając się każdemu z czterech niezbędnych warunków z osobna.

7.4.1 Wzajemne wykluczanie

Warunek wzajemnego wykluczania musi być spełniony w odniesieniu do zasobów niepodzielnych. Na przykład drukarka nie może być jednocześnie użytkowana przez kilka procesów. Skądinąd zasoby dzielone nie wymagają dostępu na zasadzie wzajemnego wykluczania, więc nie mogą powodować zakleszczeń. Pliki udostępniane tylko do czytania są dobrym przykładem zasobu dzielenego. Jeśli kilka procesów chce w tym samym czasie otworzyć plik dostępny tylko do czytania, to zezwoli się im na jednoczesne korzystanie z takiego pliku. Na zasób dzielony proces nigdy nie musi czekać. W ogólnym przypadku nie jest jednak możliwe zapobieganie zakleszczeniom przez załączenie warunku wzajemnego wykluczania – niektóre zasoby są z natury niepodzielne.

7.4.2 Przetrzymywanie i oczekiwanie

Aby zapewnić, że warunek przetrzymywania i oczekiwania nigdy nie wystąpi w systemie, musimy zagwarantować, że jeżeli kiedykolwiek proces zamawia zasób, to nie powinien mieć żadnych innych zasobów. Jeden z protokołów, który można tu zastosować, wymaga, aby każdy proces zamawiał i dostawał wszystkie swoje zasoby, zanim rozpoczęcie działania. Wymóg ten można spełnić przez dopilnowanie, by wywołania funkcji systemowych dotyczących zamówień zasobów potrzebnych procesowi poprzedzały wywołania wszystkich innych funkcji systemowych.

Alternatywny protokół pozwala procesowi na zamawianie zasobów tylko wtedy, gdy proces nie ma żadnych zasobów. Proces może zamówić jakieś zasoby i korzystać z nich. Jednakże zanim proces zamówi jakiekolwiek dodatkowe zasoby, musi wpierw oddać wszystkie zasoby, które ma w danej chwili przydzielone.

W celu zilustrowania różnicy między tymi protokołami rozważmy proces, który kopiuje dane z taśmy na przewijaku do pliku dyskowego, sortuje plik dyskowy, a następnie drukuje wyniki na drukarce. Jeżeli wszystkie zasoby mają być zamówione na początku procesu, to proces musi na wstępie zamówić przewijak taśmy, plik dyskowy i drukarkę. Potem będzie przetrzymywał drukarkę przez cały czas swojego działania, mimo że będzie jej potrzebował dopiero na końcu.

Druga metoda pozwala procesowi na zamówienie na początku tylko przewijaka taśmy i pliku dyskowego. Proces przekopiuje dane z taśmy na dysk, po czym zwolni zarówno przewijak taśmy, jak i plik dyskowy. W następnej kolejności proces musi ponownie zamówić plik dyskowy oraz drukarkę. Po skopiowaniu pliku z dysku na drukarkę proces zwalnia oba te zasoby i kończy działanie.

Opisane protokoły mają dwie podstawowe wady. Po pierwsze, wykorzystanie zasobów może być bardzo małe, ponieważ z wielu przydzielonych zasobów nie będzie nikt korzystać przez długie okresy czasu. Na przykład w opisanym przypadku na zwolnienie przewijaka taśmy i pliku dyskowego, a następnie na ponowne zamówienie pliku dyskowego i drukarki można sobie pozwolić tylko po uzyskaniu pewności, że dane pozostały w pliku dyskowym. Jeśli nie możemy być tego pewni, to musimy wynajęć wszystkie zasoby na samym początku – niezależnie od użytego protokołu.

Po drugie, może dochodzić do głodzenia; proces potrzebujący kilku popularnych zasobów może być odwlekany w nieskończoność z powodu ciągłego przydzielania choćby jednego z nich innym procesom.

7.4.3 Brak wywłaszczeń

Trzeci warunek konieczny stanowi, że przydzielone zasoby nie ulegają wywłaszczeniu. Aby zapewnić, że warunek ten nie wystąpi, można posłużyć się następującym protokołem. Gdy proces mający jakieś zasoby zgłasza zapotrzebowanie na inny zasób, który nie może być mu natychmiast przydzielony (tzn. proces musiałby czekać), wówczas proces ten traci wszystkie dotychczałe zasoby. Oznacza to, że zasoby te są zwalniane w sposób niejawny i dopisywane do listy zasobów, których proces oczekuje. Proces zostanie wznowiony dopiero wtedy, gdy będzie można mu przywrócić wszystkie jego dawne zasoby oraz dodać nowe, które zamawiał.

Można też postąpić inaczej: gdy proces zamawia jakieś zasoby, wówczas sprawdza się najpierw, czy są one dostępne. Jeśli tak, to następuje ich przydział. W przeciwnym razie sprawdza się, czy dane zasoby są przydzielone do innego procesu, który czeka na dodatkowe zasoby. Jeśli tak, to odbiera mu się te zasoby i przydziela procesowi aktualnie zamawiającemu. Jeśli zasoby nie są ani dostępne, ani przechowywane przez czekający proces, to proces zamawiający też musi zaczekać. Podeczas oczekiwania proces może utracić pewne zasoby, ale tylko wtedy, gdy inny proces ich zażąda. Proces będzie mógł być wznowiony tylko wtedy, gdy otrzyma nowe zasoby i odzyska zasoby utracone podeczas oczekiwania.

7.4.4 Czekanie cykliczne

Jednym ze sposobów zagwarantowania, że czekanie cykliczne nigdy nie wystąpi, jest wymuszenie całkowitego uporządkowania wszystkich typów zasobów i wymaganie, aby każdy proces zamawiał zasoby we wzrastającym porządku ich numeracji.

Niech $Z = \{Z_1, Z_2, \dots, Z_m\}$ będzie zbiorem typów zasobów. Każdemu typowi zasobu przyporządkujemy w sposób jednoznaczny liczbę całkowitą, która umożliwi porównywanie dwóch zasobów i określanie, czy jeden zasób poprzedza inny w naszym uporządkowaniu. Mówiąc formalnie, definiujemy wzajemnie jednoznaczną funkcję $F: Z \rightarrow N$, gdzie N jest zbiorem liczb naturalnych. Na przykład, jeśli zbiór Z typów zasobów zawiera przewijaki taśmy, napędy dysków i drukarki, to funkcja F mogłaby być zdefiniowana tak:

$$\begin{aligned} F(\text{przewijak taśmy}) &= 1, \\ F(\text{napęd dysku}) &= 5, \\ F(\text{drukarka}) &= 12. \end{aligned}$$

Mozemy teraz rozważyć następujący protokół zapobiegania zakleszczeniom. Każdy proces może zamawiać zasoby tylko we wzrastającym porządku ich numeracji. To znaczy, że proces może początkowo zamówić dowolną liczbę egzemplarzy zasobu typu – powiedzmy – Z . Potem proces może zamówić egzemplarze zasobu typu Z_i , lecz wyłącznie wtedy, gdy $F(Z_i) > F(Z)$. Jeśli potrzeba kilku egzemplarzy zasobu tego samego typu, to zamawia się je wszystkie za pomocą jednego zamówienia. Korzystając na przykład z poprzednio zdefiniowanej funkcji, proces, który chce używać jednocześnie przewijaka taśmy i drukarki, musi najpierw zamówić przewijak taśmy, a potem drukarkę.

Alternatywnie można wymagać, by proces zamawiający egzemplarz zasobu typu Z_i , miał zawsze zwolnione zasoby Z_j , takie że $F(Z_j) > F(Z_i)$.

Gdy stosuje się te protokole, wówczas warunek czekania cyklicznego nie może wystąpić. Aby to wykazać, założymy, że wystąpiło czekanie cykliczne (dowód nie wprost). Niech P_0, P_1, \dots, P_n oznacza zbiór procesów objętych czekaniem cyklicznym, przy czym proces P_i oczekuje na zasób Z_i , przetrzymywany przez proces P_{i+1} . (Stosujemy do indeksów arytmetykę modulo n , zatem P_n czeka na zasób Z_0 , przetrzymywany przez P_0). Wtedy, ponieważ proces P_{i+1} przetrzymuje zasób Z_i , i zamawia zasób Z_{i+1} , musi być $F(Z_i) < F(Z_{i+1})$ dla wszystkich i . Ale to by oznaczało, że $F(Z_0) < F(Z_1) < \dots < F(Z_n) < F(Z_0)$. Na mocy przechodniości otrzymujemy $F(Z_0) < F(Z_0)$, co jest niemożliwe. Zatem nie może również istnieć czekanie cykliczne.

Zwróciły uwagę, że funkcja F powinna być zdefiniowana zgodnie ze zwykłym porządkiem używania zasobów w systemie. Na przykład, ponieważ przewijak taśmy jest zwykłe potrzebny przed użyciem drukarki, zdefiniowanie $F(\text{przewijak taśmy}) < F(\text{drukarka})$ może być uzasadnione.

7.5 ■ Unikanie zakleszczeń

Algorytmy zapobiegania zakleszczeniom, omówione w p. 7.4, uniemożliwiały ich powstawanie przez nakładanie rygorów na wykonywanie zamówień. Gwarantowało to niespełnienie przynajmniej jednego z koniecznych warunków wystąpienia zakleszczenia, wobec czego nie mogło się ono pojawić. Możliwym skutkiem ubocznym zapobiegania zakleszczeniom jest jednak słabe wykorzystanie urządzeń i ograniczona przepustowość systemu.

Alternatywna metoda unikania zakleszczeń wymaga dodatkowych informacji o tym, jak będzie następowało zamawianie zasobów. Na przykład system z jednym przewijakiem taśmy i jedną drukarką może zostać powiadomiony, że proces P będzie najpierw zamawiał przewijak taśmy, a następnie drukarkę, zanim zwolni oba te zasoby. Natomiast proces Q może potrzebować najpierw drukarki, a potem przewijaka taśmy. Mając wszystkie informacje na temat kolejności występowania zamówień i zwolnień dla każdego procesu, system operacyjny może decydować przy każdym zamówieniu, czy proces powinien czekać, czy też nie. Przy każdym zamówieniu system będzie musiał wziąć pod uwagę zasoby bieżąco dostępne, zasoby przydzielone każdemu z procesów oraz przyszłe zamówienia i zwolnienia ze strony każdego procesu, aby zdecydować, czy bieżące zamówienie może być zrealizowane, czy też musi zostać odłożone w celu uniknięcia zakleszczenia w przyszłości.

Poszczególne algorytmy różnią się pod względem ilości i typu wymaganych informacji. W najprostszym i najbardziej użytecznym modelu zakłada się, że każdy proces zadeklaruje maksymalną liczbę zasobów każdego typu, których mógłby potrzebować. Dysponując z góry dla każdego procesu infor-

macią o wymaganej przez niego, nieprzekraczalnej ilości zasobów każdego typu, można zbudować algorytm zapewniający, że system nigdy nie wejdzie w stan zakleszczenia. Będzie to algorytm *unikania zakleszczenia* (ang. *deadlock-avoidance*). Algorytm unikania zakleszczenia sprawdza dynamicznie stan przydziału zasobów, aby zagwarantować, że nigdy nie dojdzie do spełnienia warunku czekania cyklicznego. Stan przydziału zasobów jest określony przez liczbę dostępnych i przydzielonych zasobów oraz przez maksymalne zapotrzebowania procesów.

7.5.1 Stan bezpieczny

Stan systemu jest *bezpieczny* (ang. *safe*), jeśli istnieje porządek, w którym system może przydzieć zasoby każdemu procesowi (nawet w stopniu maksymalnym), stale unikając zakleszczenia. Mówiąc bardziej formalnie, system jest w stanie bezpiecznym tylko wtedy, gdy istnieje *ciąg bezpieczny*. Ciąg procesów $\langle P_1, P_2, \dots, P_n \rangle$ jest bezpieczny w danym stanie przydziałów, jeśli dla każdego procesu P_j jego potencjalne zapotrzebowanie na zasoby może być zaspokojone przez bieżąco dostępne zasoby oraz zasoby użytkowane przez wszystkie procesy P_i , przy czym $j < i$. Jeśli więc zasoby, których wymaga proces P_n nie są natychmiast dostępne, to może on poczekać, aż zakończą się wszystkie procesy P_i . Po ich zakończeniu proces P_n może otrzymać wszystkie potrzebne mu zasoby, dokończyć przewidzianą pracę, oddać przydzielone zasoby i zakończyć działanie. Kiedy proces P_n będzie zakończony, wtedy niezbędne zasoby może otrzymać proces P_{n+1} itd. Jeśli żaden taki ciąg nie istnieje, to system określa się jako będący w stanie *zagrożenia* (ang. *unsafe*).

Stan bezpieczny nie jest stanem zakleszczenia. Odwrotnie, zakleszczenie jest stanem zagrożenia. Jednakże nie wszystkie stany zagrożenia są zakleszczeniami (rys. 7.4). Stan zagrożenia może prowadzić do zakleszczenia. Dopó-



Rys. 7.4 Obszary stanu bezpiecznego, stanu zagrożenia i zakleszczenia

ki stan jest bezpieczny, dopóty system operacyjny może unikać stanów zagrożenia (i zakleszczeń). W stanie zagrożenia system operacyjny nie może zapobiec zamówieniom procesów prowadzącym do zakleszczenia – zachowanie procesów steruje stanami zagrożenia.

W celu ilustracji rozważmy system z dwunastoma przewijakami taśm magnetycznych i trzema procesami: P_0, P_1, P_2 . Proces P_0 wymaga 10 przewijaków, proces P_1 może zażądać 4, a proces P_2 zamówi 9 przewijaków taśmy. Założmy, że w czasie t_0 proces P_0 ma 5 przewijaków, proces P_1 zajmuje 2 przewijaki i tyle samo przewijaków jest przydzielonych do procesu P_2 . (Trzy przewijaki taśmy są zatem wolne).

	<u>Maksymalne zapotrzebowanie</u>	<u>Bieżące wymagania</u>
P_0	10	5
P_1	4	2
P_2	9	2

W chwili t_0 system jest w stanie bezpiecznym. Ciąg $\langle P_1, P_0, P_2 \rangle$ spełnia warunek bezpieczeństwa, ponieważ procesowi P_1 można natychmiast przydzieleć wszystkie wymagane przez niego przewijaki, a kiedy nastąpi ich zwrot (w systemie będzie wówczas 5 wolnych przewijaków), wtedy komplet przewijaków będzie mógł otrzymać i zwrocić proces P_0 (system będzie mieć wówczas 10 dostępnych przewijaków), po czym wszystkie potrzebne przewijaki otrzyma i zwolni proces P_2 (system będzie wtedy miał dostępnych wszystkich 12 przewijaków).

Zauważmy, że jest możliwe przejście od stanu bezpiecznego do stanu zagrożenia. Założmy, że w chwili t_1 proces P_2 zamówi i otrzyma jeszcze jeden przewijak taśmy. Wówczas system przestaje być w stanie bezpiecznym. Od tej chwili tylko procesowi P_1 uda się przydzieleć wszystkie potrzebne przewijaki. Kiedy je zwróci, wtedy systemowi pozostaną tylko 4 przewijaki. Ponieważ proces P_0 ma przydzielonych 5 przewijaków, a jego maksymalne wymagania wynoszą 10, może potem wystąpić z zamówieniem następnych pięciu. Wskutek tego, że nie są one dostępne, proces P_0 będzie musiał czekać. Podobnie, proces P_2 może zapotrzebować dodatkowych sześciu przewijaków i będzie zmuszony czekać, co prowadzi do zakleszczenia.

Błądem, który tu popełniono, była zgoda na przydelenie procesowi P_2 jeszcze jednego przewijaka. Gdyby nakazano procesowi P_2 czekać dopóty, dopóki nie zakonczy się któryś z pozostałych procesów i nie odda swoich zasobów, wówczas można by było uniknąć powstania zakleszczenia.

Mając do dyspozycji pojęcie stanu bezpiecznego, możemy zdefiniować algorytmy unikania zakleszczeń, gwarantujące, że system nigdy nie wejdzie w stan zakleszczenia. Idea sprawdza się do zapewnienia, że system będzie stale pozostawał w stanie bezpiecznym. Na początku system jest w stanie bez-

piecznym. Za każdym razem, gdy proces zamawia zasób, który jest na bieżąco dostępny, system musi zdecydować, czy zasób wolno przydzielić natychmiast, czy też proces powinien poczekać. Zgoda na zamówienie zostaje udzielona tylko wtedy, kiedy przydział pozostawia system w stanie bezpiecznym.

Zauważmy, że w tym schemacie proces może oczekwać nawet wtedy, gdy zasób jest na bieżąco dostępny. Może to powodować mniejsze wykorzystanie zasobów niż wówczas, gdyby zrezygnowano z algorytmu unikania zakleszczenia.

7.5.2 Algorytm grafu przydziału zasobów

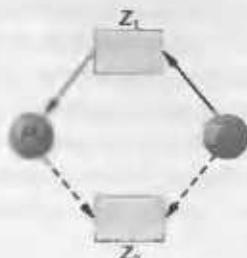
W systemie przydzielania zasobów, w którym każdy typ zasobu ma tylko jeden egzemplarz, można w celu unikania zakleszczenia zastosować odmianę grafu przydziału zasobów, zdefiniowanego w p. 7.2.2.

Oprócz krawędzi zamówień i krawędzi przydziałów wprowadzamy nowy typ krawędzi, nazywanej *krawędzią deklaracji* (ang. *claim edge*). Krawędź deklaracji $P_i \rightarrow Z_j$ wskazuje, że proces P_i może zamówić zasób Z_j , kiedyś w przyszłości. Krawędź ta ma taki sam zwrot jak krawędź zamówienia, lecz jest przedstawiana za pomocą linii przerywanej. Gdy proces P_i zamawia zasób Z_j , wówczas krawędź deklaracji $P_i \rightarrow Z_j$ jest zamieniana na krawędź zamówienia. Podobnie, gdy zasób Z_j jest zwalniany przez proces P_i , wtedy krawędź przydziału $Z_j \rightarrow P_i$ jest zamieniana z powrotem na krawędź deklaracji $P_i \rightarrow Z_j$. Podkreślimy, że oświadczenie o zapotrzebowaniu na zasoby muszą być zawczasu złożone w systemie. Tak więc, zanim proces P_i rozpocznie działanie, wszystkie jego krawędzie deklaracji muszą pojawić się w grafie przydziału zasobów. Możemy osłabić ten warunek, zezwalając na to, by krawędź deklaracji $P_i \rightarrow Z_j$ była dodawana do grafu tylko wtedy, gdy wszystkie krawędzie związane z procesem P_i są krawędziami deklaracji.

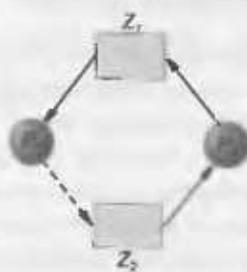
Załóżmy, że proces P_i zamawia zasób Z_j . Zamówienie może być spełnione tylko wtedy, gdy zamiana krawędzi zamówienia $P_i \rightarrow Z_j$ na krawędź przydziału $Z_j \rightarrow P_i$ nie spowoduje utworzenia cyklu w grafie przydziału zasobów. Zauważmy, że sprawdzenie bezpieczeństwa polega na użyciu algorytmu wykrywania cyklu w tym grafie. Liczba operacji algorytmu wykrywania cykli w grafie jest rzędu n^2 , jeśli n jest liczbą procesów w systemie.

Jesli nie ma żadnego cyklu, to przydział zasobu pozostawi system w stanie bezpiecznym. W wypadku znalezienia cyklu wykonanie przydziału wprowadzioby system w stan zagrożenia. Dlatego proces P_i będzie musiał poczekać na spełnienie swoich zamówień.

W celu zilustrowania tego algorytmu rozważymy graf przydziału zasobów z rys. 7.5. Założymy, że proces P_2 zamawia zasób Z_2 . Choć zasób Z_2 jest obecnie wolny, nie możemy przydzielić go procesowi P_2 , ponieważ działanie



Rys. 7.5 Graf przydziału zasobów do unikania zakleszczenia



Rys. 7.6 Stan zagrożenia w grafie przydziału zasobów

to spowodowałoby powstanie cyklu w grafie (rys. 7.6). Cykl wskazywałby na to, że system jest w stanie zagrożenia. Jeśli proces P_i zgłosiłby zamówienie na zasób Z_2 , to nastąpiłoby zakleszczenie.

7.5.3 Algorytm bankiera

Algorytm grafu przydziału zasobów nie nadaje się do systemu przydzielania zasobów, w którym każdy typ zasobu ma wiele egzemplarzy. Algorytm unikania zakleszczenia, który zaraz opiszymy, można zastosować w takim systemie, lecz jest on mniej wydajny od schematu grafu przydziału zasobów. Algorytm ten jest znany pod nazwą *algorytmu bankiera* (ang. *banker's algorithm*). Nazwę zawdzięcza temu, że mógłby on posłużyć w systemie bankowym do zagwarantowania, iż bank nigdy nie zainwestuje gotówki w sposób, który uniemożliwiłby mu zaspokojenie wymagań wszystkich jego klientów.

Gdy proces wchodzi do systemu, wówczas musi zadeklarować maksymalną liczbę egzemplarzy każdego typu zasobu, które będą mu potrzebne. Liczba ta nie może przekroczyć ogólnej liczby zasobów w systemie. Kiedy użytkownik zamawia zbiór zasobów, wtedy system musi określić, czy ich przydzielanie pozostawi system w stanie bezpiecznym. Jeśli tak, to zasoby zostaną przydzielone; w przeciwnym razie proces będzie musiał poczekać, aż inne procesy zwolnią wystarczającą ilość zasobów.

W implementacji algorytmu bankiera występuje kilka struktur danych. Struktury te przechowują stan systemu przydziału zasobów. Niech n będzie liczbą procesów w systemie, m zaś liczbą typów zasobów. Potrzebne są następujące struktury danych:

- *Dostępne*: Wektor o długości m , określający liczbę dostępnych zasobów każdego typu. $Dostępne[j] = k$ oznacza, że jest dostępnych k egzemplarzy zasobu typu Z_j .
- *Maksymalne*: Macierz o wymiarach $n \times m$, definiująca maksymalne żądania każdego procesu. Jeśli $Maksymalne[i, j] = k$, to proces P_i może zamówić co najwyżej k egzemplarzy zasobu typu Z_j .
- *Przydzielone*: Macierz o wymiarach $n \times m$, definiująca liczbę zasobów poszczególnych typów, przydzielonych do każdego z procesów. Gdy $Przydzielone[i, j] = k$, wówczas proces P_i ma przydzielonych k egzemplarzy zasobu typu Z_j .
- *Potrzebne*: Macierz o wymiarach $n \times m$, przechowująca pozostałe do spełnienia zamówienia każdego z procesów. Element $Potrzebne[i, j] = k$ oznacza, że do zakończenia swojej pracy proces P_i może jeszcze potrzebować k dodatkowych egzemplarzy zasobu typu Z_j . Zauważmy, że $Potrzebne[i, j] = Maksymalne[i, j] - Przydzielone[i, j]$.

W miarę upływu czasu struktury te zmieniają zarówno swoje wymiary, jak i wartości.

Dla uproszczenia prezentacji algorytmu bankiera uzgodnimy pewną notację. Niech X i Y będą wektorami długości n . Powiemy, że $X \leq Y$ wtedy i tylko wtedy, gdy $X[i] \leq Y[i]$ dla każdego $i = 1, 2, \dots, n$. Na przykład, jeżeli $X = (1, 7, 3, 2)$ i $Y = (0, 3, 2, 1)$, to $Y \leq X$. Jeśli $Y \leq X$ i $Y \neq X$, to $Y < X$.

Wiersze w macierzach *Przydzielone* i *Potrzebne* możemy uważać za wektory i odwoływać się do nich odpowiednio jako do *Przydzielone*, i *Potrzebne*. *Przydzielone*, określa zasoby aktualnie przydzielone do procesu P_i , a *Potrzebne*, określa dodatkowe zasoby, których proces P_i może jeszcze potrzebować do zakończenia zadania.

7.5.3.1 Algorytm bezpieczeństwa

Algorytm rozstrzygania, czy system jest, czy nie jest w stanie bezpiecznym, można opisać w sposób następujący:

1. Niech *Roboczy* i *Końcowy* oznaczają wektory o długości odpowiednio m i n . Na początku wykonujemy przypisania: *Roboczy* := *Dostępne* i *Końcowy* [i] := *false* dla $i = 1, 2, \dots, n$

2. Znajdujemy i takie, że zarówno

(a) *Końcowy* [*i*] = *false*, jak i

(b) *Potrzebne*, \leq *Roboczy*.

Jeśli takie *i* nie istnieje, to wykonujemy krok 4.

3. *Roboczy* := *Roboczy* + *Przydzielone*;

Końcowy[*i*] := *true*

Tu następuje skok do punktu 2.

4. Jeśli *Końcowy* [*i*] = *true* dla wszystkich *i*, to system jest w stanie bezpiecznym.

Rząd operacji wymagany w tym algorytmie do roztrzygnięcia o stanie bezpiecznym wynosi $m \times n^2$.

7.5.3.2 Algorytm zamawiania zasobów

Niech *Zamówienia*, oznacza wektor zamówień dla procesu *P*. Jeśli *Zamówienia*, [*j*] = *k*, to proces *P*, potrzebuje *k* egzemplarzy zasobu typu *Z_j*. Kiedy proces *P*, wykonuje zamówienie, wtedy są podejmowane następujące działania:

1. Jeśli *Zamówienia*, \leq *Potrzebne*, to wykonaj krok 2. W przeciwnym razie zgłoś sytuację błędą, ponieważ proces przekroczył deklarowane maksimum.
2. Jeśli *Zamówienia*, \leq *Dostępne*, to wykonaj krok 3. W przeciwnym razie proces *P*, musi czekać, ponieważ zasoby są niedostępne.
3. System próbuje przydzielić żądane zasoby procesowi *P*, zmieniając stan w następujący sposób:

Dostępne := *Dostępne* - *Zamówienia*;

Przydzielone := *Przydzielone* + *Zamówienia*;

Potrzebne := *Potrzebne* - *Zamówienia*;

Jeśli stan wynikający z przydziału zasobów jest bezpieczny, to transakcja dochodzi do skutku i proces *P*, otrzymuje zamawiane zasoby. Jednakże gdy nowy stan nie jest bezpieczny, wówczas proces *P*, musi czekać na realizację zamówienia *Zamówienia*, oraz jest przywracany poprzedni stan przydziału zasobów.

7.5.3.3 Przykładowa ilustracja

Rozważmy system z pięcioma procesami od *P₀* do *P₄* i trzema typami zasobów, *A*, *B*, *C*. Zasób typu *A* ma 10 egzemplarzy, zasób typu *B* ma 5 egzempla-

rzy, zasób typu C ma 7 egzemplarzy. Założmy, że w chwili t_0 migawkowy obraz systemu przedstawiał się następująco:

	<u>Przydzielone</u>			<u>Maksymalne</u>			<u>Dostępne</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Zawartość macierzy *Potrzebne* jest określona jako *Maksymalne – Przydzielone* i wynosi

	<u>Potrzebne</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Przymajemy, że system jest obecnie w stanie bezpiecznym. Rzeczywiście, ciąg $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ spełnia kryterium bezpieczeństwa.

Założymy teraz, że proces P_1 zamawia jeden dodatkowy egzemplarz zasobu typu A i dwa egzemplarze zasobu typu C, czyli $Zamówienia_1 = (1, 0, 2)$. Aby zadecydować, czy zamówienie może być natychmiast spełnione, sprawdzamy najpierw, że $Zamówienia_1 \leq Dostępne$ (tj. $(1, 0, 2) \leq (3, 3, 2)$), co jest prawdą. Następnie zakładamy, że zamówienie zostaje spełnione. Powoduje to otrzymanie następującego stanu:

	<u>Przydzielone</u>			<u>Potrzebne</u>			<u>Dostępne</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Musimy określić, czy ten nowy stan systemu jest bezpieczny. Aby tego dokonać, wykonujemy nasz algorytm bezpieczeństwa i znajdujemy, że ciąg

Relacja mniejszości ($<$) między dwoma wektorami jest zdefiniowana jak w p. 7.5.3. W celu uproszczenia notacji, jak uprzednio, traktujemy wiersze macierzy *Przydzielone* i *Zamówienia* jako wektory oraz odwołujemy się do nich odpowiednio jako do *Przydzielone*, i *Zamówienia*. Opisany tu algorytm wykrywania zakleszczenia bada każdy możliwy ciąg przydziałów dla procesów, które należy dokonczyć. Zachęcamy do porównania tego algorytmu z algorytmem bankiera z p. 7.5.3.

1. Niech *Roboczy* i *Końcowy* będą wektorami o długości odpowiednio m oraz n . Na początku podstawiamy *Roboczy* := *Dostępne* oraz, dla $i = 1, 2, \dots, n$, jeśli *Przydzielone* $[i] \neq 0$, to *Końcowy* $[i] := \text{false}$, w przeciwnym razie *Końcowy* $[i] := \text{true}$.
2. Znajdujemy indeks i taki, że zachodzą oba związki:
 - (a) *Końcowy* $[i] = \text{false}$ oraz
 - (b) *Zamówienia* $\leq Roboczy$.
 Jeśli nie ma takiego i , to wykonujemy krok 4.
3. *Roboczy* := *Roboczy* + *Przydzielone*; *Końcowy* $[i] := \text{true}$. Idziemy do kroku 2.
4. Jeśli dla pewnych i z przedziału $1 \leq i \leq n$ zachodzi *Końcowy* $[i] = \text{false}$, to system jest w stanie zakleszczenia. Co więcej, jeśli *Końcowy* $[i] = \text{false}$, to zakleszczenie dotyczy procesu P_i .

Rząd liczby operacji tego algorytmu, potrzebnych do wykrycia, czy system jest w stanie zakleszczenia, wynosi $m \times n^2$.

Zdziwienie może budzić fakt odbierania zasobów procesowi P_i (w kroku 3), gdy tylko okaże się, że *Zamówienia* $\leq Roboczy$ (w kroku 2b). Wiemy, że proces P_i nie jest aktualnie zakleszczony (ponieważ *Zamówienia* $\leq Roboczy$). Toteż zakładamy optymistycznie, że proces P_i nie będzie dłużej potrzebować zasobów do wypełnienia swojego zadania i niedługo zwróci systemowi wszystkie obecnie przydzielone zasoby. Jeśli nasze założenie okaze się błędne, to w późniejszym czasie może wystąpić zakleszczenie. Zostanie ono wykryte podczas następnego wykonania algorytmu wykrywania zakleszczenia.

Aby zilustrować ten algorytm, rozważmy system z pięcioma procesami, od P_0 do P_4 , i trzema typami zasobów: *A*, *B*, *C*. Zasób typu *A* ma 7 egzemplarzy, zasób typu *B* ma 2 egzemplarze, a zasób typu *C* ma 6 egzemplarzy. Założymy, że w chwili t_0 stan przydziału zasobów kształtuje się jak niżej:

	<u>Przydzielone</u>			<u>Zamówienia</u>			<u>Dostępne</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Przymajemy, że system nie jest w stanie zakleszczenia. Rzeczywiście, jeśli wykonamy nasz algorytm, to stwierdzimy, że ciąg $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ da w wyniku $Końcowy[i] = true$ dla wszystkich i .

Załóżmy teraz, że proces P_2 zamawia dodatkowo egzemplarz zasobu typu C. Macierz Zamówienia przyjmuje postać:

	<u>Zamówienia</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Teraz dochodzi w systemie do zakleszczenia. Nawet gdybyśmy odebrali zasoby przetrzymywane przez proces P_0 , to dostępnych zasobów nie wystarczy, aby spełnić zamówienia innych procesów. Istnieje zatem zakleszczenie dotyczące procesów P_1, P_2, P_3 i P_4 .

7.6.3 Użytkowanie algorytmu wykrywania zakleszczenia

Kiedy należy wywoływać algorytm wykrywania zakleszczenia? Odpowiedź zależy od dwóch czynników:

1. Jak często może wystąpić zakleszczenie?
2. Ile procesów ulegnie zakleszczeniu w przypadku jego wystąpienia?

Jesli zakleszczenia zdarzają się często, to algorytm ich wykrywania powinien być wywoływany często. Zasoby przydzielone zakleszczonym procesom będą pozostawały bezużyteczne do czasu wydobycia ich z tego stanu. Co więcej, liczba zakleszczonych procesów może rosnąć.

Groźba zakleszczeń występuje tylko wtedy, gdy jakiś proces zgłasza zamówienie, które nic może być natychmiast zrealizowane. Jest możliwe, że

Relacja mniejszości ($<$) między dwoma wektorami jest zdefiniowana jak w p. 7.5.3. W celu uproszczenia notacji, jak uprzednio, traktujemy wiersze macierzy *Przydzielone* i *Zamówienia* jako wektory oraz odwołujemy się do nich odpowiednio jako do *Przydzielone*, i *Zamówienia*. Opisany tu algorytm wykrywania zakleszczenia bada każdy możliwy ciąg przydziałów dla procesów, które należy dokonczyć. Zachęcamy do porównania tego algorytmu z algorytmem bankiera z p. 7.5.3.

1. Niech *Roboczy* i *Końcowy* będą wektorami o długości odpowiednio m oraz n . Na początku podstawiamy *Roboczy* := *Dostępne* oraz, dla $i = 1, 2, \dots, n$, jeśli *Przydzielone* $[i] \neq 0$, to *Końcowy* $[i] := \text{false}$, w przeciwnym razie *Końcowy* $[i] := \text{true}$.
2. Znajdujemy indeks i taki, że zachodzą oba związki:
 - (a) *Końcowy* $[i] = \text{false}$ oraz
 - (b) *Zamówienia* \leq *Roboczy*.
 Jeśli nie ma takiego i , to wykonujemy krok 4.
3. *Roboczy* := *Roboczy* + *Przydzielone*;
Końcowy $[i] := \text{true}$.
 Idziemy do kroku 2.
4. Jeśli dla pewnych i z przedziału $1 \leq i \leq n$ zachodzi *Końcowy* $[i] = \text{false}$, to system jest w stanie zakleszczenia. Co więcej, jeśli *Końcowy* $[i] = \text{false}$, to zakleszczenie dotyczy procesu P_i .

Rząd liczby operacji tego algorytmu, potrzebnych do wykrycia, czy system jest w stanie zakleszczenia, wynosi $m \times n^2$.

Zdziwienie może budzić fakt odbierania zasobów procesowi P_i (w kroku 3), gdy tylko okaże się, że *Zamówienia* \leq *Roboczy* (w kroku 2b). Wiemy, że proces P_i nie jest aktualnie zakleszczony (ponieważ *Zamówienia* \leq *Roboczy*). Toteż zakładamy optymistycznie, że proces P_i nie będzie dłużej potrzebować zasobów do wypełnienia swojego zadania i niedługo zwróci systemowi wszystkie obecnie przydzielone zasoby. Jeśli nasze założenie okaze się błędne, to w późniejszym czasie może wystąpić zakleszczenie. Zostanie ono wykryte podczas następnego wykonania algorytmu wykrywania zakleszczenia.

Aby zilustrować ten algorytm, rozważmy system z pięcioma procesami, od P_0 do P_4 , i trzema typami zasobów: *A*, *B*, *C*. Zasób typu *A* ma 7 egzemplarzy, zasób typu *B* ma 2 egzemplarze, a zasób typu *C* ma 6 egzemplarzy. Założymy, że w chwili t_0 stan przydziału zasobów kształtuje się jak niżej:

	<u>Przydzielone</u>			<u>Zamówienia</u>			<u>Dostępne</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Przymajemy, że system nie jest w stanie zakleszczenia. Rzeczywiście, jeśli wykonamy nasz algorytm, to stwierdzimy, że ciąg $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ da w wyniku $Końcowy [i] = true$ dla wszystkich i .

Załóżmy teraz, że proces P_2 zamawia dodatkowo egzemplarz zasobu typu C. Macierz Zamówienia przyjmuje postać:

	<u>Zamówienia</u>		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Teraz dochodzi w systemie do zakleszczenia. Nawet gdybyśmy odebrali zasoby przetrzymywane przez proces P_0 , to dostępnych zasobów nie wystarczy, aby spełnić zamówienia innych procesów. Istnieje zatem zakleszczenie dotyczące procesów P_1, P_2, P_3 i P_4 .

7.6.3 Użytkowanie algorytmu wykrywania zakleszczenia

Kiedy należy wywoływać algorytm wykrywania zakleszczenia? Odpowiedź zależy od dwóch czynników:

1. Jak często może wystąpić zakleszczenie?
2. Ile procesów ulegnie zakleszczeniu w przypadku jego wystąpienia?

Jeśli zakleszczenia zdarzają się często, to algorytm ich wykrywania powinien być wywoływany często. Zasoby przydzielone zakleszczonym procesom będą pozostawały bezużyteczne do czasu wydobycia ich z tego stanu. Co więcej, liczba zakleszczonych procesów może rosnąć.

Groźba zakleszczeń występuje tylko wtedy, gdy jakiś proces zgłasza zamówienie, które nie może być natychmiast zrealizowane. Jest możliwe, że

zamówienie takie jest finalną potrzebą, której zaspokojenie spowodowałoby zakończenie całego łańcucha czekających procesów. W skrajnym przypadku algorytm wykrywania zakleszczenia może być wywoływany za każdym razem, gdy zamówienie na przydział nie może być spełnione natychmiast. Można wówczas zidentyfikować nie tylko zbiór zakleszczonych procesów, lecz również proces, który do tego doprowadził. (W rzeczywistości każdy z zablokowanych procesów jest ogniwem cyklu w grafie przydziału zasobów, zatem wszystkie one łącznie powodują zakleszczenia). Gdy jest wiele typów zasobów, wówczas jedno zamówienie może spowodować powstanie wielu cykli w grafie zasobów, domykając każdy z nich; „powodujący” je proces daje się zidentyfikować.

Czywiście wykonywanie algorytmu wykrywania zakleszczenia przy każdym zamówieniu może prowadzić do nadmiernego czasu obliczeń. Mniej kosztowna możliwość polega po prostu na jego rzadszym wykonywaniu – na przykład raz na godzinę lub każdorazowo wówczas, gdy wykorzystanie procesora spadnie poniżej 40%. (Zakleszczenie, przedżej czy później, dławia przepustowość systemu i powoduje spadek wykorzystania procesora). Jeśli algorytm wykrywania jest wykonywany w dowolnych chwilach, to w grafie zasobów może powstać wiele cykli. Wskazanie pośród wielu zakleszczonych procesów „sprawcy” zakleszczenia może na ogół nie być wykonalne.

7.7 ■ Likwidowanie zakleszczenia

Kiedy algorytm wykryje zakleszczenie, wtedy można postąpić na kilka sposobów. Można poinformować operatora o wystąpieniu zakleszczenia i pozwolić mu na ręczne jego usunięcie. Inną możliwością jest pozwolenie systemowi, aby automatycznie usunął zakleszczenie. Są dwa sposoby likwidowania zakleszczenia. Jednym jest po prostu usunięcie jednego lub kilku procesów w celu przerwania czekania cyklicznego. Drugi sposób polega na odebraniu pewnych zasobów jednemu lub kilku procesom, których dotyczy zakleszczenie.

7.7.1 Zakończenie procesu

Aby zlikwidować zakleszczenie przez zaniechanie procesu, używa się jednej z dwóch metod. W obu metodach system odzyskuje wszystkie zasoby przydzielone zakończonym procesom.

- **Zaniechanie wszystkich zakleszczonych procesów:** Metoda ta rozrywa cykl zakleszczenia, lecz ponoszony przy tym koszt jest znaczny, ponieważ

waż likwidowane procesy mogły wykonywać swoje obliczenia od dawna, a ich wyniki częściowe zostaną zniszczone; prawdopodobnie spowoduje to konieczność późniejszego liczenia ich od nowa.

- **Usuwanie procesów pojedynczo, aż do wyeliminowania cyklu zakleszczenia:** Ta metoda wymaga sporego nakładu pracy na powtarzanie wykonywania algorytmu wykrywania zakleszczenia po każdym usunięciu procesu w celu sprawdzenia, czy pozostałe procesy nadal są zakleszczone.

Zauważmy, że zaniechanie procesu może nie być łatwe. Jeśli proces uaktualniał dane w pliku, to nagle zakończenie go pozostawiłoby plik w nieokreślonym stanie. Podobnie, gdyby proces właśnie drukował dane na drukarce, to przed przekazaniem drukarki następnemu zadaniu system musiałby doprowadzić ją do stanu początkowego.

Jeśli wybiera się metodę etapowego kończenia procesów, to należy rozstrzygać, który proces (lub procesy) należy zakończyć w celu usunięcia zakleszczenia. Jest to decyzja polityczna, podobnie jak w przypadku rozstrzygania o przydzielaniu procesora. Musimy odpowiedzieć na pytanie natury ekonomicznej, ponieważ powinniśmy zaniechać tych procesów, których przedwcześnie zakończenie pociągnie za sobą minimalne koszty. Niestety, termin *minimalne koszty* nie jest tu precyzyjnie określony. Na wybór procesu może się złożyć wiele czynników, w tym udzielenie odpowiedzi na poniższe pytania:

1. Jaki jest priorytet procesu?
2. Jak długo proces wykonywał już obliczenia oraz ile czasu potrzebna mu jeszcze do zakończenia przydzielonej mu pracy?
3. Ile zasobów i jakiego typu znajduje się w użytkowaniu procesu (np. czy zasoby te są łatwe do wywłaszczenia)?
4. Ilu jeszcze zasobów proces potrzebuje do zakończenia działania?
5. Ile procesów trzeba będzie przerwać?
6. Czy proces jest interakcyjny czy wsadowy?

7.7.2 Wywłaszczanie zasobów

W celu usunięcia zakleszczenia za pomocą wywłaszczenia zasobów stopniowo odbiera się pewne zasoby jednym procesem i przydziela innym, aż cykl zakleszczenia zostanie przerwany.

Jeśli do zwalczania zakleszczeń stosuje się wywłaszczenia, to należy uwzględnić następujące trzy kwestie:

- Wybór ofiary:** Które zasoby i które procesy mają ulec wywłaszczeniu? Tak jak przy konczeniu procesów, porządek wywłaszczeń powinien minimalizować ponoszone koszty. Na koszty mogą składać się takie parametry, jak liczba zasobów, które przetrzymuje zakleszczony proces, i czas, który zakleszczony proces zdążył już zużytkować na swoje wykonanie.
- Wycofanie:** Co wypadnie zrobić z procesem, który zostanie wywłaszczony z zasobu? Jest jasne, że – pozbawiony jednego z niezbędnych zasobów – nie będzie mógł kontynuować swojego normalnego działania. Trzeba będzie wycofać proces do jakiegoś bezpiecznego stanu, z którego można go będzie wznowić.
- Głodzenie:** W jaki sposób zagwarantować, że nie będzie dochodzić do głodzenia procesu? To znaczy, jak zapewnić, że wywłaszczanie nie będzie dotyczyć stale tego samego procesu?

W systemie, w którym wybór ofiary opiera się przede wszystkim na ocenie kosztów, może się zdarzyć, że w roli ofiary będzie wybierany wciąż ten sam proces. Wskutek tego proces nigdy nie zakończy swojej pracy – jest to zjawisko głodzenia, któremu należy zaradzić w każdym używanym w praktyce systemie. Mówiąc wprost, należy zapewnić, by proces mógł być wydelegowany do roli ofiary tylko skońzoną (małą) liczbą razy. Najpowszechniejszym rozwiązaniem jest uwzględnienie liczby wycofań przy ocenie kosztów.

7.8 ■ Mieszane metody postępowania z zakleszczeniami

Specjalisci utrzymują, że podstawowe strategie postępowania z zakleszczeniami (zapobieganie, unikanie i wykrywanie) stosowane indywidualnie nie nadają się do rozwiązywania wszystkich problemów związanych z przydzielaniem zasobów w systemach operacyjnych. Jedną z możliwości jest połączenie tych trzech podstawowych metod, pozwalające na uzyskanie optymalnego podejścia dla poszczególnych klas zasobów w systemie. Proponowana metoda jest oparta na spostrzeżeniu, iż zasoby można podzielić na hierarchicznie upo-

rządowane klasy. Do klas tych stosuje się technikę porządkowania zasobów opisaną w p. 7.4.4. W obrębie klas można dobierać najodpowiedniejsze sposoby postępowania z zakleszczeniami.

Łatwo wykazać, że system, w którym zastosuje się tę strategię, nie będzie narażony na zakleszczenia. W istocie, dzięki uporządkowaniu zasobów zakleszczenia nie będą mogły dotyczyć więcej niż jednej klasy. Wewnątrz danej klasy będzie stosowana jedna z podstawowych metod. W efekcie system nie będzie podatny na zakleszczenia.

Aby zobrazować tę technikę, rozważymy system składający się z następujących czterech klas zasobów:

- **Zasoby wewnętrzne:** Zasoby używane przez system, takie jak bloki kontrolne procesów.
- **Pamięć główna:** Pamięć używana przez zadanie użytkownika.
- **Zasoby zadania:** Przydzielane urządzenia (w rodzaju przewijaków taśm) i pliki.
- **Wymienny obszar pamięci:** Obszar w pamięci pomocniczej, przeznaczony na poszczególne zadania użytkowników.

W możliwym mieszanym rozwiążaniu problemu zakleszczeń w tym systemie porządkuje się klasy w sposób pokazany powyżej i stosuje się następujące metody do każdej z klas:

- **Zasoby wewnętrzne:** Można zapobiegać powstawaniu zakleszczeń dzięki uporządkowaniu zasobów, ponieważ nie trzeba dokonywać wyborów między realizowanymi zamówieniami.
- **Pamięć główna:** Można zapobiegać powstawaniu zakleszczeń, stosując wywłaszczenie, ponieważ zawsze można przesłać obraz zadania do pamięci pomocniczej, co pozwala na wywłaszczenie procesów z pamięci głównej.
- **Zasoby zadania:** Można zastosować technikę unikania zakleszczeń, ponieważ niezbędne informacje o zapotrzebowaniu na zasoby mogą być uzyskane z kart sterujących zadania.
- **Wymienny obszar pamięci:** Można zastosować wstępny przydział, ponieważ maksymalne wymagania na pamięć są na ogół znane. Przykład powyższy ukazuje, w jaki sposób, w ramach uporządkowania zasobów, można łączyć ze sobą różne podstawowe podejścia w celu otrzymania efektywnego rozwiązania problemu zakleszczenia.

7.9 ■ Podsumowanie

Stan zakleszczenia występuje wtedy, gdy dwa lub więcej procesów czeka w nieskończoność na zdarzenie, które może być spowodowane tylko przez jeden z tych czekających procesów. Mówiąc ogólnie, z zakleszczeniami można postępować na trzy sposoby:

- przestrzegać pewnego protokołu zapewniającego, że system nigdy nie wejdzie w stan zakleszczenia;
- pozwolić na występowanie zakleszczeń w systemie i wycofywanie się z nich;
- zlekceważyć problem w ogóle, uważając, że zakleszczenia nigdy nie wystąpią w systemie; takie rozwiązanie przyjęto w większości systemów operacyjnych, m.in. w systemie UNIX.

Do zakleszczenia może dojść wtedy i tylko wtedy, gdy w systemie są spełnione jednocześnie cztery konieczne warunki: wzajemne wykluczanie, przetrzymywanie i oczekiwanie, brak wywłaszczeń i czekanie cykliczne. Aby zapobiegać zakleszczeniom, należy zapewnić, że przynajmniej jeden z tych czterech warunków koniecznych nigdy nie będzie zachodził.

Inna metoda unikania zakleszczeń, mniej rygorystyczna od poprzedniego algorytmu zapobiegania, sprowadza się do pozyskania z góry informacji o przyszłym użytkowaniu zasobów przez każdy z procesów. Algorytm bankiera wymaga znajomości maksymalnej liczby zasobów każdej klasy, które mogą być zamawiane przez proces. Używając tych informacji, można zdefiniować algorytm unikania zakleszczenia.

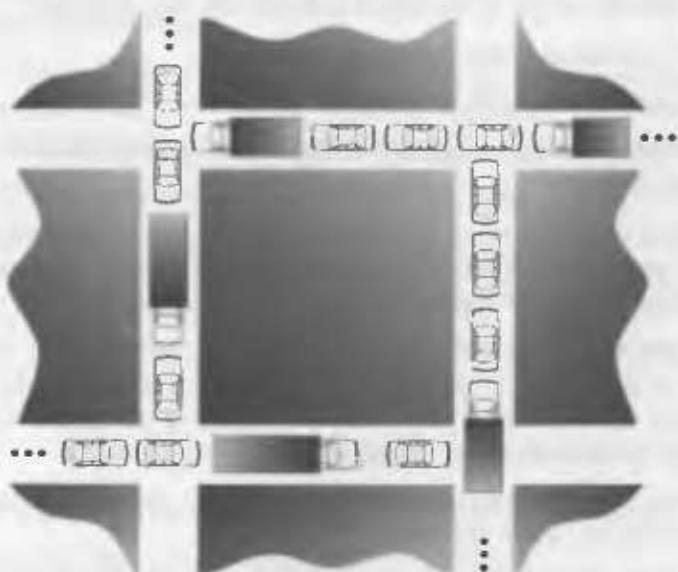
Jeśli w systemie nie zastosowano protokołu zapewniającego, że nie może dochodzić do zakleszczeń, to trzeba posłużyć się schematem wykrywania i usuwania zakleszczenia. Aby określić, czy wystąpiło zakleszczenie, należy wykonać algorytm wykrywania zakleszczenia. W przypadku wykrycia zakleszczenia system musi je usuwać za pomocą przedwczesnego kończenia niektórych z zakleszczonych procesów albo przez odbieranie niektórym z nich pewnych zasobów.

W systemie, w którym wybór ofiar do wycofania zależy w głównej mierze od oceny kosztów, może wystąpić głodzenie. W jego wyniku nieustannie wycofywany proces nigdy nie kończy wyznaczonej pracy.

Znawcy tematu są na koniec zdania, że żadna z podstawowych metod z osobna nie wystarcza do rozwiązywania wszystkich problemów przydziału zasobów w systemach operacyjnych. Podstawowe metody można stosować łącznie na zasadzie dokonywania oddzielnych wyborów optymalnego postępowania z poszczególnymi klasami zasobów w systemie.

■ Ćwiczenia

- 7.1 Podaj trzy przykłady zakleszczeń nie dotyczących środowisk systemów komputerowych.
- 7.2 Czy jest możliwe zakleszczenie obejmujące tylko jeden proces? Odpowiedź uzasadnij.
- 7.3 Powiada się, że umiejętnie zastosowany spooling eliminuje zakleszczenia. Z pewnością eliminuje on rywalizację o czytniki kart, plotery, drukarki itd. Możliwa jest nawet dyskowa symulacja taśm magnetycznych (nazywana przemieszczaniem – ang. *staging*). Do rozważenia pozostają więc tylko zasoby w postaci czasu procesora, pamięci operacyjnej i dyskowej. Czy jest możliwe zakleszczenie z udziałem tych zasobów? Jeśli tak, to w jaki sposób może powstać? Jeżeli nie, to dlaczego? Który ze schematów postępowania z zakleszczeniami wydaje się najlepszy do eliminowania tego rodzaju zakleszczeń (jeśli w ogóle są jakieś możliwe), a jeżeli zakleszczenia z udziałem tych zasobów są niemożliwe, to wskutek niespełnienia którego z warunków koniecznych?
- 7.4 Rozważ zakleszczenie w postaci korka ulicznego zobrazowanego na rys. 7.8.



Rys. 7.8 Korek uliczny do čw. 7.4

- (a) Wykaż, że cztery warunki konieczne do wystąpienia zakleszczenia są w tym przykładzie rzeczywiście spełnione.
- (b) Określ prostą regułę pozwalającą unikać zakleszczeń w tym systemie.
- 7.5 Założmy, że system jest w stanie zagrożenia. Pokaż, że jest możliwe, aby procesy ukończyły swą pracę bez zakleszczeń.
- 7.6 W rzeczywistym systemie komputerowym po dłuższych okresach użytkowania (miesiące) zarówno dostępne zasoby, jak i zapotrzebowania procesów ulegają zmianom. Zasoby zużywają się lub są zastępowane innymi, stale przychodzą i odchodzą nowe procesy, zakupuje się i włącza do systemu nowe zasoby. Zakładając, że zakleszczenie jest kontrolowane za pomocą algorytmu bankiera, określ, które z poniższych zmian mogą być wykonane bezpiecznie (bez wprowadzania możliwości zakleszczenia) i przy jakich założeniach?
- Zwiększenie wartości zmiennej *Dostępne* (dodanie nowych zasobów).
 - Zmniejszenie wartości zmiennej *Dostępne* (stałe usuwanie zasobu z systemu).
 - Zwiększenie wartości zmiennej *Maksymalne* dla pewnego procesu (proces żąda większej liczby zasobów aniżeli mu wolno).
 - Zmniejszenie wartości zmiennej *Maksymalne* dla pewnego procesu (proces uznaje, że nie będzie potrzebował tylu zasobów).
 - Zwiększenie liczby procesów.
 - Zmniejszenie liczby procesów.
- 7.7 Udowodnij, że liczba operacji w algorytmie bezpieczeństwa z p. 7.5.3 jest rzędu $m \times n^2$.
- 7.8 Rozważ system złożony z czterech zasobów tego samego typu, dzielonych przez trzy procesy, z których każdy potrzebuje co najwyżej dwóch zasobów. Pokaż, że system taki jest wolny od zakleszczeń.
- 7.9 Rozważ system złożony z m zasobów tego samego typu, dzielonych przez n procesów. Procesy mogą zamawiać i zwalniać zasoby tylko po jednym egzemplarzu w danej chwili. Wykaż, że ten system nie jest narażony na zakleszczenie, jeśli są spełnione dwa warunki:
- Maksymalne* zapotrzebowanie każdego procesu zawiera się w przedziale od 1 do m zasobów.
 - Suma wszystkich maksymalnych zapotrzebowień jest mniejsza od $m + n$.

- 7.10** Rozważmy system wykonujący 5000 zadań miesięcznie bez żadnych środków zapobiegania zakleszczeniom lub ich unikania. Zakleszczenia występują przeciętnie dwa razy w miesiącu. Operator musi kończyć i wykonywać od nowa średnio 10 zadań z powodu jednego zakleszczenia. Każde zadanie kosztuje około 2 dolarów (czas procesora), a usuwanie zadań są zazwyczaj w połowie wykonane.

Programista systemowy oszacował, że zainstalowanie w systemie algorytmu unikania zakleszczenia (w rodzaju algorytmu bankiera) zwiększyłoby średni czas wykonania zadania o około 10%. Ponieważ maszyna w chwili obecnej stoi bezczynnie przez 30% czasu, nadal można by wykonywać wszystkie 5000 zadań, choć czas cyklu przetwarzania wzrosłby średnio o około 20%.

- (a) Jakie argumenty przemawiają za zainstalowaniem algorytmu unikania zakleszczenia?
 - (b) Jakie argumenty można wysunąć przeciw instalowaniu algorytmu unikania zakleszczenia?
- 7.11** Z ogólnego algorytmu bankiera można otrzymać algorytm bankiera do obsługi zasobu jednego typu, zmniejszając po prostu liczbę wymiarów używanych w nim tablic do 1. Pokaż na przykładzie, że schematu bankiera dla wielu typów zasobów nie da się zaimplementować za pomocą stosowania do każdego zasobu z osobna schematu odnoszącego się do zasobu jednego typu.
- 7.12** Czy system może wykryć, że pewne jego procesy są głodzone? Jeśli odpowiesz „tak”, to wyjaśnij, w jaki sposób? Jeśli odpowiesz „nie”, to wyjaśnij, jak system może poradzić sobie z problemem głodzenia?
- 7.13** Rozważmy następującą migawkę stanu systemu:

	<i>Przydzielone</i>				<i>Maksymalne</i>				<i>Dostępne</i>			
	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Posługując się algorytmem bankiera, odpowiedz na następujące pytania:

- (a) Co zawiera macierz zapotrzebowień *Potrzebne*?
- (b) Czy system jest w stanie bezpiecznym?

(c) Jeśli proces P_1 złoży zamówienie $(0, 4, 2, 0)$, to czy będzie możliwe jego natychmiastowe spełnienie?

- 7.14** Rozważmy następującą politykę przydziału zasobów. Zamówienia i zwroty zasobów są możliwe w dowolnym czasie. Jeśli zamówienie na zasób nie może być spełnione z powodu niedostępności danego zasobu, to następuje sprawdzenie każdego procesu zablokowanego z powodu oczekiwania na zasoby. Jeśli w grupie takich procesów znajduje się żądana pułapka zasobów, to zasoby te odbierają się procesom i daje procesowi zamawiającemu. Wektor zasobów, na które proces oczekuje, powiększa się o odebrane mu zasoby.

Załóżmy na przykład, że system ma trzy typy zasobów i wektor *Dostępne* zainicjowany na $(4, 2, 2)$. Jeśli proces P_0 złoży zamówienie $(2, 2, 1)$, to jego prośba zostanie spełniona. Również zamówienie $(1, 0, 1)$ procesu P_1 zostanie załatwione bez zwłoki. Gdy następnie proces P_0 poprosi o $(0, 0, 1)$, wówczas zostanie zablokowany (z braku dostępnego zasobu). Jeśli obecnie proces P_2 poprosi o $(2, 0, 0)$, to otrzyma jedynie dostępny zasób $(1, 0, 0)$ i jeden z tych, które były przydzielone do procesu P_0 (ponieważ P_0 jest zablokowany). Wektor *Przydzielone* procesu P_0 zmniejsza się do wartości $(1, 2, 1)$, a wartość jego wektora *Potrzebne* wzrosnie do $(1, 0, 1)$.

(a) Czy może tu powstać zakleszczenie? Jeśli tak, to podaj przykład. Jeśli nie, to wskaz, który z koniecznych warunków nie jest spełniony?

(b) Czy może wystąpić nieskończone blokowanie się procesu?

- 7.15** Założymy, że masz już zakodowany algorytm bezpieczeństwa polegający na unikaniu zakleszczeń i teraz poproszono Cię o realizację algorytmu wykrywania zakleszczenia. Czy możesz tego dokonać, zmieniając po prostu w kodzie algorytmu bezpieczeństwa definicję *Maksymalne_i* = *Oczekiwane_i* + *Przydzielone_i*, w której *Oczekiwane_i* oznacza wektor określający zasoby oczekiwane przez i -ty proces, a wektor *Przydzielone_i* jest zdefiniowany tak jak w p. 7.5.3? Odpowiedź uzasadnij.

Uwagi bibliograficzne

Dijkstra [110] był jednym z pierwszych i najbardziej wpływowych uczestników badań nad zakleszczeniami. Holt w artykule [177] jako pierwszy sformalizował pojęcie zakleszczenia za pomocą modelu w teorii grafów, podobnego do zaprezentowanego w tym rozdziale. Holt zajął się również zagadni-

niem głodzenia [177]. Przykład zakleszczenia powodowanego przez przepis prawny ze stanu Kansas pochodzi z książki Hymana [184].

Różne algorytmy zapobiegania zakleszczeniom podał w artykule [166] Havender, który zaplanował schemat uporządkowania zasobów w systemie IBM OS/360.

Algorytm bankiera unikania zakleszczeń dla zasobu jednego typu opracował Dijkstra [110]. Habermann w artykule [159] rozszerzył ten algorytm na zasoby wielu typów. Ogólne omówienie dotyczące unikania zakleszczeń za pomocą deklarowania wymagań można znaleźć w pracach Habermanna [159], Holta [176, 177] oraz Parnasa i Habermanna [323]. Ćwiczenia 7.8 i 7.9 pochodzą z pracy Holta [176].

Przedstawiony w p. 7.6.2 algorytm wykrywania zakleszczenia przy wielu egzemplarzach zasobów poszczególnych typów został opisany przez Coffmana i in. [77]. Łączone metody postępowania z zakleszczeniami, opisane w p. 7.8, zaproponował Howard w artykule [180].

Ogólne przeglądy i użyteczną bibliografię zaproponowali: Isloor i Marsland [195], Newton [309] i Zobel [450].



Część 3

ZARZĄDZANIE ZASOBAMI PAMIĘCI

Podstawowym zadaniem systemu komputerowego jest wykonywanie programów. Podczas wykonywania programy wraz z niezbędnymi im danymi muszą znajdować się w pamięci operacyjnej (przynajmniej częściowo).

Aby polepszyć wykorzystanie jednostki centralnej, jak również szybkość odpowiadania użytkownikom, komputer musi przechowywać w pamięci pewną liczbę procesów. Istnieje wiele różnych metod zarządzania pamięcią. Odzwierciedlają one różnorodne podejścia do zarządzania pamięcią, a efektywność różnych algorytmów zależy od konkretnej sytuacji. Wybór schematu zarządzania pamięcią dla konkretnego systemu zależy od wielu czynników, zwłaszcza od sprzętowych właściwości systemu. Każdy algorytm wymaga specyficznych środków sprzętowych.

Ponieważ pamięć operacyjna jest zwykle za mała, aby pomieścić na stałe wszystkie dane i programy, system komputerowy musi rozporządzać pamięcią pomocniczą, uzupełniającą pamięć operacyjną. W większości współczesnych systemów komputerowych jako podstawowego, bezpośrednio dostępnego nośnika masowej informacji (zarówno programów, jak i danych) używa się dysków. System plików dostarcza mechanizmu pamięci dostępnej bezpośrednio oraz umożliwia kontakt zarówno z przechowywanymi na dysku danymi, jak i programami. Plik jest zbiorem powiązanych ze sobą informacji określonych przez jego twórcę. Pliki są przez system operacyjny odwzorowywane na urządzeniach fizycznych. W celu ułatwienia korzystania z plików zazwyczaj organizuje się je w katalogi.

Information and decision making in conservation

Edited by J. R. G. Turner and D. M. G. Toman

With contributions from C. A. Hart, D. H. Johnson, J. L. Karr, J. L. Karr,

J. R. G. Turner, D. M. G. Toman, and others

Published for the International Institute for Environment and Development

by John Wiley & Sons Ltd., Chichester, UK, and New York, USA

© 1990 John Wiley & Sons Ltd. All rights reserved. No part of this publication

may be reproduced, stored in a retrieval system or transmitted in any form or by any

means, electronic, mechanical, photocopying, recording or otherwise, without the

prior permission of the publisher, or a licence issued by the Copyright Clearance

Center, 27 Congress Street, Salem, MA 01970, USA, or by similar agency in the rest

of the world. Requests to photocopy material for internal or personal use under

circumstances not falling within the fair dealing provisions of the copyright

laws of your country should be addressed to the Copyright Clearance Center, 27

Congress Street, Salem, MA 01970, USA. Special requests should be addressed to

the publisher, John Wiley & Sons Ltd., Chichester, UK.

ISBN 0 471 92720 2
0 471 92720 2
0 471 92720 2

Printed in Great Britain by Clarendon Press, Oxford

Typeset by Laserline Typesetting, Chichester, UK

Illustrations by David W. Smith, Chichester, UK

Printed in the UK by Clarendon Press, Oxford

Rozdział 8

ZARZĄDZANIE PAMIĘCIĄ

W rozdziale 5 mówiliśmy, w jaki sposób jednostka centralna może być współdzielona przez zbiór procesów. W wyniku planowania przydziału procesora można zarówno zwiększyć jego wykorzystanie, jak i skrócić czas udzielania przez komputer odpowiedzi użytkownikom. W celu urzeczywistnienia tego wzrostu wydajności należy jednak w pamięci operacyjnej umieszczać kilka procesów naraz – musimy *dzielić* pamięć.

W niniejszym rozdziale opiszymy różne sposoby zarządzania pamięcią. Jest wiele różnorodnych algorytmów zarządzania pamięcią – poczynając od elementarnych podejść stosowanych dla „gołej” maszyny aż po strategie stronicowania i segmentacji. Każde podejście ma swoje zalety i wady. Wybór metody zarządzania pamięcią dla specyficznego systemu zależy od wielu czynników, ze szczególnym uwzględnieniem cech sprzętu. Przekonamy się, że wiele algorytmów wymaga wsparcia ze strony sprzętu.

8.1 ■ Podstawy

Pamięć ma zasadnicze znaczenie dla działania nowoczesnego systemu komputerowego, co pokazaliśmy w rozdz. 1. Pamięć jest wielką tablicą oznaczonych adresami słów lub bajtów. Jednostka centralna pobiera rozkazy z pamięci stosownie do wartości licznika rozkazów. Rozkazy te mogą powodować dodatkowe operacje pobrania i przechowania odnoszące się do określonych adresów.

Na przykład typowy cykl wykonania rozkazu zaczyna się od pobrania rozkazu z pamięci. Następnie rozkaz jest dekodowany i może wymagać po-

brania z pamięci argumentów. Po wykonaniu rozkazu na argumentach wyniki mogą zostać ponownie przechowane w pamięci. Warto zwrócić uwagę, że do jednostki pamięci dociera tylko strumień adresów pamięci; nie dochodzą do niej informacje o sposobie, w jaki adresy te zostały wytworzone (licznik rozkazów, indeksowanie, adresowanie pośrednie, adresy podane wprost w rozkazie itp.), ani o tym, czego dotyczą (rozkazów czy danych). Wobec tego będziemy pomijać *sposób* generowania adresu przez program. Będzie nas interesować wyłącznie ciąg adresów wytwarzany przez wykonywany program.

8.1.1 Wiązanie adresów

Program na ogół rezyduje na dysku jako binarny, wykonywalny plik. Aby nastąpiło wykonanie programu, należy go wprowadzić do pamięci operacyjnej i zaliczyć do odpowiadającego mu procesu. Jeśli sposób zarządzania pamięcią na to pozwala, to wykonywany proces może być przemieszczany między dyskiem a pamięcią operacyjną. Zbiór procesów czekających na dysku na wprowadzenie do pamięci w celu wykonania tworzy kolejkę wejściową (ang. *input queue*).

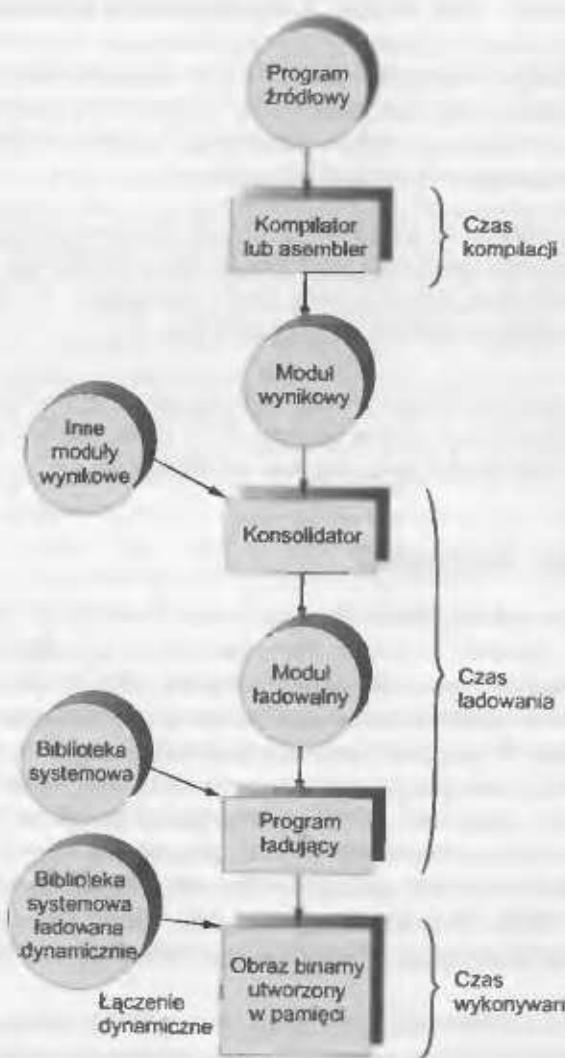
W toku normalnego postępowania jeden z procesów zostaje wybrany i załadowany do pamięci. Podczas wykonywania proces pobiera rozkazy i dane z pamięci. Po pewnym czasie proces kończy działanie i zajmowana przez niego pamięć staje się ponownie dostępna.

Większość systemów pozwala procesowi użytkowemu przebywać w dowolnej części pamięci fizycznej. Toteż, choć przestrzeń adresów komputera zaczyna się od 00000, pierwszy adres procesu użytkownika nie musi wynosić 00000. Wpływa to na zakres adresów dostępnych dla programu użytkownika. W większości przypadków program użytkownika, zanim zostanie wykonany, przechodzi przez kilka faz (niektóre z nich mogą nie występować), pokazanych na rys. 8.1. Podczas tych faz reprezentacja adresów może ulegać zmianie. W programie źródłowym adresy są wyrażone w sposób symboliczny (np. LICZNIK). Kompilator na ogół wiąże te adresy z adresami względnymi (w rodzaju: „14 bajtów, licząc od początku danego modułu”). Konsolidator lub program ładujący powiąże dalej te adresy względne¹ z adresami bezwzględnymi (np. 74014). Każde wiązanie jest odwzorowaniem z jednej przestrzeni adresowej na inną.

Powiązanie rozkazów i danych z adresami pamięci może w zasadzie zostać wykonane w dowolnym kroku poniższego ciągu działań:

- **Czas komplikacji:** Jeśli podczas komplikacji jest znane miejsce, w którym proces będzie przebywał w pamięci, to można wygenerować kod bez-

¹ Nazywane także *relokowalnymi*. – Przyp. tłum.



Rys. 8.1 Więcstopowe przetwarzanie programu użytkownika

względny (ang. *absolute code*). Jeśli na przykład z góry wiadomo, że proces użytkownika rozpoczyna się od adresu R , to wytworzony przez kompilator kod może operować adresami, poczynając od tego miejsca w góre. Gdy w późniejszym czasie ten adres początkowy ulegnie zmianie, wówczas kod taki trzeba będzie skompilować od nowa. W systemie MS-DOS pliki typu .COM zawierają programy z adresami bezwzględnymi ustalonymi podczas kompilacji.

- **Czas ładowania:** Jeśli podczas komplikacji nie wiadomo, gdzie będzie umieszczony proces w pamięci, to kompilator musi tworzyć *kod przemieszczalny* (ang. *relocatable code*). W tym przypadku ostateczne wiązanie jest opóźnione do czasu ładowania. Jeśli adres początkowy ulegnie zmianie, to wystarczy tylko załadować ponownie kod użytkowy z uwzględnieniem nowej wartości tego adresu.
- **Czas wykonania:** Jeśli proces może ulegać przemieszczeniom z jednego miejsca w pamięci do innego podczas swojego wykonania, to trzeba czekać z wiązaniem adresów aż do czasu wykonania. Wymaga to zastosowania specjalnego sprzętu, co opiszymy w p. 8.2.

Większa część tego rozdziału służy przedstawieniu sposobów, za pomocą których owe różnorodne wiązania można efektywnie realizować w systemie komputerowym, oraz omówieniu zaplecza sprzętowego.

8.1.2 Ładowanie dynamiczne

W celu lepszego wykorzystania obszaru pamięci stosuje się *ładowanie dynamiczne* (ang. *dynamic loading*). Przy ładowaniu dynamicznym podprogram nie jest wprowadzany do pamięci dopóty, dopóki nie zostanie wywołany. Wszystkie podprogramy są w postaci przemieszczalnej przechowywane na dysku. Do pamięci wprowadza się program główny i tam jest on wykonywany. Gdy jakiś podprogram chce wywołać inny podprogram, musi wówczas najpierw sprawdzić, czy ów podprogram znajduje się w pamięci. Jeśli go tam nie ma, to trzeba wywoływać program łączący i ładujący moduły przemieszczalne, który wprowadzi do pamięci potrzebny podprogram oraz uaktualni tablicę adresów programu, aby odzwierciedlić tą zmianę. Następuje wtedy przekazanie sterowania do nowo załadowanego podprogramu.

Zaletą ładowania dynamicznego jest to, że nigdy nie zostanie załadowany podprogram, którego się nie używa. Schemat ten jest szczególnie przydatny wtedy, kiedy okazjonalnie trzeba wykonać wielkie fragmenty kodu, takie jak podprogramy obsługi błędów. W tym przypadku, pomimo że ogólny rozmiar programu może być duży, używana bieżąco porcja (a więc i załadowana) może być o wiele mniejsza.

Ładowanie dynamiczne nie wymaga specjalnego wsparcia ze strony systemu operacyjnego. To użytkownicy są odpowiedzialni za takie zaprojektowanie programów, aby mogły one korzystać z tej metody. Systemy operacyjne mogą jednak pomagać programistom, dostarczając procedur bibliotecznych do realizowania ładowania dynamicznego.

8.1.3 Konsolidacja dynamiczna

Zauważmy, że na rys. 8.1 są zaznaczone również biblioteki *przyłączane dynamicznie* (ang. *dynamic linked libraries*). Większość systemów operacyjnych umożliwia tylko statyczną konsolidację, w której systemowe biblioteki języków programowania są traktowane jak każdy inny moduł wynikowy* i dołączane przez program ładowający do binarnego obrazu programu. Pomyśl dynamicznej konsolidacji jest podobny do ładowania dynamicznego. Zamiast odwlekania ładowania do czasu wykonania, opóźnia się konsolidację. Cechą ta zwykle dotyczy bibliotek systemowych, na przykład bibliotek języków programowania. Jeśli system nie ma tej właściwości, to wszystkie programy muszą mieć dołączoną do swoich obrazów binarnych kopię biblioteki języka (lub przynajmniej kopie podprogramów, do których się odwołują). Powoduje to marnotrawstwo zarówno przestrzeni dyskowej, jak i obszaru pamięci operacyjnej. W przypadku konsolidacji dynamicznej, w obrazie binarnym, w miejscu odwołania bibliotecznego znajduje się tylko namiastka procedury (ang. «*stub*»)**. Namiastka procedury jest małym fragmentem kodu, wskazującym jak odnaleźć odpowiedni, rezydujący w pamięci podprogram biblioteczny lub jak załadować bibliotekę, jeśli podprogramu nie ma w pamięci.

Wykonanie namiastki procedury powoduje sprawdzenie, czy potrzebny podprogram znajduje się już w pamięci. Jeśli podprogramu nie ma w pamięci, to zostanie on przez program do niej sprowadzony. W każdym przypadku namiastka procedury wprowadza na swoje miejsce adres potrzebnego podprogramu i powoduje jego wykonanie. Dzięki temu, gdy po raz kolejny stworzenie dojdzie do danego fragmentu kodu, wówczas podprogram biblioteczny zostanie wykonyany bezpośrednio, bez ponoszenia kosztów na jego dynamiczne dołączanie. W tym schemacie wszystkie procesy korzystające z biblioteki języka programowania wykonują tylko jedną kopię kodu bibliotecznego.

Cechę tę może rozszerzyć na aktualizację bibliotek (np. po zlokalizowaniu błędu). Biblioteka może zostać zastąpiona nową wersją i wszystkie odwołujące się do niej programy będą automatycznie używały nowej wersji. Bez dynamicznej konsolidacji wszystkie takie programy w celu uzyskania dostępu do nowej biblioteki musiałyby zostać skonsolidowane na nowo. Aby programy nie mogły przypadkowo korzystać z nowych – niezgodnych z dotychczasowymi – wersji bibliotek, informację o wersji dołącza się zarówno do pro-

* Wynik pracy kompilatora. – Przyp. tłum.

** W danym kontekście można by też mówić o „zakładce” zastępującej tymczasowo kod podprogramu bibliotecznego. – Przyp. tłum.

gramu, jak i do biblioteki. Do pamięci można załadować więcej niż jedną wersję biblioteki, każdy zaś program posłuży się swoją informacją o wersji, aby wybrać właściwą bibliotekę. Przy niewielkich zmianach zachowuje się numer wersji, natomiast ważniejsze zmiany powodują jego zwiększenie. Zatem niezgodności powodowane zmianami w bibliotece uwidaczniają się tylko w programach skompilowanych z jej nowym numerem wersji. Inne programy, skonsolidowane przed zainstalowaniem nowej biblioteki, będą nadal wykonywane przy użyciu starej biblioteki. System tego rodzaju bywa nazywany *bibliotekami dzielonymi* (ang. *shared libraries*).

W odróżnieniu od ładowania dynamicznego konsolidacja dynamiczna wymaga na ogół pomocy ze strony systemu operacyjnego. Jeżeli procesy w pamięci są chronione przed sobą wzajemnie (p. 8.4.1), to tylko system operacyjny może sprawdzać, czy potrzebny podprogram znajduje się w obszarach pamięci innych procesów oraz zezwalać, aby wiele procesów miało dostęp do tych samych adresów pamięci. Pomyśl ten ulega poszerzeniu w połączeniu ze stronicowaniem, co jest omówione w p. 8.5.5.

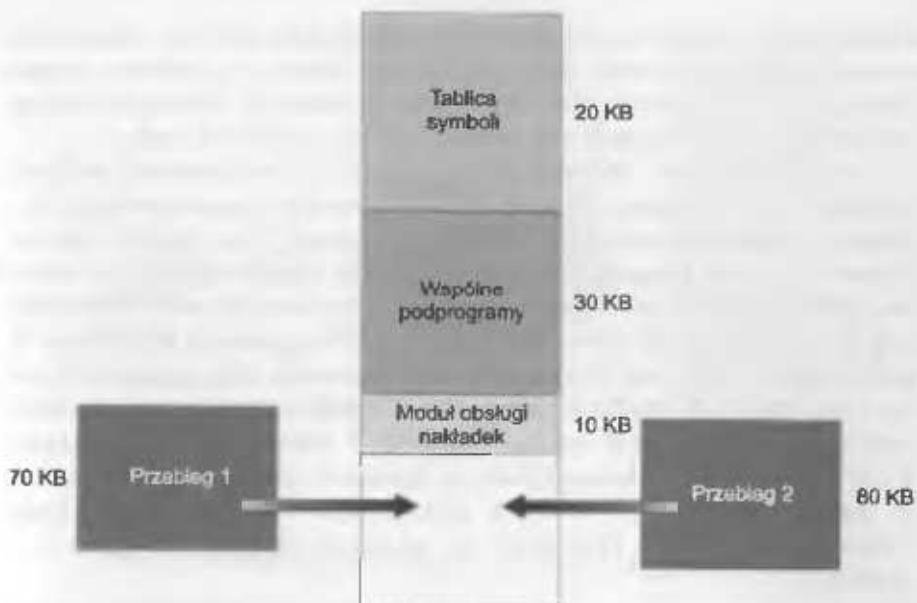
8.1.4 Nakładki

Z tego, co powiedzieliśmy dotychczas, wynikało, że cały program i dane procesu muszą w celu jego wykonania mieścić się w pamięci operacyjnej. Rozmiar procesu jest ograniczony do wielkości pamięci fizycznej. Niektóre, aby umożliwić zwiększenie wymiarów procesu ponad ilość przydzielonej mu pamięci, stosuje się technikę zwaną *nakładkami* (ang. *overlays*). Idea nakładek polega na przechowywaniu w pamięci tylko tych rozkazów i danych, które są stale potrzebne. Inne rozkazy są wprowadzane w miarę zapotrzebowania na miejsca zajmowane uprzednio przez rozkazy już zbyteczne.

Rozważmy na przykład dwuprzebiegowy asembler. W pierwszym przebiegu konstruuje on tablicę symboli, w drugim przebiegu generuje kod maszynowy. Asembler taki można by podzielić na kod przebiegu 1, kod przebiegu 2, tablicę symboli i wspólne podprogramy, wspierające działanie obu przebiegów. Założymy, że rozmiary tych składowych są następujące (KB oznacza „kilobajt”, czyli 1024 bajty):

Kod przebiegu 1	70 KB
Kod przebiegu 2	80 KB
Tablica symboli	20 KB
Wspólne podprogramy	30 KB

Do załadowania wszystkiego naraz potrzeba by było 200 KB pamięci. Jeżeli mamy do dyspozycji tylko 150 KB, to nie możemy wykonać naszego procesu.



Rys. 8.2 Nakładki dwuprzeciegowego asemblera

Zauważmy jednak, że kody przebiegu 1 i przebiegu 2 nie muszą znajdować się w pamięci w tym samym czasie. Definiujemy zatem dwie nakładki: nakładkę *A* złożoną z tablicy symboli, wspólnych podprogramów i kodu przebiegu 1 oraz nakładkę *B* złożoną z tablicy symboli, wspólnych podprogramów i kodu przebiegu 2.

Dodajemy ponadto moduł obsługi nakładek (10 KB) i rozpoczynamy od wykonania nakładki *A* w pamięci. Po zakończeniu przebiegu 1 następuje skok do modułu obsługi nakładek, który na miejsce nakładki *A* czyta do pamięci nakładkę *B*, po czym rozpoczyna przebieg 2. Nakładka *A* potrzebuje tylko 120 KB, natomiast nakładka *B* zajmuje 130 KB (rys. 8.2). Możemy w ten sposób wykonywać przebiegi naszego asemblera, mając do dyspozycji 150 KB pamięci operacyjnej. Będzie on ladowany nieco szybciej, gdyż do rozpoczęcia wykonywania trzeba przesłać mniej danych. Co prawda, będzie on działał nieco wolniej ze względu na dodatkowe operacje czytania kodu nakładki *B* do pamięci na miejsce nakładki *A*.

Kody nakładek *A* i *B* są przechowywane na dysku w postaci obrazów bezwzględnych pamięci i są czytane przez moduł obsługi nakładek w zależności od potrzeb. Do konstruowania nakładek są potrzebne specjalne algorytmy przemieszczania i konsolidacji.

Podobnie jak przy ładowaniu dynamicznym, nakładki nie wymagają specjalnego wsparcia ze strony systemu operacyjnego. Mogą być w całości wy-

konane przez użytkownika za pomocą prostej struktury plików, czytania zawartości plików do pamięci oraz wykonywania skoków w określone miejsca pamięci w celu wykonania nowo przeczytanych instrukcji. System operacyjny odnotowuje tylko większą liczbę operacji wejścia-wyjścia niż zwykle.

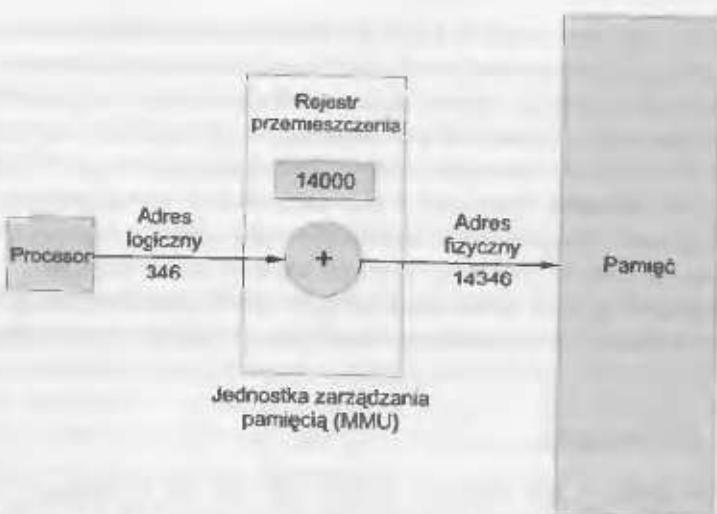
Programista musi natomiast zaprojektować i zaprogramować strukturę nakładek nader starannie. Może to oznaczać poważne przedsięwzięcie, wymagające dogłębnej znajomości budowy programu, jego kodu i struktur danych. Ponieważ program jest z definicji wielki (małe programy nie muszą być nakładkowane), zrozumienie programu w wystarczającym stopniu może więc być trudne. Z tych powodów użycie nakładek ogranicza się obecnie do mikrokomputerów i innych systemów o ograniczonej ilości pamięci fizycznej i nie mających środków sprzętowych, umożliwiających zastosowanie bardziej zaawansowanych technik. Niektóre z kompilatorów stosowanych w mikrokomputerach ułatwiają pracę programisty, zawierając środki obsługi nakładek. Godne polecenia są automatyczne techniki umożliwiające wykonywanie wielkich programów w ograniczonym obszarze pamięci fizycznej.

8.2 ■ Logiczna i fizyczna przestrzeń adresowa

Adres wytworzony przez procesor jest zazwyczaj nazywany *adresem logicznym* (ang. *logical address*), a adres oglądany przez jednostkę pamięci (tj. ten, który zostaje umieszczony w jej *rejestrze adresowym*) na ogół zwie się *adresem fizycznym* (ang. *physical address*).

Schematy ustalania adresów podczas komplikacji oraz ładowania tworzą środowisko, w którym adresy logiczne i fizyczne są takie same. Z kolei schematy wiązania adresów podczas wykonywania rozkazów prowadzą do środowiska, w którym adresy logiczne i fizyczne są różne. W tym przypadku często określamy adres logiczny mianem *adresu wirtualnego* (ang. *virtual address*). W tym tekście terminy „adres logiczny” i „adres wirtualny” są używane zamiennie. Zbiór wszystkich adresów logicznych generowanych przez program jest nazywany *logiczną przestrzenią adresową* (ang. *logical address space*). Zbiór wszystkich adresów fizycznych odpowiadających tym adresom logicznym nazywa się *fizyczną przestrzenią adresową* (ang. *physical address space*). Tak więc adresy logiczne i fizyczne powstające wskutek wiązania adresów podczas wykonywania procesu różnią się.

Odwzorowywanie adresów wirtualnych na fizyczne, odbywające się podczas działania programu, jest dokonywane przez *jednostkę zarządzania pamięcią* (ang. *memory-management unit* – MMU), będącą urządzeniem sprzę-



Rys. 8.3 Przemieszczanie dynamiczne z użyciem rejestru przemieszczenia

towym. Istnieje kilka różnych sposobów wykonywania takiego odwzorowania, co zostanie omówione w p. 8.4.1, 8.5, 8.6 i 8.7. Na razie zilustrujemy to odwzorowywanie za pomocą schematu działania prostego urządzenia MMU, będącego uogólnieniem schematu rejestru bazowego opisanego w p. 2.4.

Jak widać na rys. 8.3, schemat ten wymaga nieco innej pomocy ze strony sprzętu niż to omówiliśmy w p. 2.4. Rejestr bazowy nazywa się teraz *rejestrem przemieszczenia** (ang. *relocation register*). Wartość z rejestru przemieszczenia jest dodawana do każdego adresu wytwarzanego przez proces użytkownika w chwili odwoływania się do pamięci. Jeśli na przykład baza wynosi 14 000, to gdy użytkownik próbuje zaadresować komórkę 0, wówczas wartość tego adresu jest dynamicznie zmieniana na odwołanie do komórki 14 000; odwołanie do komórki 346 przemieszcza się do komórki 14 346. System operacyjny MS-DOS, pracujący na procesorach rodziny Intel 80X86, używa do ładowania i wykonywania procesów czterech rejestrów przemieszczenia.

Zwróciły uwagę, że program użytkownika nigdy nie ma do czynienia z rzeczywistymi adresami fizycznymi. Program może utworzyć wskaźnik do komórki 346, zapamiętać go w pamięci, działać na nim, porównywać go z innymi adresami – za każdym razem w postaci liczby 346. Przemieszczeniu względem rejestru bazowego ulegnie on tylko wtedy, gdy zostanie użyty jako adres pamięci (np. w operacji pośredniego pobrania lub przechowania). Program użytkownika działa na adresach *logicznych*. Sprzęt odwzorowujący pamięć zamienia adresy logiczne na adresy fizyczne. Ustalanie adresu podczas

* Albo rejestrem relokacji. – Przyp. tłum.

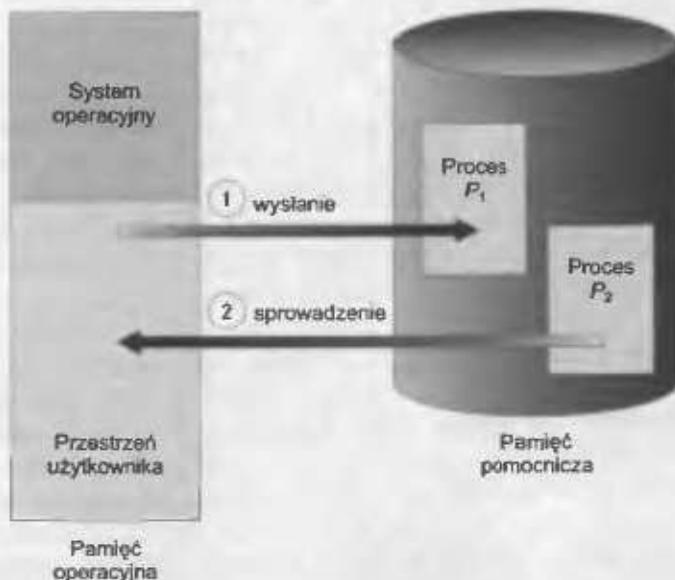
wykonywania jest omówione w p. 8.1.1. Ostateczna komórka, do której odnosi się adres pamięci, jest nieokreślona do chwili wykonania tego odwołania.

Zauważmy, że mamy obecnie do czynienia z dwoma rodzajami adresów: adresami logicznymi (z przedziału od 0 do max) i adresami fizycznymi (z przedziału od $R + 0$ do $R + max$, przy czym R jest wartością bazy). Użytkownik operuje tylko adresami logicznymi i ma wrażenie, że proces działa w komórkach od 0 do max . Program użytkownika dostarcza adresów logicznych; zanim adresy te zostaną użyte, muszą być odwzorowane na adresy fizyczne.

Koncepcja logicznej przestrzeni adresów powiązanej z odrębną, fizyczną przestrzenią adresów jest podstawą właściwego zarządzania pamięcią.

8.3 ■ Wymiana

Wykonanie procesu jest możliwe wtedy, gdy jest on w pamięci. Jednakże proces może być tymczasowo *wymieniany* (ang. *swapped*), tj. odsyłany z pamięci operacyjnej do pamięci pomocniczej i z powrotem – w celu kontynuowania działania. Założymy na przykład, że mamy do czynienia z wieloprogramowym środowiskiem sterowanym rotacyjnym algorytmem planowania przydziału procesora. Po wyczerpaniu kwantu czasu zarządcą pamięci rozpoczyna wymianę procesu, który na razie zakończył działanie, na proces, który zajmie zwolnione miejsce w pamięci (rys. 8.4). W międzyczasie planista



Rys. 8.4 Wymiana dwóch procesów z użyciem dysku jako pamięci pomocniczej

przydzielu procesora przydzieli kwant czasu innemu procesowi w pamięci. Każdy proces po zużyciu kwantu czasu zostanie wymieniony z innym procesem. W idealnych warunkach zarządcą pamięci może wymieniać procesy wystarczająco szybko, aby w pamięci były zawsze procesy gotowe do wykonania wtedy, kiedy planista przydzielu procesora zechce dokonać kolejnego jego przydziału. Kwant czasu musi być także dostatecznie duży, aby między kolejnymi wymianami można było wykonać sensowną porcję obliczeń.

Pewnego wariantu polityki wymiany używa się w algorytmach planowania priorytetowego. Jeżeli nadjdzie proces o wyższym priorytecie i zaząda obsługi, to zarządcą pamięci może z niej usunąć proces o niższym priorytecie, aby móc załadować i wykonać proces o wyższym priorytecie. Gdy proces o wyższym priorytecie skończy działanie, wówczas proces o niższym priorytecie może być z powrotem sprowadzony do pamięci i kontynuowany. Ten wariant wymiany jest niekiedy nazywany *wylaczaniem* (ang. *roll out*) i *wstawianiem* (ang. *roll in*).

Zazwyczaj proces, który ulega wymianie, powraca do pamięci w to samo miejsce, w którym przebywał uprzednio. Ograniczenie to jest podkutowane metodą wiązania adresów. Jeśli wiązanie jest wykonywane podczas tłumaczenia lub ładowania, to proces nie może być przesunięty w inne miejsce. Jeśli adresy ustala się podczas wykonania, to istnieje możliwość sprowadzenia procesu do innego obszaru pamięci, ponieważ adresy fizyczne są obliczane na bieżąco.

Du wymiany jest potrzebna *pamięć pomocnicza* (ang. *backing store*). Jest nią na ogół szybki dysk. Musi on być wystarczająco pojemny, aby pomieścić wszystkie kopie obrazów pamięci wszystkich użytkowników. Powinien także umożliwiać bezpośredni dostęp do tych obrazów pamięci. System utrzymuje *kolejkę procesów gotowych* (ang. *ready queue*), składającą się ze wszystkich procesów, których obrazy pamięci są w pamięci pomocniczej lub operacyjnej i które są gotowe do działania. Ilekroć planista przydziału procesora decyduje się wykonać proces, tylekroć wywołuje ekspedytora. Ekspedytor (ang. *dispatcher*) sprawdza, czy następny proces z kolejki jest w pamięci operacyjnej. Jeśli procesu tam nie ma i nie ma wolnego obszaru pamięci, to ekspedytor odsyła na dysk któryś z procesów przebywających w pamięci i na jego miejsce wprowadza potrzebny proces. Następnie jak zwykle aktualnia stany rejestrów i przekazuje sterowanie do wybranego procesu.

Jest zrozumiałe, że czas przełączania kontekstu w systemie, w którym stosuje się wymianę, jest dość długi. Aby wyrobić sobie pogląd o czasie przełączania kontekstu, założmy, że proces użytkownika ma rozmiar 100 KB oraz że pamięć pomocniczą stanowi standardowy dysk twardy o szybkości

przesyłania 1 MB na sekundę. Przesłanie 100 KB kodu procesu do lub z pamięci operacyjnej zabierze wówczas

$$100 \text{ KB} / 1000 \text{ KB/s} = 1/10 \text{ s} = 100 \text{ ms}$$

Zakładając, że nie trzeba przemieszczać głowic dysku oraz że średni czas wykrywania sektora wynosi 8 ms, czas wymiany wyniesie 108 ms. Ponieważ należy przesyłać w obie strony, całkowity czas wymiany wyniesie 216 ms.

Aby wykorzystanie procesora było wydajne, należy zadbać o względnie długi czas wykonania każdego procesu w stosunku do czasu wymiany. Na przykład przy planowaniu przydziału procesora za pomocą algorytmu rotacyjnego kwant czasu powinien być, wobec poprzednich wyliczeń, znacznie większy niż 0,216 s.

Zauważmy, że główną częścią czasu wymiany jest czas przesyłania. Łączny czas przesyłania jest wprost proporcjonalny do ilości wymienianej pamięci. Jeśli dysponujemy komputerem z 1 MB pamięci operacyjnej i rezydującym w niej systemem operacyjnym, zajmującym 100 KB, to maksymalny rozmiar procesu użytkownika wyniesie 900 KB. Jednak wiele procesów użytkowych może zajmować znacznie mniej miejsca w pamięci – powiedzmy, mogą zajmować po 100 KB. Wymiany procesu o rozmiarze 100 KB można dokonać w 108 ms, co kontrastuje z wartością 908 ms dla procesu o rozmiarze 900 KB. Toteż warto wiedzieć dokładnie, ile pamięci zajmuje proces użytkownika, a nie tylko ile może jej zajmować. Należy wówczas wymieniać tylko to, co jest aktualnie używane – skracając czas wymiany. Aby schemat ten był efektywny, użytkownik musi informować system o każdej zmianie zapotrzebowania na pamięć. W procesie z dynamiczną gospodarką pamięcią trzeba będzie zatem korzystać z funkcji systemowych służących do *zamawiania pamięci i zwalniania pamięci* w celu informowania systemu operacyjnego o zmieniających się potrzebach.

Z wymianą są związane również inne ograniczenia. Jeśli chcemy dokonać wymiany procesu, to musimy mieć pewność, że proces jest zupełnie bezczynny. Szczególną uwagę należy zwrócić na trwające operacje wejścia-wyjścia. Jeśli proces czeka z powodu operacji wejścia-wyjścia, to możemy mieć ochotę usunąć go z pamięci, aby zyskać wolne miejsce. Jednak jeśli operacja wejścia-wyjścia ma asynchroniczny dostęp do buforów w pamięci użytkownika, to proces nie może ulec wymianie. Założymy, że operacja wejścia-wyjścia została ustawiona w kolejce, ponieważ urządzenie jest zajęte. Wówczas, gdyby wysłano z pamięci proces P_1 i zastąpiono go obrazem procesu P_2 , operacja wejścia-wyjścia mogłaby usiłować użyć pamięci należącej obecnie do procesu P_2 . Istnieją dwa rozwiązania tego problemu: (1) nigdy nic wymieniać procesu, w którym trwają operacje wejścia-wyjścia. (2) wykonywać operacje wejścia-wyjścia tylko za pośrednictwem buforów systemu operacyjnego.

Przesyłanie danych między systemem operacyjnym a pamięcią procesu następuje wówczas po powtórnym wprowadzeniu procesu do pamięci.

Założenie, że wymiana pociąga za sobą niewielki ruch głowic, wymaga dalszego wyjaśnienia. Zrobimy to dopiero w rozdz. 13, który dotyczy budowy pamięci pomocniczej. Ogólnie biorąc, obszar wymiany jest przydzielany na dysku w postaci osobnej porcji pamięci, niezależnej od systemu plików, aby korzystanie z niego było możliwie jak najszyszsze.

Obecnie standardową wymianę stosuje się w niewielu systemach. Zajmuje ona zbyt wiele czasu i pozostawia za mało czasu do działania, aby można ją było uznać za rozsądne rozwiązanie zagadnienia zarządzania pamięcią. Niemniej jednak w wielu systemach można spotkać zmodyfikowane wersje algorytmu wymiany.

Zmodyfikowana metoda wymiany jest stosowana w wielu wersjach systemu UNIX. W normalnych warunkach wymiana była zabroniona, lecz mogła rozpoczęć się w sytuacji, gdy przy nagromadzeniu wielu procesów zajęta pamięć osiągała wartość progową. Wymianę wstrzymywano wówczas, gdy załadowanie systemu zostało zmniejszone. Zarządzanie pamięcią w systemie UNIX jest opisane w p. 21.6.

Pierwsze komputery osobiste nie były wyposażone w odpowiedni sprzęt, który umożliwiałby implementację bardziej zaawansowanych metod zarządzania pamięcią (nie miały też systemów operacyjnych, które robiłyby taki użytek z właściwości sprzętu). Stosowano je natomiast do wykonywania wielu wielkich procesów za pomocą zmodyfikowanej wersji wymiany. Pierwszym przykładem jest system operacyjny Microsoft Windows 3.1, który umożliwia współbieżne wykonywanie procesów w pamięci. Jeśli do załadowania nowego procesu brakuje miejsca w pamięci głównej, to któryś ze starych procesów zostaje wysłany na dysk. Jednakże ten system operacyjny nie zapewnia pełnej wymiany, ponieważ to użytkownik, a nie planista, rozstrzyga o tym, kiedy należy wywalać jeden proces na rzecz drugiego. Każdy proces usunięty na dysk pozostaje tam (i nie jest wykonywany) do chwili, w której użytkownik wybierze go do działania. Następne systemy operacyjne rodem z Microsoft, na przykład system Windows NT, robią użytek z zaawansowanych możliwości sprzętowych jednostek zarządzania pamięcią, które dziś spotyka się nawet w komputerach osobistych. W punkcie 8.7.2 jest opisany sprzęt do zarządzania pamięcią występujący w rodzinie procesorów Intel 386 stosowany w wielu komputerach osobistych. Jest tam również omówione zarządzanie pamięcią na tym procesorze wykonywane przez inny, rozwinięty system operacyjny dla komputerów PC, tj. przez system IBM OS/2.

8.4 ■ Przydział ciągły

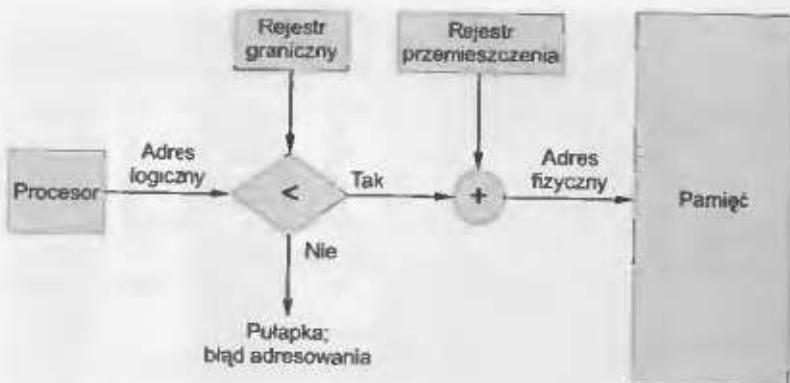
Pamięć operacyjna (główna) musi pomieścić zarówno system operacyjny, jak i rozmaite procesy użytkownika. Jest ona zazwyczaj podzielona na dwie części – jedną dla rezydującego systemu operacyjnego i drugą dla procesów użytkownika. System operacyjny można umieścić w pamięci dolnej albo w pamięci górnej. Na decyzję w tej sprawie wpływa głównie lokalizacja wektora przerwań. Ponieważ wektor przerwań znajduje się często w pamięci dolnej, powszechniejszym rozwiązaniem jest umieszczenie systemu operacyjnego w pamięci dolnej. Z tego powodu rozważać będziemy tylko sytuację, w której system operacyjny rezyduje w pamięci dolnej (rys. 8.5). Postępowanie w drugim przypadku jest podobne.

8.4.1 Przydział pojedynczego obszaru

Jeśli system operacyjny pozostaje w pamięci dolnej, a proces użytkownika wykonuje się w pamięci górnej, to powstaje potrzeba ochrony kodu i danych systemu operacyjnego przed zmianami (przypadkowymi lub złośliwymi), które może spowodować proces użytkownika. Procesy użytkowników należy także chronić wzajemnie przed sobą. Ochronę pamięci można osiągnąć przez użycie rejestru przemieszczenia, omówionego w p. 8.2, w połączeniu z rejestrzem granicznym, opisany w p. 2.5.3. Rejestr przemieszczenia zawiera wartość najmniejszego adresu fizycznego: rejestr graniczny zawiera zakres adresów logicznych (np. przemieszczenie = 100 040, ograniczenie = 74 600). Gdy korzysta



Rys. 8.5 Podział pamięci



Rys. 8.6 Zastosowanie rejestrów sprzętowych: przemieszczenia i granicznego

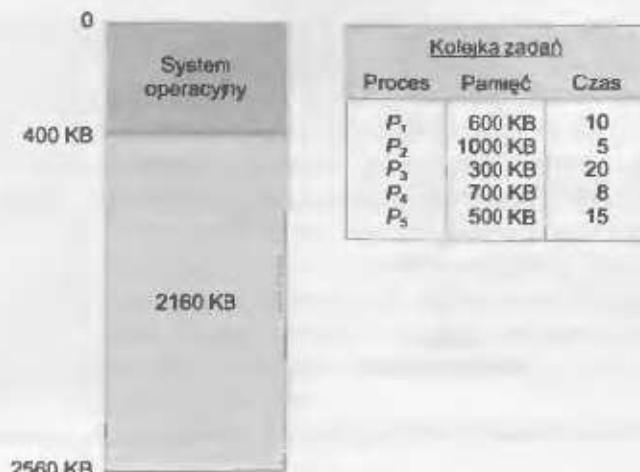
się z tych rejestrów, wówczas każdy adres logiczny musi być mniejszy od wartości rejestru granicznego. Jednostka zarządzania pamięcią przekształca dynamicznie adres logiczny przez dodanie do niego wartości z rejestru przemieszczenia. Odwzorowany w ten sposób adres dociera do pamięci (rys. 8.6).

Gdy planista przydziału procesora wybierze proces do wykonania, wówczas ekspedytor ładuje odpowiednie wartości do rejestru przemieszczenia i do rejestru granicznego, co jest częścią przełączania kontekstu. Ponieważ każdy adres wygenerowany przez procesor jest porównywany z zawartością tych rejestrów, możemy w ten sposób chronić zarówno system operacyjny, jak i programy oraz dane innych użytkowników przed zmienianiem przez bieżący proces.

Zwrócić uwagę na to, że posługiwanie się rejestrów przemieszczenia pozwala skutecznie zmieniać na bieżąco rozmiar systemu operacyjnego. Elastyczność taka jest pożądana w wielu sytuacjach. System operacyjny zawiera na przykład kod i obszar buforów modułów sterujących urządzeń. Jeśli moduł sterujący urządzeniami (lub inna usługa systemu operacyjnego) nie jest w częstym użyciu, to utrzymywanie jego kodu i danych w pamięci operacyjnej nie jest wskazane, gdyż zajmowany przez niego obszar można by wykorzystać do innych celów. Kod taki nazywa się niekiedy kodem przejściowym (ang. *transient*) systemu operacyjnego – pojawia się i znika stosownie do potrzeb. Korzystanie z takiego kodu powoduje zatem zmianę rozmiaru systemu operacyjnego podczas wykonywania programu.

8.4.2 Przydzielanie wielu obszarów

Ponieważ na ogół jest pożądane, aby w pamięci pozostawało w tym samym czasie kilka procesów użytkowych, musimy rozważyć problem przydzielania pamięci różnym procesom oczekującym w kolejce wejściowej na wprowa-

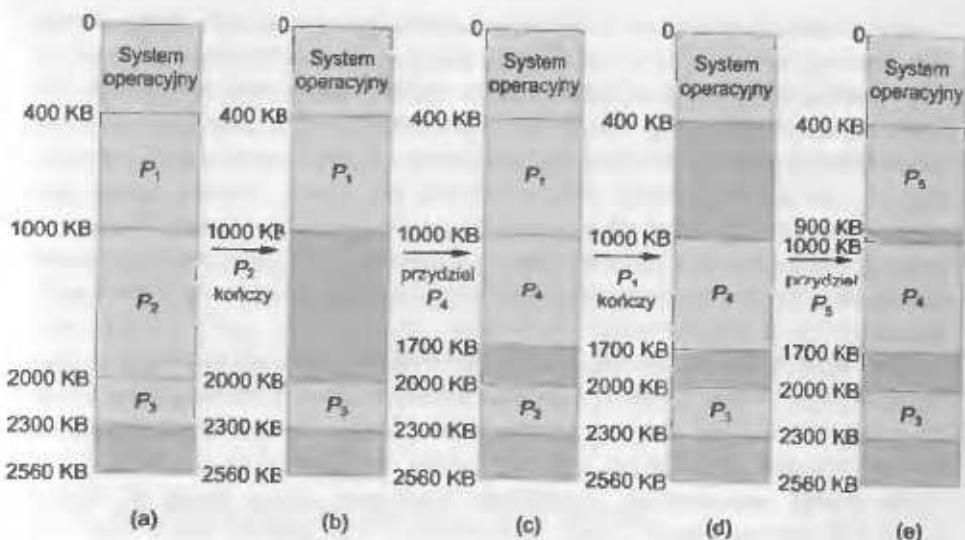


Rys. 8.7 Przykład planowania

dzenie do pamięci operacyjnej. Jednym z najprostszych schematów przydziału pamięci jest podzielenie jej na pewną liczbę obszarów (ang. *partitions*) o stałym rozmiarze. Każdy obszar może zawierać dokładnie jeden proces. Zatem stopień wieloprogramowości jest ograniczony przez liczbę obszarów. Kiedy powstaje wolny obszar, wtedy następuje wybranie jakiegoś procesu z kolejki wejściowej i wprowadzenie go do tego obszaru. Gdy proces kończy działanie, wówczas zajmowany przez niego obszar zwalnia się dla innego procesu. Schemat taki został po raz pierwszy zastosowany w systemie operacyjnym IBM OS/360 (nazwano go MFT); obecnie nie jest już używany. Schemat omówiony dalej (nazywany MVT)* jest uogólnieniem schematu ze stałą liczbą obszarów i jest używany przede wszystkim w środowisku wsadowym. Zauważmy jednak, że wiele z pomysłów, które są tu przedstawione, można stosować także w środowisku z podziałem czasu, jeśli do zarządzania pamięcią używa się czystej segmentacji (p. 8.6).

System operacyjny przechowuje tablicę z informacjami o tym, które części pamięci są dostępne, a które zajęte. Na początku cała pamięć jest dostępna dla procesów użytkowych i jest traktowana jako jeden wielki blok pamięci – *dziura* (ang. *hole*). Gdy przybywa proces z zapotrzebowaniem na pamięć, wówczas poszukuje się dla niego wystarczająco obszernej dziury. Jeśli zostanie znaleziona, to przydziela się z niej pamięć tylko w niezbędną ilość, pozostawiając resztę na przyszłe potrzeby.

* Nazwy są skrótami określonych angielskich: MFT – *multiprogramming with a fixed number of tasks*; MVT – *multiprogramming with a variable number of tasks*, czyli wieloprogramowość ze stałą (odpowiednio – zmiennej) liczbą zadań. – Przyp. tłum.



Rys. 8.8 Przydzielanie pamięci i planowanie długoterminowe

Załóżmy na przykład, że mamy 2560 KB dostępnej pamięci i system operacyjny, który stale zajmuje w niej 400 KB. W tej sytuacji na procesy użytkowe pozostało 2160 KB (rys. 8.7). Mając kolejkę wejściową jak na rys. 8.7 oraz planowanie zadań metodą FCFS, możemy natychmiast przydzielić pamięć procesom P_1 , P_2 i P_3 , tworząc mapę pamięci jak na rys. 8.8(a). Pozostała dziura o wielkości 260 KB nie nadaje się dla żadnego z reszty procesów w kolejce wejściowej. Przy użyciu rotacyjnego algorytmu planowania przydziału procesora z kwantem wynoszącym 1 jednostkę czasu proces P_1 zakończy działanie w chwili 14, zwalniając przydzieloną mu pamięć. Sytuację tę pokazano na rys. 8.8(b). Powracamy wówczas do kolejki wejściowej, aby wybrać następny proces – o numerze P_4 , co spowoduje odwzorowanie pamięci jak na rys. 8.8(c). Proces P_1 skończy pracę w chwili 28, wytwarzając stan przedstawiony na rys. 8.8(d). Następnie proces P_5 otrzyma pamięć, co zobaczono na rys. 8.8(e).

Przykład ten jest ilustracją ogólnej budowy schematu przydziału. Nadechodzące do systemu procesy są umieszczane w kolejce wejściowej. Przydzielając pamięć procesom, system operacyjny uwzględnia zapotrzebowanie na pamięć każdego procesu oraz ilość wolnej pamięci. Proces, któremu przydzielono przestrzeń, jest wprowadzany do pamięci i zaczyna rywalizować o przydział procesora. Gdy proces kończy pracę, wówczas zwalnia swoją pamięć, a system operacyjny może wtedy umieścić w niej obraz innego procesu z kolejki wejściowej.

W każdej chwili jest znana lista rozmiarów dostępnych bloków oraz kolejka wejściowa. System operacyjny może porządkować kolejkę wejściową zgodnie z algorytmem planowania. Procesy otrzymują przydzielony pamięci aż do chwili, kiedy okaże się, że zapotrzebowanie na pamięć złożone przez kolejny proces nie może być spełnione. Żaden dostępny blok pamięci (dziura) nie jest dość duży, żeby pomieścić ten proces. System operacyjny może wówczas zaczekać na pojawienie się odpowiednio dużego bloku lub może przeskoczyć pozycję w kolejce wejściowej, żeby sprawdzić, czy nie da się zaspokoić mniejszego wymagania na pamięć któregoś z pozostałych procesów.

Na ogół w każdej chwili istnieje zbiór dziur o różnych rozmiarach, rozproszonych po całej pamięci¹. Gdy proces nadchodzi i zamawia pamięć, wtedy przegląda się ten zbiór w poszukiwaniu dziury wystarczająco dużej dla danego procesu. Jeśli dziura jest zbyt wielka, to dzieli się ją na dwie: jedna część zostaje przydzielona przybyłemu procesowi, druga wraca do zbioru dziur. Gdy proces kończy pracę, zwalnia swój blok pamięci, który znów zostaje umieszczony w zbiorze dziur. Jeśli nowa dziura przylega do innych dziur, to łączy się przyległe dziury, tworząc jedną, większą dziurę. Należy wówczas sprawdzić, czy jakieś procesy oczekują na pamięć oraz czy nowo odzyskana i zreorganizowana pamięć może spełnić wymagania któregoś z tych oczekujących procesów.

Postępowanie takie jest szczególnym przypadkiem ogólnego problemu *dynamicznego przydziału pamięci* (ang. *dynamic storage allocation*) polegającego na rozstrzyganiu, jak na podstawie listy wolnych dziur spełnić zamówienie na obszar o rozmiarze n . Problem ten ma wiele rozwiązań. Przegląda się zbiór dziur, aby określić, która z nich nadaje się najlepiej do przydziału. Najbardziej znane strategie wyboru wolnego obszaru ze zbioru dostępnych dziur noszą nazwy: *pierwsze dopasowanie* (ang. *first-fit*), *najlepsze dopasowanie* (ang. *best-fit*) i *najgorsze dopasowanie* (ang. *worst-fit*).

- **Pierwsze dopasowanie:** Przydziela się *pierwszą* dziurę o wystarczającej wielkości. Szukanie można rozpocząć od początku wykazu dziur lub od miejsca, w którym je ostatnio zakończono. Szukanie kończy się z chwilą napotkania dostatecznie dużej, wolnej dziury.
- **Najlepsze dopasowanie:** Przydziela się *najmniejszą* z dostatecznie dużych dziur. Należy przejrzeć całą listę, chyba że jest ona uporządkowana według wymiarów. Ta strategia zapewnia najmniejsze pozostałości po przydiale.

¹ Na określenie niezagospodarowanych fragmentów pamięci używa się też terminu „nieuzYTEK” (ang. *garbage*). – Przyp. tłum.

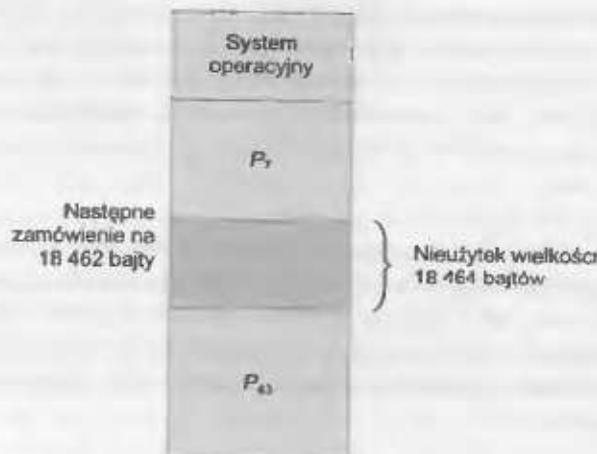
- **Najgorsze dopasowanie:** Przydziela się *największą dziurę*. Również w tym przypadku należy przeszukać całą listę, chyba że jest ona uporządkowana według wymiarów. Strategia taka pozostawia po przydziale największą dziurę, która może okazać się bardziej użyteczna niż pozostałość wynikającą z podejścia polegającego na przydziale najlepiej pasującej dziury.

Symulacje wykazały, że strategie wyboru pierwszej lub najlepiej pasującej dziury są lepsze od wyboru największej dziury zarówno pod względem zmniejszania czasu, jak i zużycia pamięci. Ani przydzielanie pierwszych, ani najlepiej pasujących dziur nie jest faktycznie lepsze pod względem wykorzystania pamięci, ale przydziały na zasadzie najlepszego dopasowania są z reguły szybsze.

8.4.3 Fragmentacja zewnętrzna i wewnętrzna

Opisane w punkcie 8.4.2 algorytmy są obarczone *zewnętrzną fragmentacją* (ang. *external fragmentation*). W miarę ładowania i usuwania procesów z pamięci przestrzeń wolnej pamięci zostaje podzielona na małe kawałki. Istnienie zewnętrznej fragmentacji objawia się tym, że suma wolnych obszarów w pamięci wystarcza na spełnienie zamówienia, ale nie tworzą one spójnego obszaru. Pamięć jest poszatkowana na dużą liczbę małych dziur. Powracając do rys. 8.8, możemy zaobserwować dwie takie sytuacje. Na rysunku 8.8(a) łączna fragmentacja wynosi 260 KB, co nie wystarcza na to, aby spełnić wymagania któregoś z dwóch pozostałych procesów P_4 i P_5 . Z kolei na rys. 8.8(c) łączna zewnętrzna fragmentacja wynosi 560 KB (300 KB + 260 KB). Wystarczyłoby to do wykonania procesu P_5 (który potrzebuje 500 KB) pod warunkiem, że byłby to spojny obszar. Wolna przestrzeń pamięci została podzielona na dwa kawałki, z których żaden nie jest dostatecznie duży, by spełnić wymagania pamięciowe procesu P_5 .

Fragmentacja taka może stanowić poważny problem. W najgorszym przypadku może dojść do powstania bloków wolnej (marnowanej) pamięci między każdym dwoma procesami. Gdyby cała ta pamięć mogła być jednym wolnym blokiem, można by było wykonać więcej procesów. Dokonywanie wyboru według strategii pierwszego dopasowania albo najlepszego dopasowania może wpływać na wielkość fragmentacji. (Strategia pierwszego dopasowania jest lepsza w jednych systemach, a strategia najlepszego dopasowania – w innych). Innym czynnikiem jest ulokowanie przydziału na którymś z końców wolnego bloku (gdzie zostawić reszkę wolnej pamięci – w górnej czy dolnej części bloku?). Niezależnie od użytego algorytmu zewnętrzna fragmentacja będzie powodować trudności.

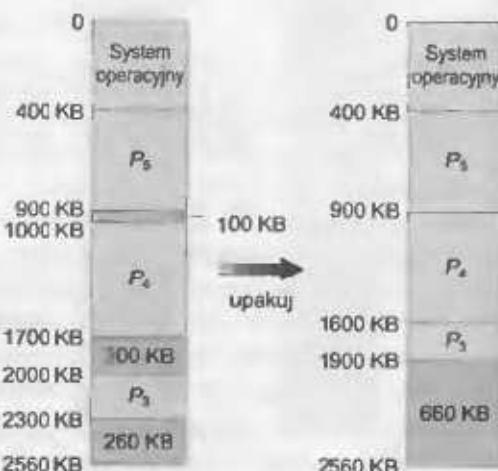


Rys. 8.9 Przydział pewnej liczby bajtów pamięci

W zależności od ogólnej ilości miejsca w pamięci i średniej długości procesu zewnętrznej fragmentacji może być problemem błahym lub poważnym. Analiza statystyczna strategii pierwszego dopasowania wykazuje na przykład, że nawet przy pewnych optymalizacjach, jeśli jest przydzielonych N bloków, to z powodu fragmentacji będzie ginąć $0,5 N$ innych bloków. Zatem jedna trzecia pamięci może być bezużyteczna! Nazywa się to *regułą 50 percent* (ang. *50-percent rule*).

Inny problem wynikający ze stosowania schematu przydzielania wielu obszarów jest pokazany na rys. 8.9. Weźmy pod uwagę dziurę wielkości 18 464 bajtów. Założymy, że następny proces wymaga 18 462 bajtów. Jeśli przydzielimy dokładnie zamówiony blok, to pozostałe dwubajtowy nieużytek. Nakład na trzymanie informacji o takiej dziurze znacznie przekraczy samą jej wielkość. Na ogół stosuje więc się metodę dołączania bardzo małych dziur do większych przydziałów. W ten sposób przydzielona pamięć może być nieco większa niż zamawiana. Różnica między tymi dwiema wielkościami stanowi *wewnętrzną fragmentację* (ang. *internal fragmentation*), czyli bezużyteczny kawałek pamięci wewnątrz przydzielonego obszaru.

Jednym z rozwiązań problemu zewnętrznej fragmentacji jest *upakowanie*. Chodzi o takie poprzemieszczanie zawartości pamięci, aby cała wolna pamięć znalazła się w jednym wielkim bloku. Na przykład pamięć, której zagospodarowanie przedstawiono na rys. 8.8(e), mogłaby być upakowana tak, jak widać na rys. 8.10. Trzy dziury wielkości 100 KB, 300 KB i 260 KB można by połączyć w jedną – o wielkości 660 KB.



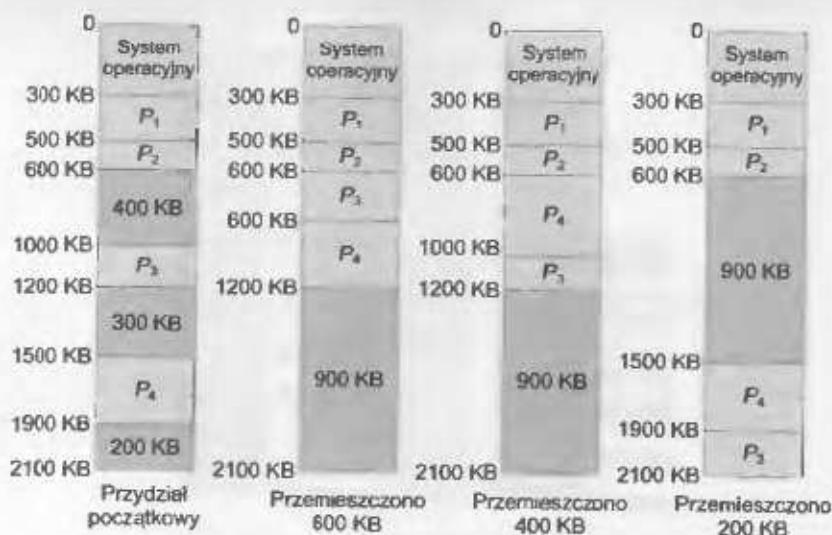
Rys. 8.10 Upakowanie pamięci

Upakowanie pamięci nie zawsze jest możliwe. Zauważmy, że na rys. 8.10 przemieszczono procesy P_4 i P_3 . Aby procesy te mogły pracować w nowych miejscach, należy zmienić wszystkie ich wewnętrzne adresy. Jeśli ustalanie adresów jest statyczne i wykonywane podczas tłumaczenia lub ładowania, to upakowanie nie jest możliwe. Możliwość upakowywania występuje tylko przy dynamicznym wiązaniu adresów wykonywanym podczas działania procesu.

Jeśli adresy są wiązane dynamicznie, to przemieszczenie procesu wprowadza się do przesunięcia programu i danych oraz do zmiany rejestru przemieszczenia dla odzwierciedlenia nowego adresu bazowego.

Jeśli istnieje możliwość upakowania, to należy oszacować jego koszt. Najprostszy algorytm upakowania polega po prostu na przesunięciu wszystkich procesów w kierunku jednego końca pamięci; wszystkie dziury lokują się na drugim końcu, tworząc jeden wielki obszar dostępnej pamięci. Schemat ten może być dość kosztowny.

Rozważmy przydział pamięci zobrazowany na rys. 8.11. Jeśli użyjemy opisanego prostego algorytmu, to będziemy musieli przesunąć procesy P_3 i P_4 , co da łącznie 600 KB do przesunięcia. W danej sytuacji moglibyśmy po prostu przesunąć proces P_4 powyżej procesu P_3 , czyli tylko 400 KB, albo przesunąć proces P_3 poniżej procesu P_4 , tym samym tylko 200 KB. Zwrócić uwagę, że w ostatnim przypadku wielka dziura wolnej pamięci powstanie nie przy końcu pamięci, lecz pośrodku. Zauważmy również, że gdyby kolejka zawierała tylko jeden proces potrzebujący 450 KB, wtedy można by spełnić to konkretne zamówienie przez przeniesienie na inne miejsce procesu P_1 (np.



Rys. 8.11 Porównanie różnych sposobów upakowania pamięci

poniżej procesu P_4). Chociaż w takim rozwiązaniu nie powstałyby jedna, duża dziura, powstały obszar wystarczłyby jednak do pilnego spełnienia zamówienia. Wybranie optymalnej strategii upakowywania jest dosyć trudne.

Upakowaniu może również towarzyszyć wymiana. Proces może zostać wysłany z pamięci operacyjnej do pomocniczej i wprowadzony na powrót w późniejszym terminie. Po wysłaniu procesu pamięć zostaje zwolniona i może być zagospodarowana przez inny proces. Z powrotem procesu do pamięci może się wiązać kilka kłopotów. W przypadku statycznego ustalania adresów proces powinien być wprowadzony w dokładnie to samo miejsce, które zajmował przed wymianą. To ograniczenie może powodować konieczność usunięcia z pamięci innego procesu w celu zwolnienia miejsca.

Jesli stosuje się przemieszczanie dynamiczne (np. przy użyciu rejestru przemieszczenia i rejestru granicznego), to proces może zostać wysłany do innego miejsca pamięci. W tym przypadku odnajduje się wolny blok pamięci, dokonując w razie potrzeby upakowania, i umieszcza w nim proces.

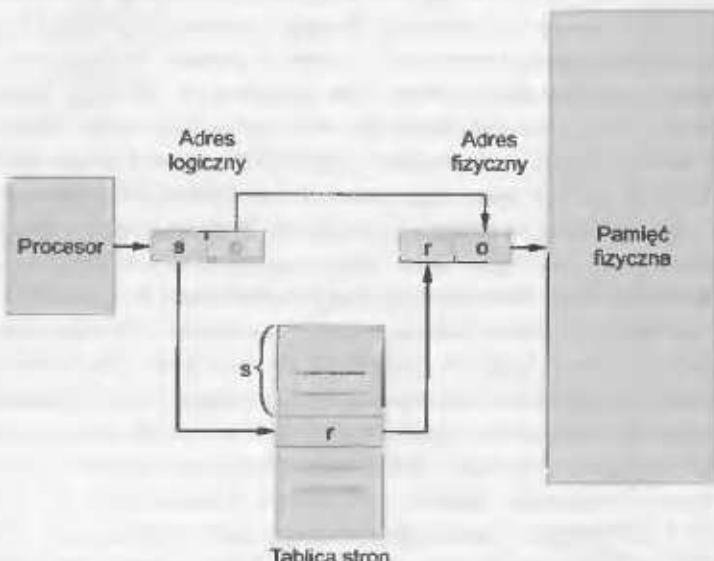
Jedna z metod upakowywania polega na usuwaniu z pamięci procesów, które mają być przesłane do innego miejsca, i wprowadzaniu ich na nowe miejsca. Jesli wymiana procesów, czyli ekspediowanie ich z pamięci operacyjnej do pomocniczej i z powrotem, jest już częścią systemu, to dodatkowy kod realizujący upakowywanie może być minimalny.

8.5 ■ Stronicowanie

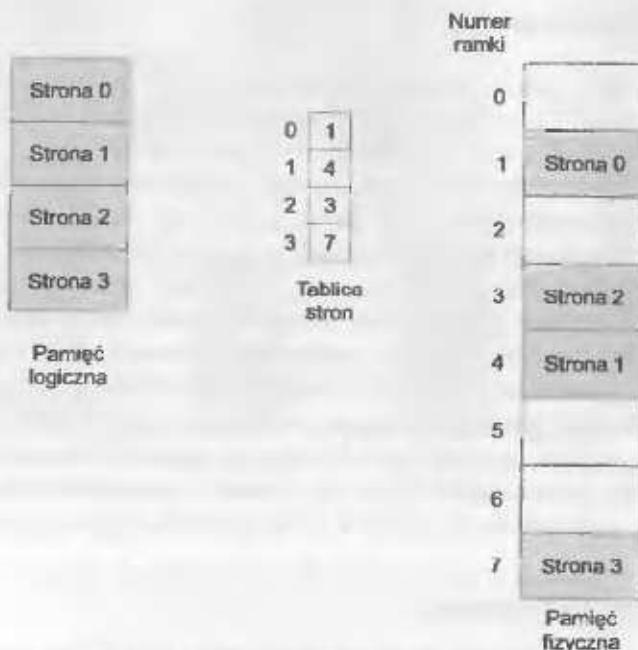
Inne możliwe rozwiązanie problemu zewnętrznej fragmentacji polega na dopuszczeniu nieciągłości logicznej przestrzeni adresowej procesu, tj. zezwoleniu na to, aby procesowi można było przydzielac dowolne dostępne miejsca w pamięci fizycznej. Jednym ze sposobów implementacji takiego rozwiązania jest zastosowanie schematu *stronicowania* (ang. *paging*). Stosując stronicowanie, omija się problem dopasowywania kawałków pamięci o zmiennych rozmiarach do miejsca w pamięci pomocniczej, co jest bolączką większości uprzednio umówionych schematów zarządzania pamięcią. Gdy trzeba wymienić pewne fragmenty kodu lub danych pozostających w pamięci operacyjnej, wówczas należy znaleźć miejsce w pamięci pomocniczej. Problemy fragmentacji, omówione w odniesieniu do pamięci operacyjnej, dotyczą w tym samym stopniu pamięci pomocniczej, z tym że dostęp do niej jest znacznie powolniejszy, co uniemożliwia upakowywanie. Rozmaite formy stronicowania, dzięki ich zaletom w porównaniu z poprzednimi metodami, są w powszechnym użyciu w wielu systemach operacyjnych.

8.5.1 Metoda podstawowa

Pamięć fizyczną dzieli się na bloki stałej długości zwane *ramkami* (ang. *frames*). Pamięć logiczna jest również podzielona na bloki takiego samego rozmiaru zwane *stronami* (ang. *pages*). Gdy ma nastąpić wykonanie procesu,



Rys. 8.12 Sprzęt stronicujący



Rys. 8.13 Model stronicowania pamięci logicznej i fizycznej

wówczas jego strony, przebywające w pamięci pomocniczej, są wprowadzane w dowolne ramki pamięci operacyjnej. Pamięć pomocniczą dzieli się na stały długości bloki tego samego rozmiaru co ramki w pamięci operacyjnej.

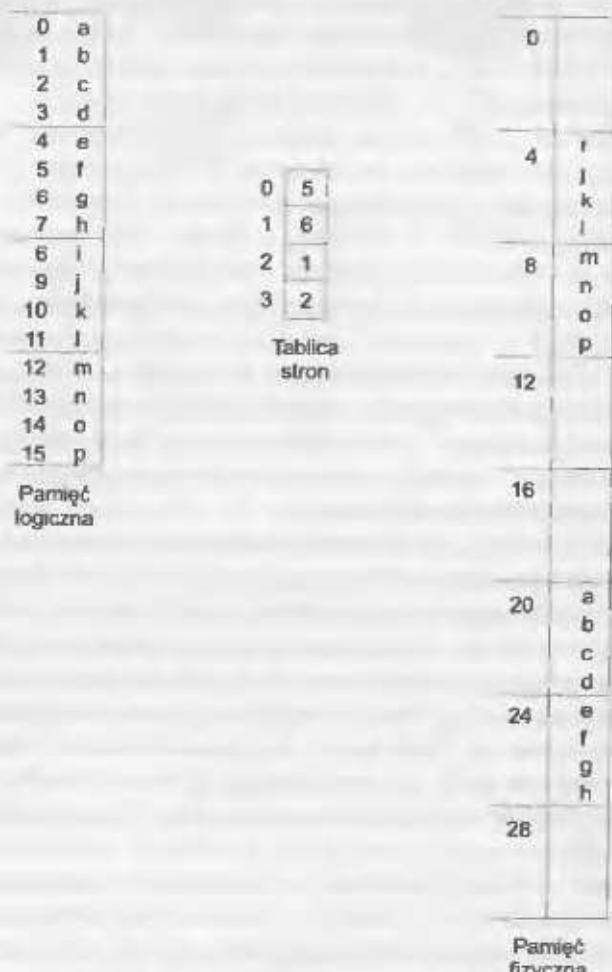
Sprzętowe zaplecze stronicowania jest pokazane na rys. 8.12. Każdy adres wygenerowany przez procesor dzieli się na dwie części: numer strony s i odległość na stronie o (ang. *page number, page offset*). Numer strony jest używany jako indeks w *tablicy stron* (ang. *page table*). Tablica stron zawiera adresy bazowe wszystkich stron w pamięci operacyjnej. W połączeniu z odlegością na stronie adres bazowy definiuje adres fizyczny pamięci – posłany do jednostki pamięci. Model pamięci stronicowanej jest przedstawiony na rys. 8.13.

Rozmiar strony (a także rozmiar ramki) jest określony przez sprzęt. Rozmiar ten jest zwykle potągią 2 z przedziału od 512 B do 16 MB na stronę – w zależności od architektury komputera. Wybór potęgi 2 jako rozmiaru strony znacznie ułatwia tłumaczenie adresu logicznego na numer strony i odległość na niej. Jeśli rozmiar logicznej przestrzeni adresowej wynosi 2^m , a rozmiar strony wynosi 2^n jednostek adresowych (bajtów lub słów), to $m - n$ bardziej znaczących bitów adresu logicznego wskazuje numer strony, a n mniej znaczących bitów wskazuje odległość na stronie. Adres logiczny wygląda zatem następująco:

numer strony	odległość na stronie
<i>s</i>	<i>o</i>
$m - n$	n

przy czym *s* jest indeksem w tablicy stron, a *o* przemieszczeniem w obrębie strony.

Jako konkretny, choć minimalny, przykład, rozważmy pamięć na rys. 8.14. Używając stron o rozmiarze 4 słów i pamięci fizycznej wielkości 32 słów (8 stron), pokazujemy w tym przykładzie, jak pamięć oglądana z punktu widze-



Rys. 8.14 Przykład stronicowania pamięci złożonej z 32 bajtów przy użyciu stron 4-bajtowych

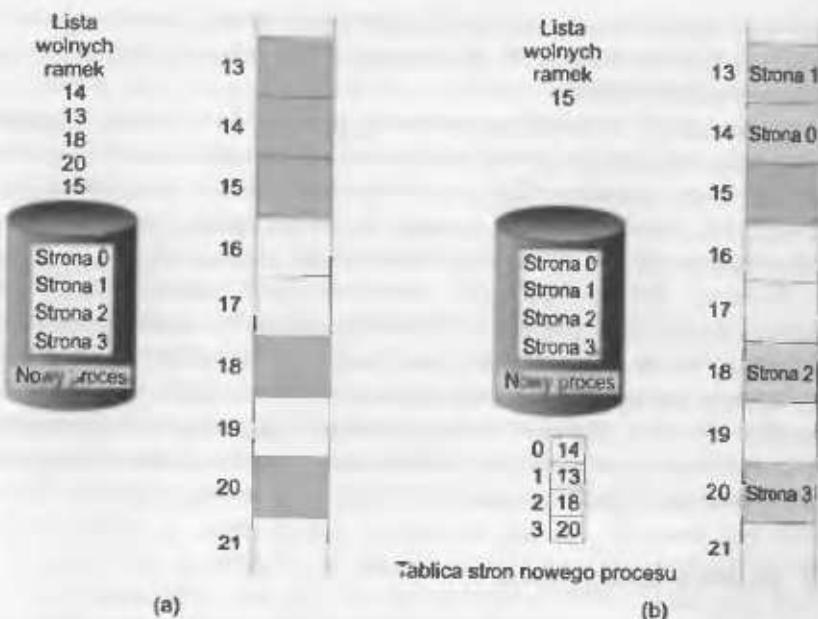
nia użytkownika może być odwzorowana na pamięć fizyczną. Adres logiczny 0 oznacza odległość 0 na stronie 0. Używając numeru strony jako indeksu tablicy stron, otrzymujemy, że strona 0 znajduje się w ramce 5. Zatem logiczny adres 0 odwzorowuje się na adres fizyczny 20 ($(5 \times 4) + 0$). Adres logiczny 3 (strona 0, odległość 3) zostanie odwzorowany na adres fizyczny 23 ($(5 \times 4) + 3$). Adres logiczny 4 oznacza odległość 0 na stronie 1. Stosownie do zawartości tablicy stron przypada mu ramka 6. Zatem adresowi 4 zostaje przyporządkowany adres fizyczny 24 ($(6 \times 4) + 0$). Adres logiczny 13 odwzorowuje się na adres fizyczny 9.

Zwróciły uwagę, że stronicowanie jest w istocie odmianą dynamicznego przemieszczania. Każdy adres logiczny zostaje powiązany za pośrednictwem sprzętu stronicującego z jakimś adresem fizycznym. Uważny czytelnik uświadomi sobie, że stronicowanie przypomina używanie tablicy rejestrów bazowych (rejestrów przemieszczeń) – po jednym na każdą ramkę pamięci.

Korzystając ze stronicowania, eliminuje się zewnętrzną fragmentację. Każda wolna ramka może być przydzielona potrzebującemu jej procesowi. Może tu jednak wystąpić fragmentacja wewnętrzna. Zauważmy, że ramki są przydzielane jako jednostki o ustalonej wielkości. Jeśli wymagania pamięciowe procesu nie odpowiadają dokładnie wielkością rozmiaru ramek, to ostatnia przydzielona rama nie będzie całkowicie wypełniona. Na przykład, gdy strona ma 2048 B, a proces 72 766 B, wówczas proces potrzebuje 35 ramek i 1086 B. Procesowi takiemu zostanie przydzielonych 36 ramek, co spowoduje wewnętrzną fragmentację o wielkości $2048 - 1086 = 962$ B. W najgorszym przypadku proces może potrzebować n stron plus jedno słowo. Przydzieliemu się $n + 1$ ramek, z czego na straty z powodu wewnętrznej fragmentacji przypadnie prawie cała rama.

Jeśli rozmiar procesu jest niezależny od rozmiaru strony, to można oczekiwac, że wewnętrzna fragmentacja wyniesie pół strony na proces. Z rózwiazań tych wynika, że odpowiedniejsze byłyby małe rozmiary strony. Jednak z każdą stroną wiążą się dodatkowe koszty spowodowane koniecznością przechowywania wpisu w tablicy stron. Ten nakład z kolei zmniejsza się ze wzrostem rozmiaru strony. Również wydajność operacji wejścia-wyjścia rośnie ze zwiększeniem się ilości danych do przesłania (rozdz. 13). Z upływem lat rozmiary stron na ogół się powiększyły, gdyż zwiększyły się rozmiary procesów, zbiorów danych i pamięci operacyjnych. Typowa dzisiejsza strona liczy 2 lub 4 KB.

Gdy proces zostaje przedłożony w systemie do wykonania, wówczas sprawdza się jego rozmiar wyrażony w stronach. Dla każdej strony procesu jest potrzebna jedna rama. Zatem, jeśli proces wymaga n stron, to w pamięci powinno być dostępnych przynajmniej n ramek. Jeśli istnieje n swobodnych ramek, to przydziela się je przybywającemu procesowi. Pierwsza strona pro-



Rys. 8.15 Wolne ramki: (a) przed przydzielaniem; (b) po przydzielaniu

cesu będzie załadowana do którejś z przydzielonych ramek, a numer tej ramki – wpisany do tablicy stron danego procesu. Nastecną stronę wprowadzi się do innej ramki, umieszczając numer ramki w tablicy stron itd. (rys. 8.15).

Ważnym aspektem stronicowania jest wyraźne rozdzielenie pamięci oglądanej przez użytkownika od pamięci fizycznej. Program użytkownika powstaje przy założeniu, że obszar pamięci jest ciągły i zawiera tylko ten jeden program. W rzeczywistości program użytkownika jest rozrzucony fragmentami po pamięci fizycznej, w której są przechowywane również inne programy. Różnice między pamięcią widzaną przez użytkownika a pamięcią fizyczną usuwa sprzęt tłumaczący adresy. Adresy logiczne są przekładane na adresy fizyczne. Odwzorowanie to jest ukrywane przed użytkownikiem i nadzorowane przez system operacyjny. Zauważmy, że proces użytkownika z definicji nie jest w stanie siegnąć do pamięci, której nie jest właścicielem. Nie ma on żadnej możliwości zaadresowania pamięci leżącej poza jego tabelą stron, z kolei tabela ta zawiera tylko te strony, które należą do procesu.

Ponieważ system operacyjny zarządza pamięcią fizyczną, musi znać wszystkie szczegóły przydzielania pamięci fizycznej: które ramki są przydzielone, które ramki są wolne, jaka jest ogólna liczba ramek itd. Informacje te są przechowywane w strukturze danych zwanej *tablicą ramek* (ang. *frame table*). Tablica ramek ma po jednej pozycji dla każdej fizycznej ramki strony.

W każdej pozycji tablicy znajdują się informacje o tym, czy ramka jest wolna czy też zajęta, a jeśli jest zajęta, to do której strony jakiego procesu – lub procesów – jest przydzielona.

Ponadto system operacyjny musi mieć pewność, że procesy użytkowników działają w przestrzeni użytkowników, a wszystkie adresy logiczne są odwzorowywane na adresy fizyczne. Jeśli użytkownik wywoła funkcję systemową (np. w celu wykonania operacji wejścia-wyjścia) i przekaże w parametrze adres (np. bufor), to adres ten musi zostać odwzorowany na poprawny adres fizyczny. System operacyjny utrzymuje kopię tablicy stron każdego użytkownika, podobnie jak kopie licznika rozkazów i zawartości rejestrów. Kopią tablicy stron służy do tłumaczenia adresów logicznych na fizyczne, ilekroć system operacyjny musi sam dokonać odwzorowania adresu logicznego na adres fizyczny. Również na jej podstawie ekspedytor określa zawartość sprzętowej tablicy stron podczas przydzielania procesowi jednostki centralnej. Stronicowanie wydłuża więc czas przełączania kontekstu.

8.5.2 Budowa tablicy stron

Każdy system operacyjny ma własne metody przechowywania tablic stron. Większość systemów przydziela tablicę stron do każdego procesu. W bloku kontrolnym procesu, oprócz wartości innych rejestrów (np. licznika rozkazów), przechowuje się wskaźnik do tablicy stron. Kiedy ekspedytor zostanie powiadomiony, że ma rozpocząć proces, wtedy musi załadować do rejestrów sprzętowych stan rejestrów użytkownika i określić właściwe wartości sprzętowej tablicy stron według przechowywanej w pamięci tablicy stron użytkownika.

8.5.2.1 Zaplecze sprzętowe

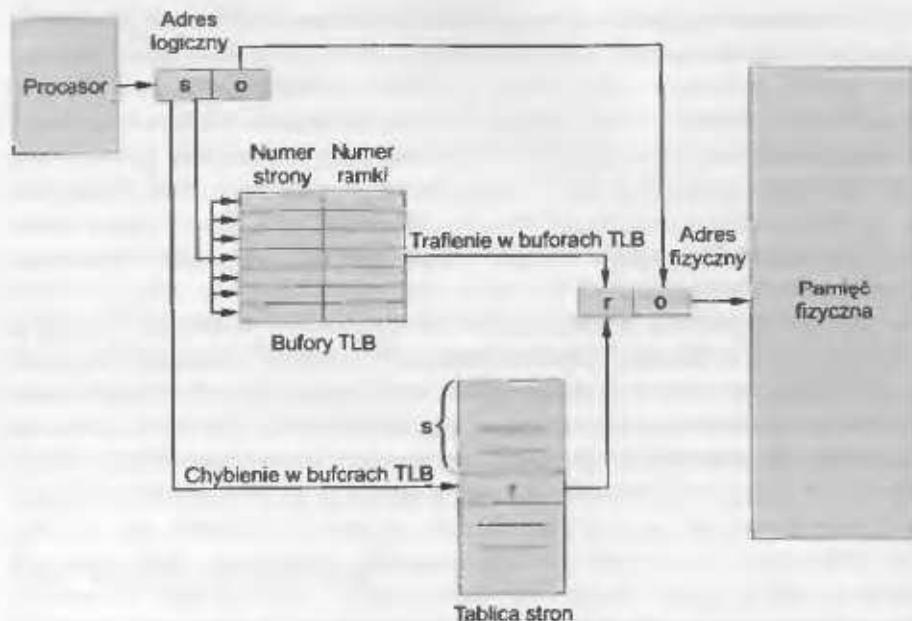
Sprzętowa implementacja tablicy stron może być wykonana na wiele różnych sposobów. W najprostszym przypadku tablicę stron implementuje się jako zbiór rejestrów specjalnego przeznaczenia. Rejestry te powinny być zbudowane z układów logicznych o wielkiej szybkości działania, aby umożliwiały wydajne tłumaczenie adresów stron. Ponieważ każde odwołanie do pamięci musi przejść przez fazę odwzorowywania strony, bardzo ważna jest wydajność mechanizmu tłumaczenia adresów. Ekspedytor określa zawartość tych rejestrów, podobnie jak to robi z innymi rejestrami procesora. Oczywiście rozkazy służące do ładowania i modyfikacji rejestrów tablicy stron są wykonywane w trybie uprzywilejowanym, tak więc tylko system operacyjny może zmieniać odwzorowanie pamięci. Przykładem takiej architektury jest komputer PDP-11 firmy DEC. Jego adres ma 16 bitów, a rozmiar strony wynosi 8 KB. Tablica stron zawiera zatem 8 wpisów przechowywanych w szybkich rejestrach.

Zastosowanie rejestrów do tablicy stron wystarcza wówczas, gdy tablica stron jest względnie mała (np. 256 pozycji). Większość obecnych komputerów pozwala jednak na posługiwanie się bardzo wielkimi tablicami stron (np. 1 milion pozycji). W takich maszynach użycie szybkich rejestrów do implementacji tablicy stron jest niemożliwe. Zamiast tego tablicę stron przechowuje się w pamięci operacyjnej, a do wskazywania jej położenia służy *rejestr bazowy tablicy stron* (ang. *page-table base register* – PTBR). Zmiana tablic stron wymaga tylko zmiany zawartości tego rejestru, co znacznie skraca czas przełączania kontekstu.

Przy tym podejściu kłopoty sprawia czas potrzebny na dostęp do komórki pamięci użytkownika. Jeśli chcemy dotrzeć do komórki i , to musimy najpierw trafić do tablicy stron za pośrednictwem rejestru bazowego tej tablicy i dodać przesunięcie o numer strony, na której wypada komórka i . To zadanie wymaga dostępu do pamięci. W rezultacie otrzymujemy numer ramki, który w połączeniu z odlegością na stronie utworzy aktualny adres. Dopiero wówczas osiągamy dostęp do potrzebnego miejsca w pamięci. Schemat ten wymaga dwóch kontaktów z pamięcią w celu uzyskania dostępu do bajtu (jeden do wpisu w tablicy stron i drugi – do danego bajtu). Zatem dostęp do pamięci ulega dwukrotnemu spowolnieniu. Takie opóźnienie w większości przypadków może być nie do przyjęcia. Równie dobrze moglibyśmy uciec się do wymiany!

Standardowym rozwiązaniem tego problemu jest zastosowanie specjalnej, malej i szybko przeszukiwanej, sprzętowej pamięci podręcznej, zwanej rozmaicie – *rejestrami asocjacyjnymi* (ang. *associative registers*) lub *buforami translkcji adresów stron* (ang. *translation look-aside buffers* – TLBs). Zbiór rejestrów asocjacyjnych jest zbudowany z wyjątkowo szybkich układów pamięciowych. Każdy rejestr składa się z dwu części: klucza i wartości. Porównywanie danego obiektu z kluczami w rejestrach asocjacyjnych odbywa się równocześnie dla wszystkich kluczy. Jeśli obiekt zostanie znaleziony, to wynikiem jest odpowiadające danemu kluczowi pole wartości. Szukanie to jest szybkie, jednak stosowny sprzęt – drogi. Liczba pozycji w buforze TLB wynosi zazwyczaj od 8 do 2048.

Rejestry asocjacyjne są używane *wraz z tablicą stron* w następujący sposób. Rejestry asocjacyjne zawierają tylko kilka wpisów z tablicy stron. Gdy procesor utworzy adres logiczny, wówczas wynikający z niego numer strony jest porównywany ze zbiorem rejestrów asocjacyjnych, które zawierają numery stron i odpowiadających im ramek. Jeśli numer strony zostanie odnaleziony w rejestrach asocjacyjnych, to odpowiedni numer ramki uzyskuje się natychmiast i używa go przy dostępie do pamięci. W porównaniu z bezpośredniim dostępem do pamięci całe zadanie może wydłużyć się o mniej niż 10%.



Rys. 8.16 Sprzęt stronicujący z buforami TLB

Jeśli numeru strony nie ma rejestrach asocjacyjnych, to trzeba odwołać się do miejsca w pamięci, w którym jest przechowywana tablica stron. Po uzyskaniu numeru ramki można jej użyć do dostępu do pamięci (rys. 8.16). Ponadto dołącza się dany numer strony i ramki do rejestrów asocjacyjnych, tak że przy następnym odwołaniu numery te zostaną szybko odnalezione. Jeśli rejesty asocjacyjne są już pełne, to system operacyjny musi wybrać któryś z nich do zastąpienia wartości. Niestety, po każdym wyborze nowej tablicy stron (np. każdorazowo podczas przełączania kontekstu), rejesty asocjacyjne muszą zostać opróżnione (ich zawartość ginie), gdyż należy zapewnić, że następny wykonywany proces nie uzyje do tłumaczenia adresów złych informacji. W przeciwnym razie w rejestrach asocjacyjnych zostałyby stare wpisy zawierające poprawne adresy wirtualne, lecz z błędymi, czyli niedopuszczalnymi adresami fizycznymi, pozostałymi po poprzednim procesie.

Procent numerów stron odnajdywanych w rejestrach asocjacyjnych nosi nazwę *współczynnika trafień* (ang. *hit ratio*). Współczynnik trafień równy 80% oznacza, że w 80 przypadkach na 100 potrzebny numer strony znajduje się w rejestrach asocjacyjnych. Jeśli przeglądnięcie rejestrów asocjacyjnych zabiera 20 ns, a dostęp do pamięci 100 ns, to gdy numer strony jest w rejestrach asocjacyjnych, wówczas odwzorowywany dostęp do pamięci zajmuje 120 ns. Jeśli odnalezienie numeru strony w rejestrach asocjacyjnych nie po-

wiedzie się (20 ns), to należy najpierw sięgnąć do pamięci po tablicę stron i numer ramki (100 ns), po czym odwołać się do właściwego słowa w pamięci (100 ns); łącznie trwa to 220 ns. Aby określić efektywny czas dostępu do pamięci (ang. *effective memory-access time*), musimy zastosować do każdego z tych przypadków wagę wynikającą z prawdopodobieństwa jego wystąpienia:

$$\text{efektywny czas dostępu} = 0,80 \times 120 + 0,20 \times 220 = 140 \text{ ns}$$

W tym przykładzie dostęp do pamięci jest obarczony spowolnieniem wynoszącym 40% (140 ns wobec 100 ns).

Dla 98-procentowego współczynnika trafień otrzymalibyśmy:

$$\text{efektywny czas dostępu} = 0,98 \times 120 + 0,02 \times 220 = 122 \text{ ns}$$

Takie zwiększenie współczynnika trafień spowodowałoby zwiększenie czasu dostępu do pamięci tylko o 22%.

Współczynnik trafień jest w oczywisty sposób zależny od liczby rejestrów asocjacyjnych. Przy użyciu od 16 do 512 rejestrów asocjacyjnych jest możliwe osiągnięcie współczynnika trafień od 80 do 98%. Procesor Motorola 68030 (stosowany w systemach Apple Macintosh) ma 22 pozycje w buforze TLB. Procesor Intel 80486 (spotykany w niektórych komputerach osobistych) ma 32 rejesty, a jego deklarowany współczynnik trafień wynosi 98%.

8.5.2.2 Ochrona

Do ochrony pamięci w środowisku stronnicowanym służą bity ochrony przypisane każdej ramce. Zazwyczaj bity te znajdują się w tablicy stron. Jeden bit może określać stronę jako dostępną do czytania i pisania albo przeznaczoną wyłącznie do czytania. Każde odwołanie do pamięci przechodzi przez tablicę stron w celu odnalezienia właściwego numeru ramki. W tym samym czasie, w którym jest obliczany adres fizyczny, można sprawdzać bity ochrony w celu zapobieżenia jakimkolwiek próbom pisania na stronie dostępnej tylko do czytania. Usiłowanie pisania na stronie przeznaczonej wyłącznie do czytania spowoduje przejście do systemu operacyjnego (próba naruszenia ochrony pamięci).

Taką metodę można łatwo rozszerzyć, aby uzyskać doskonalszą ochronę pamięci. Można zbudować sprzęt, który umożliwi albo tylko czytanie, albo czytanie i pisanie, albo wyłącznie wykonywanie. Wyodrębniając bity ochrony dla każdego rodzaju dostępu do pamięci, można też zezwolić na dowolną ich kombinację, przy czym próby niedopuszczalnych rodzajów dostępu będą wychwytywane przez system operacyjny.

Każdy wpis w tablicy stron zostaje z reguły uzupełniony o dodatkowy *bit poprawności* (ang. *valid-invalid bit*). Stan „poprawne” tego bitu oznacza, że

strona, z którą jest on związanny, znajduje się w logicznej przestrzeni adresowej procesu, a więc jest dozwolona (poprawna). Jeżeli bit poprawności ma wartość „niepoprawne”, to znaczy, że strona nie należy do logicznej przestrzeni adresowej procesu. Niedozwolone adresy są wychwytywane za pomocą sprawdzania stanu bitu poprawności. System operacyjny nadaje wartość bitowi poprawności w odniesieniu do każdej strony, zezwalając lub zakazując na korzystanie ze strony. Na przykład w systemie z 14-bitową przestrzenią adresową (od 0 do 16 383) możemy mieć program, który powinien używać tylko adresów od 0 do 10 468. Jeśli strona ma rozmiar 2 KB, to otrzymujemy sytuację przedstawioną na rys. 8.17. Adresy należące do stron 0, 1, 2, 3, 4 i 5 są odwzorowywane zwyczajnie, za pomocą tablicy stron. Jednak przy każdej próbie utworzenia adresu odnoszącego się do stron 6 lub 7 wartość bitu poprawności będzie równa „niepoprawne”, komputer spowoduje więc przejście do systemu operacyjnego (niedozwolone odwołanie do strony).

Zauważmy, że skoro adresy programu sięgają tylko wartości 10 468, więc każde odwołanie przekraczające tę wartość jest niedozwolone. Niemniej jednak odwołania do strony 5 kwalifikują się jako poprawne, co zezwala na do-

			0
			1
	Numer ramki	Bit poprawności	2 Strona 0
00000	Strona 0	0 2 p	3 Strona 1
	Strona 1	1 3 p	4 Strona 2
	Strona 2	2 4 p	5
	Strona 3	3 7 p	6
	Strona 4	4 8 p	7 Strona 3
10.468	Strona 4	5 9 p	8 Strona 4
	Strona 5	6 0 n	9 Strona 5
12.287		7 0 n	+
			Strona n

Tablica stron

Rys. 8.17 Bit poprawności (p) lub niepoprawności (n) w tablicy stron

stęp aż adresu 12 287. Niepoprawne są tylko adresy z przedziału od 12 288 do 16 383. Kłopot ten wynika z rozmiaru strony (2 KB) i odzwierciedla wewnętrzną fragmentację występującą w stronicowaniu.

Procesy rzadko działają w całym zakresie swoich adresów. W rzeczywistości używają one tylko małego ułamka dostępnej im przestrzeni adresowej. W takich przypadkach byłoby marnotrawstwem tworzenie tablicy stron z wpisami wszystkich stron należących do przedziału adresowego. Większa część takiej tablicy pozostawałaby bezużyteczna, zajmując cenną przestrzeń pamięci. Niektóre systemy korzystają ze sprzętowego *rejestru długości tablicy stron* (ang. *page-table length register* – PTLR) do wskazywania rozmiaru tablicy stron. Zawartość tego rejestru jest badana dla każdego adresu logicznego w celu sprawdzenia, czy dany adres należy do przedziału dozwolonego dla procesu. Niespełnienie tego testu powoduje błąd i przejście do systemu operacyjnego.

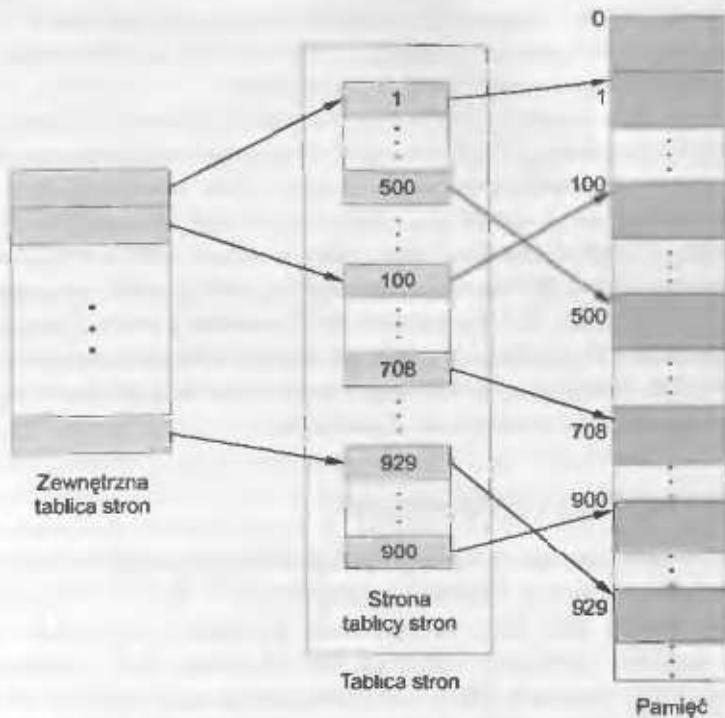
8.5.3 Stronicowanie wielopoziomowe

Większość współczesnych systemów komputerowych umożliwia stosowanie bardzo wielkich przestrzeni adresów logicznych (od 2^{32} do 2^{64}). W takim środowisku sama tablica stron staje się zbyt duża. Rozważmy na przykład system, w którym logiczna przestrzeń adresowa jest 32-bitowa. Jeśli rozmiar strony w takim systemie wyniesie 4 KB (2^{12}), to tablica stron może zawierać do miliona wpisów ($2^{32} / 2^{12}$). Ponieważ każda pozycja w tablicy ma 4 B, więc każdy proces może potrzebować do 4 MB fizycznej przestrzeni adresowej na samą tablicę stron. Jest zrozumiałe, że nie chcielibyśmy przydzielać na taką tablicę ciąglego obszaru w pamięci operacyjnej. Jednym z prostych rozwiązań jest tutaj podzielenie tablicy stron na mniejsze części. Można to wykonać na kilka sposobów.

Jeden sposób polega na zastosowaniu schematu stronicowania dwupoziomowego, w którym sama tablica stron jest podzielona na strony (rys. 8.18). Aby to zilustrować, powróćmy do naszego przykładu z 32-bitową maszyną o stronach wielkości 4 KB. Adres logiczny dzieli się na 20-bitowy numer strony i 12-bitową odległość na stronie. Ponieważ dzielimy tablicę stron na strony, numer strony podlega więc dalszemu podziałowi na 10-bitowy numer strony i 10-bitową odległość na stronie. Adres logiczny przyjmuje więc postać

numer strony		odległość na stronie
s_1	s_2	o
10	10	12

przy czym s_1 jest indeksem do zewnętrznej tablicy stron, a s_2 oznacza przesunięcie na stronie tej zewnętrznej tablicy. Tłumaczenie adresu w takiej architekturze jest pokazane na rys. 8.19. W architekturze komputera VAX stosuje



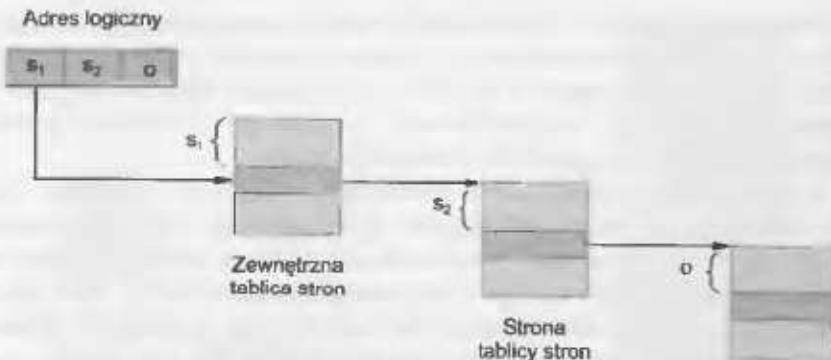
Rys. 8.18 Schemat dwupoziomowej tablicy stron

się stronicowanie dwupoziomowe. Komputer VAX ma organizację 32-bitową, a jego strony mają rozmiar 512 B. Logiczna przestrzeń adresowa procesu jest podzielona na cztery równe sekcje zawierające po 2^{30} B. Każda sekcja reprezentuje inną część logicznej przestrzeni adresowej procesu. Najstarsze 2 bity adresu logicznego określają odpowiednią sekcję. Następnych 21 bitów reprezentują numer strony logicznej w danej sekcji, a ostatnich 9 bitów zawiera odległość na danej stronie. Dzieląc tablicę stron w ten sposób, system operacyjny może nie interesować się poszczególnymi jej częściami dopóty, dopóki proces ich nie zażąda. Adres w architekturze VAX ma postać następującą:

sekcja	strona	odległość
<i>c</i>	<i>s</i>	<i>o</i>
2	21	9

przy czym *c* jest numerem sekcji, *s* – indeksem do tablicy stron, a *o* oznacza przesunięcie na stronie.

Rozmiar jednopoziomowej tablicy stron jednosegmentowego procesu w maszynie VAX wciąż jednak wynosi 2^{21} bitów \times 4 B = 8 MB. Aby



Rys. 8.19 Tłumaczenie adresu w dwupoziomowej architekturze 32-bitowej

jeszcze bardziej zmniejszyć zużycie pamięci głównej, komputer VAX stronięcie tablice stron procesów użytkowych.

W systemie z 64-bitową logiczną przestrzenią adresową schemat stronicowania dwupoziomowego okazuje się niewystarczający. Aby to zilustrować, założymy, że strona w takim systemie ma 4 KB (2^{12}). Wówczas tablica stron będzie zawierać do 2^{52} wpisów. Gdybyśmy użyli stronicowania dwupoziomowego, to wygodnie by było, żeby wewnętrzne tablice stron miały długość jednej strony, czyli zawierały 2^{10} 4-bajtowych pozycji. Adresy przybrałyby wówczas następującą postać:

strona zewnętrzna	strona wewnętrzna	odległość
s_1	s_2	o
42	10	12

Zewnętrzna tablica stron będzie zawierać 2^{42} pozycji, czyli 2^{44} bajtów. Oczywiście sposobem uniknięcia tak wielkiej tablicy jest podzielenie tablicy zewnętrznej na mniejsze części. Metoda ta jest również stosowana w niektórych procesorach 32-bitowych w celu uzyskania dodatkowej elastyczności i wydajności.

Można to zrobić na różne sposoby. Możemy postronicować zewnętrzną tablicę stron, w wyniku czego uzyskamy trzypoziomowy schemat stronicowania. Przypuszcmy, że zewnętrzna tablica stron jest zbudowana ze stron o standardowym rozmiarze (2^{10} wpisów, czyli 2^{12} B); mimo to 64-bitowa przestrzeń adresowa nie wygląda zachęcająco:

druga	strona zewnętrzna	strona zewnętrzna	strona wewnętrzna	odległość
s_1	s_2	s_3	o	
32	10	10	10	12

Zewnętrzna tablica stron ma więc 2^{34} B.

Następnym krokiem byłoby stronicowanie czteropoziomowe, w którym tablica stron drugiego zewnętrznego poziomu podlegałaby również stronicowaniu. Architektura komputera SPARC (o 32-bitowym adresowaniu) zawiera schemat stronicowania trzypodziomowego, podczas gdy 32-bitowy procesor Motorola 68030 na stronicowanie czteropoziomowe.

Jaki jest wpływ stronicowania wielopoziomowego na wydajność systemu? Zakładając, że każdy poziom jest przechowywany w postaci osobnej tablicy w pamięci operacyjnej, przekształcenie adresu logicznego na fizyczny może wymagać czterech dostępów do pamięci. Zwiększyliśmy więc pięciokrotnie czas potrzebny do wykonania jednego dostępu do pamięci! Jednakże i w tym przypadku korzyść przynosi zastosowanie pamięci podręcznej, dzięki czemu wydajność pozostaje w rozsądnych granicach. Dla współczynnika trafień wynoszącego 98% otrzymujemy

$$\text{efektywny czas dostępu} = 0,98 \times 120 + 0,02 \times 520 = 128 \text{ ns}$$

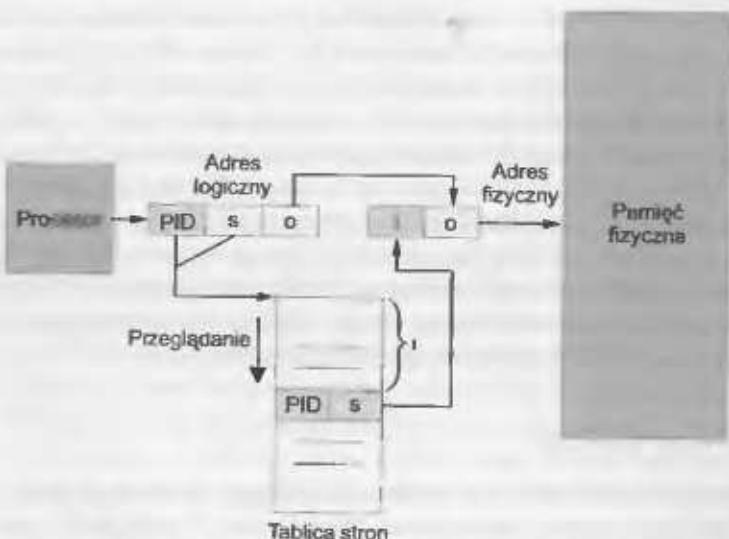
Tak więc pomimo dodatkowych poziomów przeszukiwania tablic otrzymujemy tylko 28-procentowe wydłużenie czasu dostępu do pamięci.

8.5.4 Odwrócona tablica stron

Zazwyczaj z każdym procesem jest związana tablica stron. Tablica stron ma po jednym wpisie⁴ dla każdej strony wirtualnej, używanej przez proces (lub po jednej przegródce dla każdego wirtualnego adresu, niezależnie od jego poprawności). Jest to naturalna reprezentacja, gdyż procesy odnoszą się do stron pamięci za pośrednictwem wirtualnych adresów stron. System operacyjny musi następnie tłumaczyć te odniesienia na fizyczne adresy pamięci. Ponieważ tablica jest uporządkowana według adresów wirtualnych, więc system operacyjny może obliczyć, gdzie znajduje się w niej odpowiedni adres pamięci fizycznej i używać go bezpośrednio. Jedną z wad takiego postępowania jest wielkość tablic stron, które mogą zawierać miliony pozycji. Tablice takie mogą zużywać wiele ilości pamięci fizycznej, potrzebnej tylko po to, by pokazywać, jak jest użytkowany inny obszar pamięci fizycznej.

W celu rozwiązania tego problemu w niektórych systemach operacyjnych zastosowano odwróconą tablicę stron (ang. *inverted page table*). Odwrócona tablica stron ma po jednej pozycji dla każdej rzeczywistej strony pamięci (ramki). Każda pozycja zawiera adres wirtualny strony przechowywanej w ramce rzeczywistej pamięci oraz informacje o procesie, do którego strona należy. W systemie istnieje zatem tylko jedna tablica stron, mająca tylko po

⁴ Termin „wpis” (ang. *entry*) rozumiemy jako pozycję tablicy celowo zapelnioną. Przez „pozycję” rozumiemy raczej samo oddzielne miejsce w tablicy, choć oba pojęcia stosujemy też zamiennie. Przyp. tłum.



Rys. 8.20 Odwrócona tablica stron

jednej pozycji dla każdej strony pamięci fizycznej. Na rysunku 8.20 jest zilustrowane działanie odwróconej tablicy stron. Porównajmy go z rys. 8.12, ukazującym działanie standardowej tablicy stron. Przykładami systemów, w których stosuje się schemat tego rodzaju są komputery: IBM System/38, IBM RISC System 6000, IBM RT i stacje robocze Hewlett-Packard Spectrum.

Aby to zilustrować, opiszemy uproszczoną wersję implementacji odwróconej tablicy stron zastosowanej w komputerze IBM RT. Każdy adres wirtualny w tym systemie składa się z trójki:

<identyfikator-procesu, numer-strony, odległość>

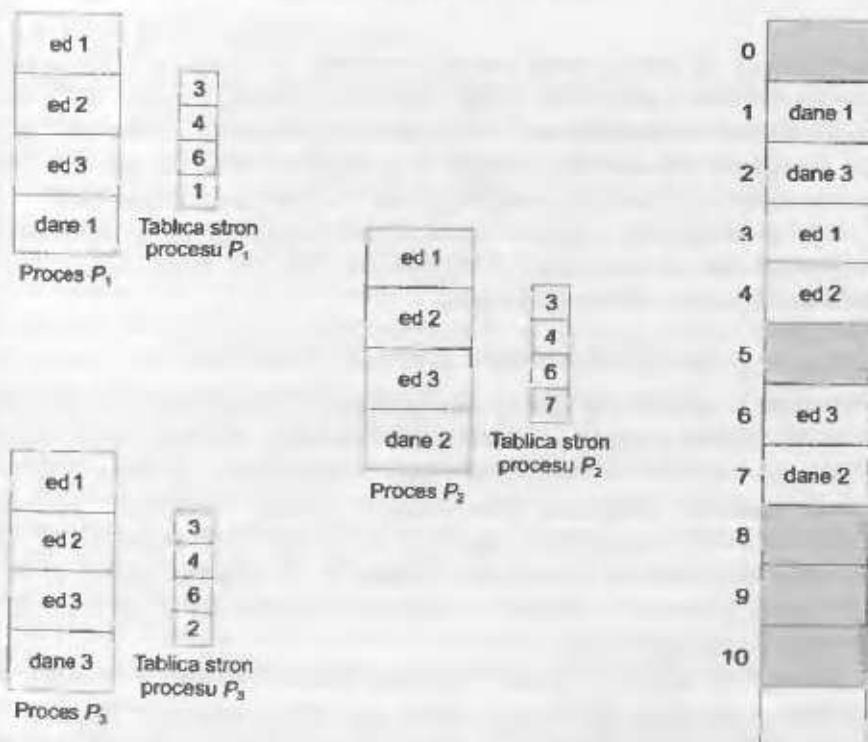
Każdy wpis w odwróconej tablicy stron jest parą *<identyfikator-procesu, numer-strony>*. Gdy pojawi się odwołanie do pamięci, wówczas część adresu wirtualnego złożona z elementów pary *<identyfikator-procesu, numer-strony>* zostanie przekazana podsystemowi pamięci. Następuje wówczas przeszukanie odwróconej tablicy stron w celu dopasowania adresu. Jeśli dopasowanie się powiedzie, powiedzmy – spełni je i -ty element tablicy, to tworzy się adres fizyczny *<i, odległość>*. Niedopasowanie oznacza, że usiłowano użyć niedozwolonego adresu.

Aczkolwiek opisany schemat zmniejsza rozmiar pamięci potrzebnej do pamiętania wszystkich tablic stron, jednak zwiększa czas potrzebny do przeszukania tablicy przy odwołaniu do strony. Ponieważ odwrócona tablica stron jest uporządkowana według adresów fizycznych, a przeglądanie dotyczy adre-

sów wirtualnych, zatem w celu znalezienia dopasowania należy przeszukać ją w całości. Szukanie takie może trwać o wiele za długo. Aby złagodzić tę niedogodność, stosuje się *tablicę haszowania* (ang. *hash table*), która umożliwia ograniczenie szukania do jednego lub co najwyżej kilku wpisów z tablicy stron. Oczywiście, każdy dostęp do tablicy haszowania to jeszcze jeden zwrot do pamięci w opisywanej procedurze, tak więc jedno odniesienie do pamięci wirtualnej wymaga przynajmniej dwukrotnego pobrania zawartości pamięci rzeczywistej: jednej pozycji w tablicy haszowania i jednego wpisu w tablicy stron. Dla poprawienia działania stosuje się rejestrysty pamięci asocjacyjnej, w których przechowuje się ostatnio zlokalizowane wpisy. Rejestry te są przeglądane w pierwszej kolejności, przed zaglądami do tablicy haszowania.

8.5.5 Strony dzielone

Inną zaletą stronicowania jest możliwość dzielenia wspólnego kodu. Sprawa ta nabiera szczególnego znaczenia w środowisku z podziałem czasu. Rozważmy system z 40 użytkownikami, z których każdy korzysta z edytora. Jeśli



Rys. 8.21 Dzielenie kodu w środowisku stronicowanym

edytor składa się ze 150 KB kodu i 50 KB obszaru danych, to do obsługi 40 użytkowników potrzeba 8000 KB pamięci. Jeśli jednak kod jest *wznawialny* (ang. *reentrant*), to można się nim dzielić, jak na rys. 8.21. Widzimy na nim trzystronicowy edytor (każda strona po 50 KB; większy rozmiar strony został użyty dla uproszczenia rysunku), który jest dzielony przez trzy procesy. Każdy proces ma własną stronę danych.

Kodem wznawialnym (nazywanym również kodem czystym) jest kod, który nie modyfikuje sam siebie. Jeśli kod ma być wznawialny, to nie może nigdy zmienić się podczas wykonania. Dzięki temu kilka procesów może wykonywać ten sam kod w tym samym czasie. Każdy proces ma swoją kopię rejestrów i obszar danych do przechowywania swoich danych podczas działania. Dane w dwóch różnych procesach będą, oczywiście, różne dla każdego procesu.

W pamięci fizycznej wystarczy przechowywać tylko jedną kopię edytora. Tablica stron każdego z użytkowników odwzorowuje adresy rozkazów na tę samą fizyczną kopię edytora, natomiast strony danych są odwzorowywane na różne ramki. Tak więc do obsługi 40 użytkowników wystarczy tylko jedna kopia edytora (150 KB) oraz 40 kopii obszarów danych, po 50 KB każda. Łączne zapotrzebowanie na przestrzeń wynosi teraz 2150 KB zamiast 8000 KB, co stanowi wyraźną oszczędność.

Podobnemu dzieleniu mogą podlegać inne, intensywnie eksploatowane programy, jak kompilatory, systemy okien, baz danych itp. Aby kod mógł być dzielony przez wiele procesów, musi być wznawialny. To, że kod dzielony jest przeznaczony wyłącznie do czytania, nie powinno zależeć tylko od jego poprawności. Właściwość tę powinien wymuszać system operacyjny. Dzielenie tej samej pamięci przez procesy systemu jest podobne do dzielenia przez wątki przestrzeni adresowej zadania, jak to jest opisane w rozdz. 4.

W systemach z odwróconą tablicą stron implementowanie pamięci dzielonej napotyka trudności. Pamięć dzieloną realizuje się na ogół za pomocą dwóch adresów wirtualnych, które są odwzorowywane na jeden adres fizyczny, lecz tej standardowej metody nie można zastosować, gdyż dla każdej strony fizycznej istnieje tylko jeden wpis strony wirtualnej. Jedna strona fizyczna nie może więc mieć dwóch (lub więcej) wspólnych adresów wirtualnych.

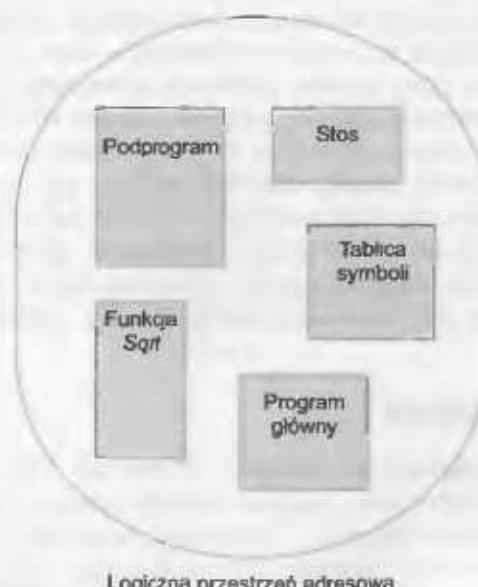
8.6 ■ Segmentacja

Ważnym aspektem zarządzania pamięcią, którego nie daje się uniknąć przy stronicowaniu, jest oddzielenie sposobu widzenia pamięci przez użytkownika od rzeczywistej pamięci fizycznej. Użytkownik wyobraża sobie pamięć inaczej niż w rzeczywistości wygląda pamięć fizyczna. Punkt widzenia użytkownika jest odwzorowywany na pamięć fizyczną. Odwzorowywanie pozwala na występowanie różnic między pamięcią logiczną a pamięcią fizyczną.

8.6.1 Metoda podstawowa

Jak użytkownik widzi pamięć operacyjną? Czy myślisz o pamięci jako o jednowymiarowej tablicy bajtów, z których jedne zawierają instrukcje, a inne – dane, czy też wybiera jakiś inny, lepszy punkt widzenia? Istnieje powszechna zgoda co do tego, że użytkownik lub programista systemu nie traktuje pamięci jako liniowej tablicy bajtów. Woli on wyobrażać sobie pamięć jako zbiór zróżnicowanych w wymiarach segmentów, bez konieczności porządkowania ich w jakiś sposób (rys. 8.22).

Zastanów się, jak myślisz o pisanym przez siebie programie? Myślisz o nim jak o zbiorze podprogramów, procedur, funkcji lub modułów z wyróżnionym programem głównym. Mogą w nim być także różne struktury danych: wykazy, tablice, stosy, zmienne proste itd. Każdy z tych modułów lub obiektów danych jest identyfikowany za pomocą nazwy. Mówisz o „tablicy symboli”, „funkcji *Sqrt*”, „programie głównym”, nie dbając o to, pod jakimi adresami w pamięci elementy te przebywają. Nie obchodzi Cię, czy tablica symboli jest zapamiętana przed, czy za funkcją *Sqrt*. Każdy z tych segmentów ma sobie właściwą długość; jest ona wewnętrznie określona przez cel, któremu segment służy w programie. Elementy wewnętrznych segmentów są identyfikowane za pomocą ich odległości od początku segmentu: pierwsza instrukcja programu, siedemnasta pozycja w tablicy symboli, piąta instrukcja funkcji *Sqrt* itd.



Rys. 8.22 Program z punktu widzenia użytkownika

Segmentacją (ang. *segmentation*) nazywa się schemat zarządzania pamięcią, który urzeczywistnia opisany sposób widzenia pamięci przez użytkownika. Przestrzeń adresów logicznych jest zbiorem segmentów. Każdy segment ma nazwę i długość. Adresy określają zarówno nazwę segmentu, jak i odległość wewnątrz segmentu. Użytkownik określa więc każdy adres za pomocą dwóch wielkości: nazwy segmentu i odległości. (Zestawmy ten schemat ze stronowaniem, w którym użytkownik określał tylko pojedynczy adres, dzielony następnie przez sprzęt na numer strony i odległość – w sposób niewidoczny dla programisty).

Dla ułatwienia implementacji segmenty są numerowane, a w odwołaniach do nich, zamiast nazw segmentów, używa się numerów segmentów. Adres logiczny tworzy więc parę

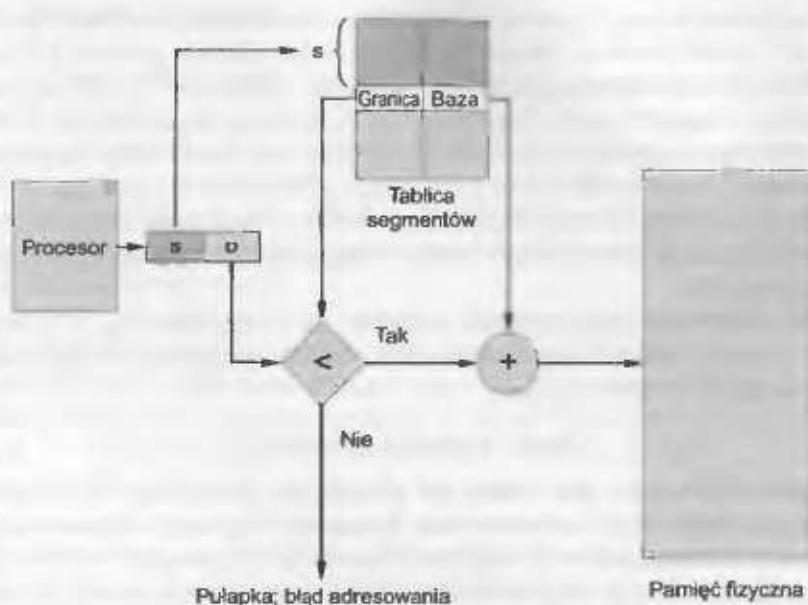
<numer-segmentu, odległość>

Program użytkownika jest zazwyczaj tłumaczony za pomocą kompilatora, przy czym kompilator automatycznie konstruuje segmenty odpowiadające programowi wejściowemu. Kompilator Pascala może wytwarzać osobne segmenty dla: (1) zmiennych globalnych; (2) stasu wywołań procedur, przeznaczonego na przechowanie parametrów i adresów powrotnych; (3) porcji kodu poszczególnych procedur lub funkcji; (4) lokalnych zmiennych każdej procedury lub funkcji. Kompilator Fortranu może tworzyć osobny segment dla każdego bloku wspólnego. Tablice mogą być przypisane do oddzielnych segmentów. Program ładujący pobierze wszystkie te segmenty i poprzedziła im numery segmentów.

8.6.2 Sprzęt

Choć użytkownik może się teraz odwoływać do obiektów w programie za pomocą adresu dwuwymiarowego, rzeczywista pamięć fizyczna jest wciąż – rzecz jasna – jednowymiarowym ciągiem bajtów. Należy zatem zdefiniować implementację odwzorowującą dwuwymiarowe adresy określone przez użytkownika na jednowymiarowe adresy fizyczne. Odwzorowanie to daje *tablica segmentów* (ang. *segment table*). Każda pozycja w tablicy segmentów składa się z *bazy segmentu* i *granicy segmentu*. Baza segmentu zawiera początkowy adres fizyczny segmentu w pamięci, a granica segmentu określa jego długość.

Zastosowanie tablicy segmentów jest przedstawione na rys. 8.23. Adres logiczny składa się z dwóch części: numeru segmentu s i odległości w tym segmencie o . Numer segmentu jest używany jako indeks do tablicy segmentów. Odległość o adresu logicznego musi zawierać się w przedziale od 0 do granicy segmentu. Jeśli tak nie jest, to uaktywnia się pułapka w systemie operacyjnym (adres logiczny poza końcem segmentu). Jeśli odległość jest poprawna,



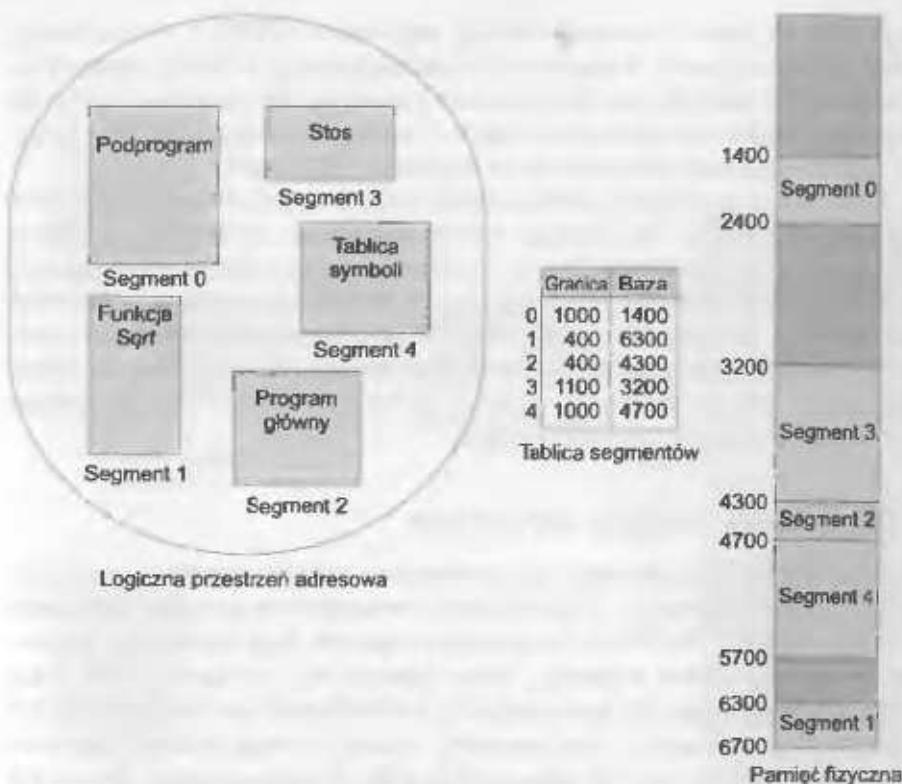
Rys. 8.23 Sprzęt do segmentacji

to dodaje się ją do bazy segmentu w celu wytworzenia fizycznego adresu potrzebnego bajtu. Tablica segmentów jest zatem wykazem par rejestrów bazy i granicy.

Jako przykład rozważmy sytuację uwidocznioną na rys. 8.24. Mamy pięć segmentów ponumerowanych od 0 do 4. Segmente są przechowywane w pamięci fizycznej tak, jak pokazano. W tablicy segmentów są oddzielne pozycje dla każdego segmentu; w każdej z nich znajduje się adres początkowy danego segmentu w pamięci fizycznej (baza) i długość tego segmentu (granica). Na przykład segment 2 ma długość 400 B i zaczyna się od adresu 4300. Zatem odwołanie do bajtu 53 segmentu 2 jest odwzorowywane na adres $4300 + 53 = 4353$. Odwołanie do bajtu 852 w trzecim segmencie odwzorowuje się na 3200 (baza segmentu 3) + 852 = 4052. Odwołanie do bajtu 1222 segmentu 0 spowoduje awaryjne przejście do systemu operacyjnego, ponieważ segment ten ma tylko 1000 B długości.

8.6.3 Implementacja tablicy segmentów

Segmentacja jest blisko związana z zaprezentowanym wcześniej modelem zarządzania obszarami pamięci. Zasadnicza różnica polega na tym, że jeden program może składać się z kilku segmentów. Nicmniej jednak koncepcja



Rys. 8.24 Przykład segmentacji

segmentacji jest bardziej złożona, dlatego opisujemy ją po omówieniu stronicowania. Podobnie jak tablica stron, tablica segmentów może być umieszczona w szybkich rejestrach albo w pamięci operacyjnej. Dostęp do tablicy segmentów przechowywanej w rejestrach jest szybki; dodawanie do bazy i porównywanie z wartością graniczną (limitem) mogą być wykonywane jednocześnie dla zaoszczędzenia czasu.

Jednakże gdy program składa się z wielkiej liczby segmentów, wówczas trzymanie tablicy segmentów w rejestrach staje się niemożliwe, toteż tablicę tę przechowuje się w pamięci operacyjnej. *Rejestr bazowy tablicy segmentów* (ang. *segment-table base register* – STBR) wskazuje na tablicę segmentów. Podobnie, ze względu na dużą zmienność liczby segmentów przypadających na program, stosuje się *rejestr długości tablicy segmentów* (ang. *segment-table length register* – STLR). Mając adres logiczny (s, o), sprawdza się najpierw poprawność numeru s segmentu, tj. czy jest on mniejszy od zawartości rejestru długości tablicy segmentów ($s < \text{STRL}$). Następnie dodaje się numer

segmentu do rejestru bazowego tablicy segmentów ($STBR + s$) i otrzymuje adres przechowywania w pamięci odpowiedniej pozycji w tablicy segmentów. Zawartość tej pozycji czyta się z pamięci i postępuje jak uprzednio: sprawdza się, czy odległość nie przekracza długości segmentu i oblicza się adres fizyczny potrzebnego bajtu jako sumę bazy segmentu i odległości.

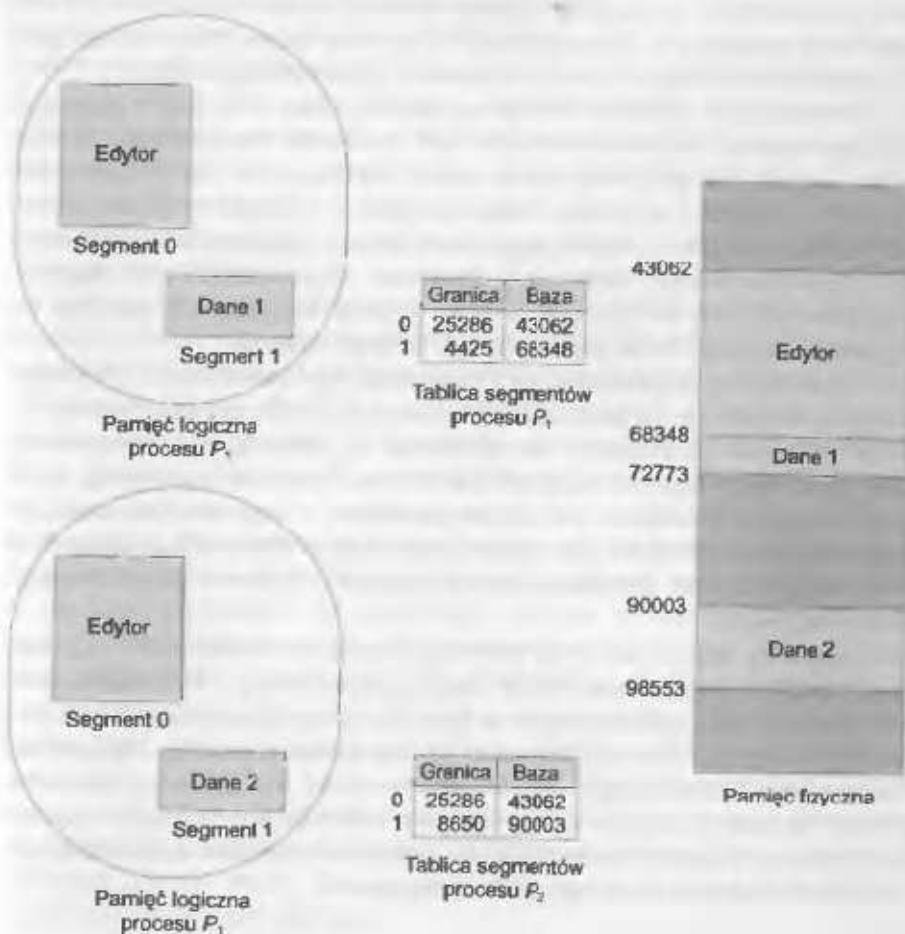
Tak jak w przypadku stronicowania, odwzorowanie takie wymaga dwu odwołań do pamięci dla każdego adresu logicznego, dwukrotnie w efekcie spowalniając system komputerowy. Trzeba więc jakoś temu zaradzić. Na ogół stosuje się zbiór rejestrów asocjacyjnych, w których są przechowywane ostatnio używane pozycje tablicy segmentów. Tak jak uprzednio, względnie mały zbiór rejestrów asocjacyjnych może na ogół skrócić czas zużywany na dostęp do pamięci, tak aby był on co najwyżej 10 lub 15% dłuższy niż czas dostępu do pamięci z pominięciem odwzorowań.

8.6.4 Ochrona i wspólne użytkowanie

Szczególną zaletą segmentacji jest powiązanie ochrony pamięci z segmentami. Ponieważ segmenty są określonymi semantycznie porcjami programu, można oczekiwać, że wszystkie elementy segmentu będą używane w ten sam sposób. Mamy zatem segmenty, które składają się z rozkazów, oraz takie, które zawierają dane. W nowoczesnych architekturach nie ma samomodyfikujących się rozkazów, toteż segmenty rozkazów mogą być zdefiniowane jako przeznaczone tylko do czytania lub tylko do wykonywania. Sprzęt odwzorowujący pamięć będzie sprawdzał bity ochrony związane z każdą pozycją tablicy segmentów, nie dopuszczając do nieuprawnionych dostępów do pamięci, takich jak próby pisania do segmentu przeznaczonego wyłącznie do czytania albo próby użycia jako danych segmentu, który jest przeznaczony tylko do wykonania. Po umieszczeniu tablicy w osobnym segmencie sprzęt zarządzający pamięcią będzie automatycznie sprawdzał, czy indeksy tej tablicy są poprawne i nie wykraczają poza jej granice. W ten sposób wiele typowych błędów programowych może być wykrywanych przez sprzęt, zanim zdążyłyby one spowodować większe spustoszenia.

Do innych zalet segmentacji należy dzielenie kodu lub danych. Każdy proces ma swoją tablicę segmentów, której ekspedytor używa do zdefiniowania sprzętowej tablicy segmentów w chwili przydzielania procesora danemu procesowi. Dzielenie segmentów występuje wtedy, gdy wpisy w tablicach dwóch różnych procesów wskazują na to samo w pamięci fizycznej (rys. 8.25).

Dzielenie występuje na poziomie segmentów. Zatem dowolna informacja może być dzielona, jeśli została zdefiniowana jako segment. Dzieleniu może podlegać kilka segmentów, więc program wielosegmentowy może być dzielony.



Rys. 8.25 Dzielenie segmentów w systemie pamięci segmentowanej

Rozważmy na przykład użytkowanie edytora tekstu w systemie z podziałem czasu. Cały kod edytora może być dość duży i składać się z wielu segmentów. Segmente te można dzielić między wszystkich użytkowników, zmniejszając zapotrzebowanie na fizyczną pamięć konieczną do redagowania tekstów. Zamiast n kopiami edytora zadowolimy się jedną jego kopią. Nienaj jednak każdemu użytkownikowi trzeba będzie zapewnić osobne segmenty na przechowywanie zmiennych lokalnych. Z tych segmentów, rzeczą jasną, nie wolno korzystać wspólnie.

Jest również możliwe dzielenie tylko fragmentów programów. Na przykład wspólne pakiety podprogramów mogą być dzielone między wielu użytkowni-

ków, jeśli zdefiniuje się je jako segmenty dzielone, przeznaczone tylko do czytania. Dwa programy w Fortranie mogą na przykład używać tego samego podprogramu *Sqrt*, do czego wystarcza w pamięci tylko jedna jego kopia.

Chociaż takie dzielenie wydaje się proste, trzeba brać pod uwagę jego delikatne aspekty. Segmenty kodu na ogół zawierają odwołania do adresów w ich obrębie. Na przykład rozkaz skoku warunkowego ma zwykły adres docelowy wyrażony w postaci numeru segmentu i odległości (przesunięcia) od początku segmentu. Numer segmentu w adresie docelowym będzie numerem segmentu danego kodu. Jeśli będziemy chcieli dzielić ten segment, wszystkie wspólnie korzystające z niego procesy będą musiały określić ów segment dzielnego kodu za pomocą tego samego numeru.

Gdybyśmy na przykład chcieli, by procedura *Sqrt* była dzielona i w jednym procesie określono by ją jako segment numer 4, w innym zaś jako segment 17, to jak procedura *Sqrt* miałaby się odwoływać do siebie samej? Ponieważ istnieje tylko jedna fizyczna kopia procedury *Sqrt*, musi ona odwoływać się do siebie w sposób jednakowy dla obu użytkowników – musi mieć jednoznaczny numer segmentu. W miarę jak wzrasta liczba użytkowników korzystających ze wspólnego segmentu, powiększa się też trudność z wyborem akceptowalnego numeru segmentu.

Segmenty danych, które nie zawierają fizycznych wskaźników i są przeznaczone tylko do czytania, można dzielić jako segmenty z różnymi numerami, podobnie jak segmenty kodu, w których nie ma bezpośrednich odwołań do ich własnych rozkazów, lecz są tylko odwołania pośrednie. Na przykład rozkaz skoku warunkowego, w którym adres skoku jest określony jako odległość względem bieżącej wartości licznika rozkazów lub względem rejestru zawierającego bieżący numer segmentu, umożliwia unikanie w kodzie bezpośrednich odwołań do bieżącego numeru segmentu.

8.6.5 Fragmentacja

Planista długoterminowy musi znaleźć i przydzielić pamięć wszystkim segmentom programu użytkownika. Sytuacja ta przypomina stronicowanie, z wyjątkiem tego, że segmenty mają zmienne długości, natomiast wszystkie strony mają ten sam rozmiar. Toteż, podobnie jak w przypadku schematu ze zmiennymi obszarami, przydział pamięci^{*} jest zagadnieniem przydziału dynamicznego, które jest – na ogół – rozwiązywane za pomocą algorytmu najlepszego lub pierwszego dopasowania.

Segmentacja może powodować zewnętrzną fragmentację, jeśli każdy z bloków wolnej pamięci jest za mały, by pomieścić cały segment. W tym

* Dla segmentów. – Przyp. tłum.

przypadku proces może po prostu czekać na zwiększenie obszaru pamięci (lub przynajmniej na pojawienie się większej dziury) lub też można zastosować upakowanie w celu utworzenia większej dziury. Ponieważ segmentacja jest w istocie algorytmem dynamicznego przemieszczania, do upakowywania pamięci można przystępować w dowolnej chwili. Jeśli planista przydziela procesora musi zwlekać z jakimś procesem z powodu kłopotów z przydziałem pamięci, to może (lecz nie musi) przeskoczyć miejsce w kolejce do procesora w poszukiwaniu procesu o niższym priorytecie, ale za to mniejszego, więc zdatnego do wykonania.

Jak poważny jest problem zewnętrznej fragmentacji przy zastosowaniu segmentowania? Czy długoterminowe planowanie z upakowywaniem może tu pomóc? Odpowiedzi na te pytania zależą przede wszystkim od średniego rozmiaru segmentu. W skrajnym przypadku możemy zdefiniować każdy proces jako jeden segment. To podejście sprowadza się do schematu obszarów o zmiennej długości. W przypadku drugiej skrajności każde słowo może zostać umieszczone w osobnym segmencie i ulegać oddzielnemu przemieszczaniu. Przy takim podejściu usuwa się zewnętrzna fragmentację całkowicie; jednak każdy bajt potrzebowałby rejestru bazowego do wykonywania przemieszczeń, co podwoiłoby zużycie pamięci! Oczywiście następnym logicznym krokiem – przy zastosowaniu małych, stałymi rozmiarów segmentów – jest stronicowanie. Uogólniając, możemy powiedzieć, że jeśli średni rozmiar segmentu jest mały, to zewnętrzna fragmentacja także będzie mała. (Dla porównania rozważmy wkładanie walizek do bagażnika samochodu – nigdy nie będą one idealnie do siebie pasować. Jeśli jednak otworzymy walizki i poukładamy poszczególne rzeczy luzem, to dopasowanie wszystkiego znacznie się poprawi). Ponieważ poszczególne segmenty są mniejsze niż cały proces, jest większa szansa na to, że uda się je pomieścić w dostępnych blokach pamięci.

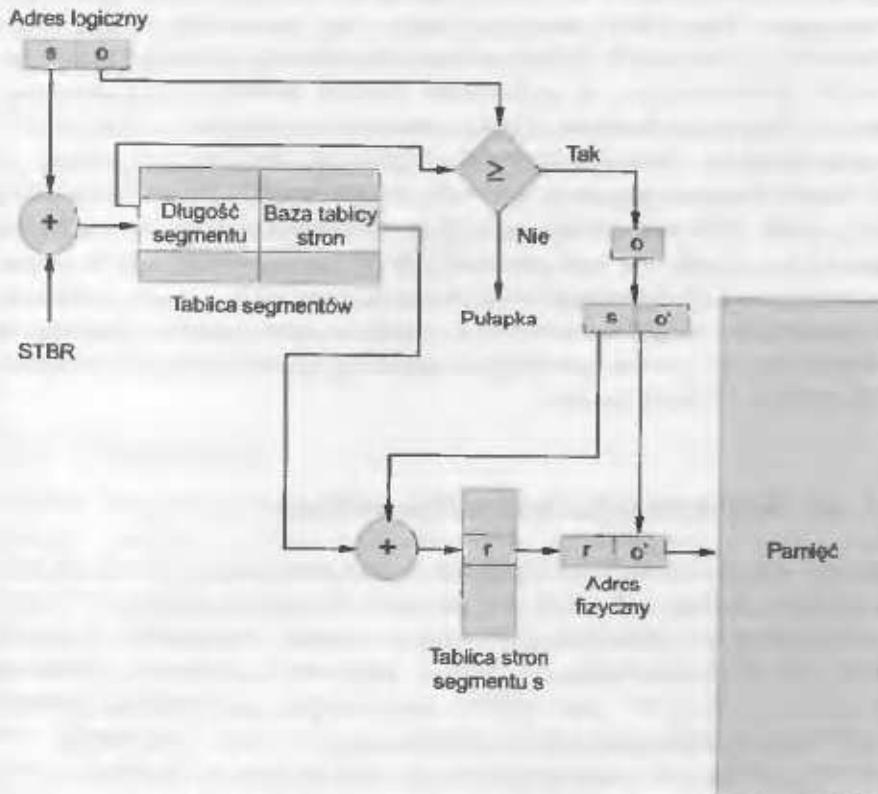
8.7 ■ Segmentacja ze stronicowaniem

Zarówno stronicowanie, jak i segmentacja mają swoje zalety i wady. W rzeczywistości, w dwu najbardziej popularnych, używanych obecnie mikroprocesorach przyjęto odmienne rozwiązania: rodzina procesorów Motorola 68000 jest zaprojektowana dla prostej przestrzeni adresowej, natomiast w rodzinie procesorów Intel 80X86 wprowadzono segmentację pamięci. W obu przypadkach zmierza się do łączenia stronicowania i segmentacji. Jest możliwe łączenie obu tych schematów w celu ulepszenia każdego z nich. Najlepiej ilustrują to dwie architektury: nowatorski, choć niezbyt szeroko używany system MULTICS oraz procesor Intel 386.

8.7.1 System MULTICS

W systemie MULTICS adres logiczny składa się z 18-bitowego numeru segmentu i 16-bitowej odległości. Mimo że schemat taki tworzy 34-bitową przestrzeń adresową, można się pogodzić z kosztami organizacji tablicy segmentów; potrzeba tylko tylu wpisów w tablicy, ile jest segmentów, a utrzymywanie pustych pozycji w tablicy segmentów jest zbyteczne.

Jednakże przy segmentach dochodzących do 64 K słów 36-bitowych, średni rozmiar segmentu może być znaczny i fragmentacja zewnętrzna może stanowić problem. Jeśli nawet fragmentacja zewnętrzna nie stanowi problemu, to czas przeszukiwania, zużyty na przydzielanie pamięci dla segmentu przy użyciu metody pierwszego dopasowania lub najlepszego dopasowania, może być długi. Możemy więc marnować pamięć z powodu zewnętrznej fragmentacji lub (albo również) tracić czas z powodu długich przeszukiwań.



Rys. 8.26 Segmentacja stronicowana w komputerze GE 645 (system MULTICS)

Zastosowane w tej sytuacji rozwiązanie polegało na *stronicowaniu segmentów*. Stronicowanie powoduje usunięcie zewnętrznej fragmentacji i uproszczenie sprawy przydziału: każda pusta ramka może być użyta na potrzebną stronę. Każda strona w systemie MULTICS składa się z 1 K słów. Wobec tego odległość w segmencie (16 bitów) podzielono na 6-bitowy numer strony i 10-bitową odległość na stronie. Numer strony jest indeksem w tablicy stron, z której uzyskuje się numer ramki. Wreszcie numer ramki w połączeniu z odlegością na stronie tworzy adres fizyczny. Schemat tłumaczenia adresu jest przedstawiony na rys. 8.26. Zauważmy, że różnica między takim rozwiązaniem a czystą segmentacją polega na tym, że wpis w tablicy segmentów zawiera nie adres segmentu, lecz adres bazowy tablicy stron tego segmentu.

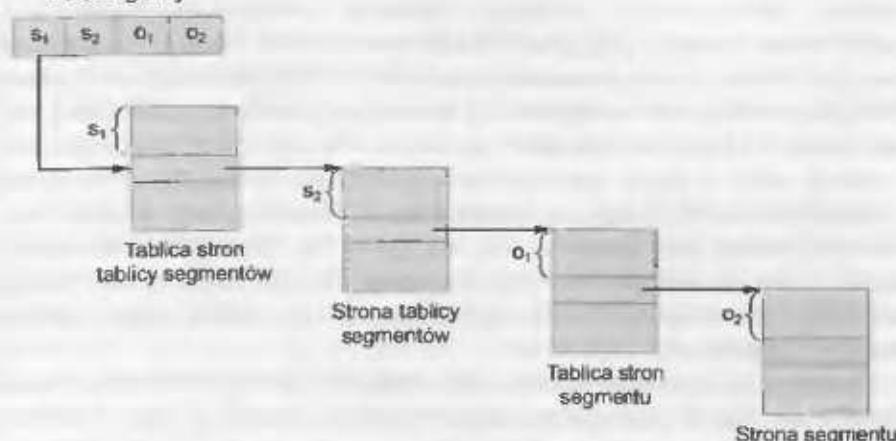
Każdy segment musi obecnie mieć oddzielną tablicę stron. Jednak zawsze, że długość każdego segmentu, określona przez jego wpis w tablicy segmentów, jest ograniczona, tablica stron nie musi mieć pełnego rozmiaru. Powinno w niej być tyle pozycji, ile ich naprawdę potrzeba. Jak zwykle w przypadku stronicowania, ostatnia strona każdego segmentu na ogół nie jest całkiem zapelniona. W efekcie otrzymujemy średnio pół strony wewnętrznej fragmentacji na segment. W konsekwencji, choć wyeliminowaliśmy fragmentację zewnętrzną, wprowadziliśmy fragmentację wewnętrzną i zwiększyliśmy nakład pamięci na przechowywanie tablic.

Prawdę powiedziałbym, nawet zaprezentowany przed chwilą obraz stronicowanej segmentacji w systemie MULTICS jest uproszczeniem. Skoro numer segmentu jest wielkością 18-bitową, należy brać pod uwagę możliwość wystąpienia 262 144 segmentów, co wymaga wyjątkowo dużej tablicy segmentów. Aby pokonać ten problem, MULTICS stronicuje tablicę segmentów! Numer segmentu (18 bitów) dzieli się na 8-bitowy numer strony i 10-bitową odległość na stronie. Wskutek tego tablica segmentów przyjmuje postać tablicy stron zawierającej do 2^8 pozycji. Ogólnie biorąc, adres logiczny w systemie MULTICS wygląda zatem następująco:

numer segmentu		odległość	
s_1	s_2	o_1	o_2
8	10	6	10

przy czym s_1 jest indeksem do tablicy stron tablicy segmentów, a s_2 jest przemieszczeniem na stronie tablicy segmentów. Kiedy już znajdziemy potrzebną nam stronę tablicy segmentów, wtedy wartość o_1 oznacza przemieszczenie w tablicy stron potrzebnego segmentu i wreszcie o_2 jest przemieszczeniem na stronie zawierającej słowo, do którego chce się uzyskać dostęp (rys. 8.27).

Adres logiczny



Rys. 8.27 Hierarchia adresu w systemie MULTICS

Aby zapewnić sensowną wydajność zastosowano 16 rejestrów asocjacyjnych zawierających adresy 16 ostatnio używanych stron. Każdy rejestr składa się z dwóch części: klucza i wartości. Klucz jest 24-bitowym polem, powstały z połączenia numeru segmentu i numeru strony. Wartość reprezentuje numer ramki.

8.7.2 System OS/2 w wersji 32-bitowej

System operacyjny IBM OS/2 w wersji 32-bitowej działa na procesorze typu Intel 386 (i późniejszych). W architekturze 386 zastosowano do zarządzania pamięcią segmentację ze stronicowaniem. Maksymalna liczba segmentów w jednym procesie wynosi 16 K, a każdy segment może mieć do 4 GB. Rozmiar strony wynosi 4 KB. Nie podamy tu pełnego opisu struktury zarządzania pamięcią przez procesor 386, natomiast przedstawimy jej podstawowe zasady.

Przestrzeń adresów logicznych procesu jest podzielona na dwie strefy. Pierwsza strefa składa się z co najwyżej 8 KB prywatnych segmentów procesu. Druga strefa zawiera do 8 KB segmentów wspólnych dla wszystkich procesów. Informacje dotyczące pierwszej strefy są przechowywane w *tablicy lokalnych deskryptorów* (ang. *local descriptor table* – LDT), a informacje dotyczące drugiej strefy są przechowywane w *tablicy globalnych deskryptorów* (ang. *global descriptor table* – GDT). Każda pozycja w tablicy LDT lub GDT ma 8 bajtów i zawiera szczegółowe dane o konkretnym segmencie, m.in. adres jego początku i długość.

Adres logiczny jest parą (selektor, odległość), przy czym selektor jest liczbą 16-bitową o postaci

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

w której *s* określa numer segmentu, *g* wskazuje, czy segment jest w tablicy deskryptorów globalnych (GDT) czy lokalnych (LDT), a pole *p* dotyczy ochrony. Odległość jest 32-bitową liczbą określającą położenie bajtu (słowa) w adresowanym segmencie.

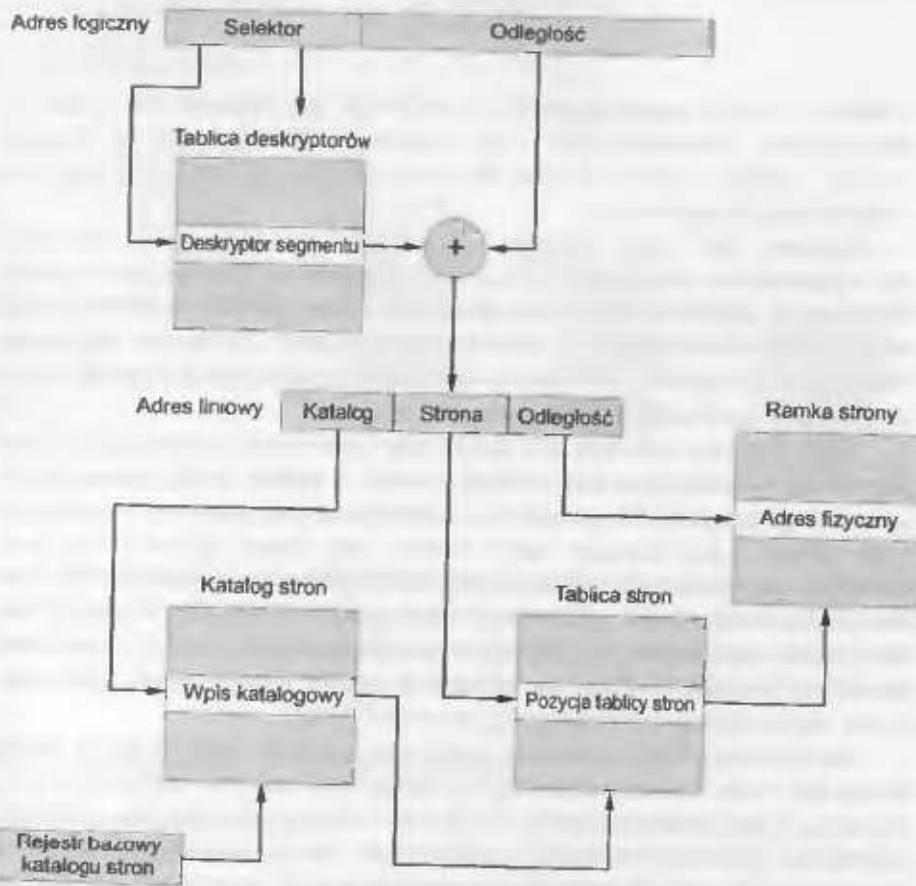
Procesor ma sześć rejestrów segmentów, co umożliwia adresowanie w procesie w dowolnej chwili szesnastu segmentów. Ma on również sześć 8-bajtowych rejestrów mikroprogramowych służących do przechowywania odpowiednich deskryptorów z tablicy LDT lub GDT. Ta pamięć podręczna eliminuje w procesorze 386 konieczność czytania deskryptora z pamięci operacyjnej podczas każdego do niej odwołania.

Adres fizyczny maszyny 386 ma 32 bity i jest tworzony następująco. Rejestr wyboru wskazuje na odpowiednią pozycję w tablicy deskryptorów lokalnych lub globalnych. Na podstawie adresu początku segmentu (bazowego) i jego długości jest tworzony *adres liniowy* (ang. *linear address*). Najpierw sprawdza się poprawność adresu ze względu na długość segmentu. Jeśli adres jest niepoprawny, to jest generowany błąd pamięci i kończy się to przejściem do systemu operacyjnego. Jeśli adres jest poprawny, to wartość odległości dodaje się do wartości bazowej, co daje w wyniku liniowy adres 32-bitowy. Adres ten jest następnie tłumaczony na adres fizyczny.

Jak powiedzieliśmy wcześniej, każdy segment jest stronicowany, a każda strona ma 4 KB. Tablica stron mogłaby zatem zawierać do 1 miliona pozycji. Ponieważ każda pozycja zajmuje 4 B, więc każdy proces mógłby potrzebować 4 MB fizycznej przestrzeni adresowej na samą tylko tablicę stron. Jest jasne, że nie chcemy umieszczać takiej tablicy stron w postaci ciągłego obszaru w pamięci operacyjnej. W maszynie 386 jako rozwiązanie przyjęto stronicowanie dwupoziomowe. Adres liniowy podzielono się na 20-bitowy numer strony i 12-bitową odległość na stronie. Ponieważ tablica stron podlega stronicowaniu, numer strony podzielono z kolei na 10-bitowy wskaźnik do katalogu stron i 10-bitowy wskaźnik do tablicy stron. Adres logiczny wygląda następująco:

numer strony		odległość na stronie
<i>s₁</i>	<i>s₂</i>	<i>o</i>
10	10	12

Schemat tłumaczenia adresów w omawianej architekturze jest podobny do przedstawionego na rys. 8.19. Tłumaczenie adresów w procesorze Intel jest pokazane bardziej szczegółowo na rys. 8.28. W celu polepszenia wydajności użytkowania



Rys. 8.28 Tłumaczenie adresu w procesorze Intel 80386

pamięci fizycznej tablice stron procesora Intel 386 można wysyłać na dysk. Wówczas w pozycjach katalogu stron stosuje się bit poprawności, aby zaznaczać, czy tablica, na której pozycję wskazano, znajduje się w pamięci operacyjnej czy na dysku. Jeśli tablica jest na dysku, to system operacyjny może skorzystać z 31 innych bitów w celu określenia położenia tablicy na dysku. Tablicę taką można potem na żądanie sprowadzić do pamięci operacyjnej.

8.8 ■ Podsumowanie

Zakres metod zarządzania pamięcią w wieloprogramowych systemach operacyjnych rozciąga się od prostych algorytmów zarządzania systemem z jednym użytkownikiem aż po skomplikowane techniki stronicowanej segmentacji

pamięci. Zasadniczym kryterium wyboru metody dla konkretnego systemu jest rodzaj sprzętu. Każdy adres pamięci utworzony przez procesor musi być sprawdzony pod kątem dopuszczalności i ewentualnie odwzorowany na adres fizyczny. Sprawdzania nie daje się (efektywnie) realizować za pomocą oprogramowania. Wynikają z tego powodu ograniczenia narzucone przez dostępny rodzaj sprzętu.

Omówione algorytmy zarządzania pamięcią (przydział ciągły, stronicowanie, segmentacja oraz łączenie stronicowania i segmentacji) różnią się pod wieloma względami. Poniższy wykaz zawiera niektóre z aspektów, które należy brać pod uwagę przy porównywaniu różnych strategii zarządzania pamięcią.

- **Wspomaganie sprzętowe:** Prosty rejestr bazowy lub para – rejestr bazowy i rejestr graniczny – wystarcza w schematach z pojedynczymi lub zwielokrotnionymi obszarami, podczas gdy stronicowanie i segmentacja wymagają tablic do definiowania odwzorowań adresów.
- **Wydajność:** W miarę komplikowania się algorytmu wzrasta czas wymagany do odwzorowania adresu logicznego na adres fizyczny. W prostych systemach wystarczy porównanie lub dodanie do adresu logicznego – operacje te są szybkie. Stronicowanie i segmentacja mogą być równie szybkie, jeśli tablica jest zaimplementowana w szybkich rejestrach. Jeśli jednak tabela znajduje się w pamięci, to czas dostępu użytkownika do pamięci może się wyraźnie pogorszyć. Spadek wydajności można zmniejszać do akceptowalnego poziomu za pomocą zbioru rejestrów asocjacyjnych.
- **Fragmentacja:** System wieloprogramowy na ogół działa wydajniej przy zwiększym poziomie wieloprogramowości. Dla danego zbioru procesów zwiększenie poziomu wieloprogramowości jest osiągalne tylko przez upakowanie większej liczby procesów w pamięci. Aby temu podążyć, należy zmniejszać ilość pamięci marnowanej wskutek fragmentacji. W systemach, w których zastosowano stałymiarowe jednostki przydziału, takie jak schemat z jednym obszarem lub stronicowanie, dają się odczuć uciążliwości wynikające z fragmentacji wewnętrznej. Systemy, w których jednostki przydziału pamięci są zmiennej długości (jak w przypadku schematu zwielokrotnionych obszarów bądź segmentacji), są zagrożone fragmentacją zewnętrzną.
- **Przemieszczanie:** Jednym ze sposobów unikania zewnętrznej fragmentacji jest upakowanie. Upakowanie powoduje przesunięcie programu w pamięci w sposób dla niego niezauważalny. Takie postępowanie wymaga, aby adresy logiczne były ustalane dynamicznie, podczas działania pro-

gramu. Jeśli adresy są ustalane tylko podczas lądowania, to pamięci nie można upakowywać.

- **Wymiana:** Do każdego algorytmu można dołączyć wymianę procesów. W odstępach określonych przez system operacyjny, wynikających zazwyczaj z przyjętych reguł planowania przydziału procesora, procesy podlegają kopiowaniu z pamięci operacyjnej do pamięci pomocniczej, skąd w późniejszym czasie są kopowane z powrotem do pamięci operacyjnej. Schemat taki pozwala na wykonywanie większej liczby procesów, aniżeli da się ich pomieścić równocześnie w pamięci.
- **Wspólne użytkowanie:** Innym środkiem podwyższenia poziomu wieloprogramowości jest dzielenie kodu i danych między różnych użytkowników. Dzielenie wymaga na ogół użycia stronicowania lub segmentacji, dostarczających małych pakietów informacji (stron lub segmentów) do podziału. Dzielenie pamięci umożliwia wykonywanie wielu procesów przy ograniczonej ilości pamięci, z tym ze wspólne programy i dane muszą być zaprojektowane starannie.
- **Ochrona:** Przy zastosowaniu stronicowania lub segmentacji różne sekcje programu użytkownika mogą być określone jako przeznaczone wyłącznie do wykonywania, wyłącznie do czytania lub do czytania i pisania. Takie ograniczenie jest niezbędne przy dzielonym kodzie lub dzielonych danych i jest z reguły przydatne w każdej sytuacji jako prosty środek kontroli wykonania programu pozwalający na wykrycie typowych błędów programowania.

■ Ćwiczenia

- 8.1 Wyjaśnij różnicę między adresami logicznymi a fizycznymi.
- 8.2 Wyjaśnij różnicę między wewnętrzną a zewnętrzną fragmentacją pamięci.
- 8.3 Wyjaśnij następujące algorytmy przydziału miejsca w pamięci:
 - (a) pierwsze dopasowanie,
 - (b) najlepsze dopasowanie,
 - (c) najgorsze dopasowanie.
- 8.4 Proces usunięty z pamięci traci możliwość używania procesora (przynajmniej chwilowo). Opisz inne sytuacje, w których proces traci możli-

- wość użytkowania procesora, ale nie zostaje wysłany do pamięci pomocniczej.
- 8.5 Mając dane obszary pamięci o rozmiarach: 100 KB, 500 KB, 200 KB, 300 KB i 600 KB (tak uporządkowane), rozważ, w jaki sposób algorytmy z punktów (a), (b) i (c) ulokują procesy o wielkości 212 KB, 417 KB, 112 KB i 426 KB (w takim porządku)? Który z algorytmów zagospodaruje pamięć najlepiej?
- 8.6 Rozważ system, w którym program można podzielić na dwie części: kod i dane. Procesor rozróżnia, czy potrzebuje rozkazu (pobranie rozkazu) czy danych (pobranie lub zapamiętanie danych). W tym celu rejestrów bazowy i graniczny występują w dwu parach – jednej dla rozkazów i jednej dla danych. Para rejestrów (bazowy i graniczny) przeznaczona dla rozkazów automatycznie umożliwia tylko czytanie, więc programy mogą dzielić różni użytkownicy. Omów zalety i wady takiego schematu.
- 8.7 Dlaczego rozmiary stron są zawsze potęgami liczby 2?
- 8.8 Rozważmy przestrzeń adresów logicznych utworzoną z ośmiu stron po 1024 słowa każda, którą odwzorowano na pamięć fizyczną składającą się z 32 ramek.
- Ile bitów zawiera adres logiczny?
 - Z ilu bitów składa się adres fizyczny?
- 8.9 Co powoduje, że proces, który nie jest właścicielem strony w systemie ze stronicowaniem, nie ma do niej dostępu? W jaki sposób system operacyjny mógłby umożliwiać dostęp do innych obszarów pamięci? Dlaczego powinien to umożliwiać lub dlaczego nie powinien na to pozwalać?
- 8.10 Rozważmy system stronicowania z tablicą stron przechowywaną w pamięci.
- Ile potrwa odwołanie do pamięci stronicowanej, jeśli czas dostępu do pamięci wynosi 200 ns?
 - Ile wyniesie czas efektywnego odwołania do pamięci, jeśli dodamy rejestrów asocjacyjne i 75% adresów wszystkich odwołań do tablicy stron będzie znajdowanych w tych rejestrach? (Załóżmy, że jeśli wpis z tablicy stron znajduje się w rejestrach asocjacyjnych, to jego odnalezienie dokonuje się w czasie zerowym).
- 8.11 Jakią są skutki zezwolenia na to, by dwie pozycje tablicy stron wskazywały tę samą ramkę w pamięci? Wyjaśnij, jak można spożytkować uzy-

skiwany efekt w celu zmniejszenia czasu potrzebnego do kopiowania wielkich ilości pamięci z jednego miejsca do innego. Co stanie się na jednej stronie w wyniku uaktualnienia jakiegoś bajta na drugiej z tych stron?

- 8.12** Dlaczego segmentację i stronicowanie łączy się czasami w jeden schemat?
- 8.13** Opisz mechanizm, z pomocą którego jeden segment będzie mógł należeć do przestrzeni adresowej dwóch różnych procesów.
- 8.14** Wyjaśnij, dlaczego łatwiej jest dzielić moduł wznowialny przy użyciu segmentacji niż przy zastosowaniu czystego stronicowania?
- 8.15** Wspólne użytkowanie segmentów przez procesy bez wymagania, by dany segment miał ten sam numer w każdym procesie, jest możliwe w systemie segmentacji z dynamicznym łączeniem.
- Zdefiniuj system umożliwiający statyczne łączenie i wspólne użytkowanie segmentów bez wymagania, aby numery segmentów wspólnych były takie same.
 - Opisz schemat stronicowania pozwalający dzielić strony bez wymagania, by ich numery były takie same.
- 8.16** Rozważmy następującą tabelę segmentów:

<u>Segment</u>	<u>Baza</u>	<u>Długość</u>
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Jakie adresy fizyczne odpowiadają następującym adresom logicznym:

- 0, 430
- 1, 10
- 2, 500
- 3, 400
- 4, 112

- 8.17** Rozważ schemat tłumaczenia adresów procesora Intel pokazany na rys. 8.28.

- (a) Opisz wszystkie kroki wykonywane przez procesor Intel 80386 podczas tłumaczenia adresu logicznego na adres fizyczny.
- (b) Jakie zalety dla systemu operacyjnego ma sprzęt wyposażony w tak złożone środki tłumaczenia adresów pamięci?
- (c) Czy taki system tłumaczenia adresów ma jakieś wady?
- 8.18 W komputerze IBM/370 ochrona pamięci odbywa się za pomocą kluczy. Klucz ma 4 bity. Kazdemu blokowi 2048 KB pamięci jest przyporządkowany klucz (klucz pamięci). Również procesor ma przypisany klucz (klucz ochrony). Operacja zapamiętania jest dozwolona, jeśli oba klucze są równe lub gdy któryś z nich jest zerem. Który z poniższych schematów zarządzania pamięcią można by z powodzeniem zastosować na takim sprzęcie:
- (a) maszyna nieosłonięta,
 - (b) system z jednym użytkownikiem,
 - (c) wieloprogramowanie ze stałą liczbą procesów,
 - (d) wieloprogramowanie ze zmieniątą liczbą procesów,
 - (e) stronicowanie,
 - (f) segmentacja?

Uwagi bibliograficzne

Dynamiczny przydział pamięci przeanalizował Knuth w książce [218, p. 2.5]. Za pomocą symulacji wykazał on, że metoda pierwszego pasującego przydziału jest w ogólnym przypadku lepsza od metody przydziału pasującego najlepiej. Tym zagadnieniem zajmowali się ponadto: Shore [387], Bays [25], Stephenson [406] i Brent [52]. Adaptowalny schemat zarządzania pamięcią na zasadzie dokładnych dopasowań (ang. *exact-fit*) zaprezentowali Oldehoeft i Allan [315]. Omówienie reguły 50 procent zostało przedstawione przez Knutha w książce [218].

Koncepcję stronicowania zawdzięczamy twórcom systemu Atlas opisanego w artykułach Kilburna i in. [213] oraz Howartha i in. [182]. Ideę segmentacji omówił po raz pierwszy Dennis w artykule [104]. Stronicowane segmentowanie wprowadzono po raz pierwszy w komputerze GE 645, dla którego zaprojektowano system MULTICS opisany przez Organicka w książce [317].

Odwroconą tablicę stron omówiono w artykule dotyczącym zarządcy pamięci w komputerze IBM RT; uczynili to: Chang i Mergen [68].

Pamięci podręczne, włącznie z pamięcią asocjacyjną, zostały opisane i przeanalizowane przez Smitha w artykule [395], który zawiera także poszerzoną bibliografię tego tematu. Hennessy i Patterson w książce [169] przedstawili sprzętowe aspekty rejestrów asocjacyjnych (TLB), pamięci podręcznych oraz jednostek zarządzania pamięcią (MMU).

Rodzinę mikroprocesorów Motorola 68000 zaprezentowano w książce [300]. Mikroprocesor Intel 8086 opisano w opublikowanej dokumentacji [189]. Sprzęt stronicujący Intel 80386 opisano w dokumentacji [191]. Stroncowanie w procesorze Intel 80386 omówił również Tanenbaum [416]. Nowszy sprzęt Intel 80486 przedstawiono w dokumentacji [192].

Rozdział 9

PAMIĘĆ WIRTUALNA

W rozdziale 8 omówiliśmy różne strategie zarządzania pamięcią, które znalazły zastosowanie w systemach komputerowych. Wszystkie one miały wspólny cel: jednoczesne utrzymywanie wielu procesów w pamięci dla umożliwienia wieloprogramowości. Każda z tych strategii wymagała jednakże, aby w pamięci znajdował się cały proces przed jego wykonaniem.

Pamięć wirtualna jest techniką, która umożliwia wykonywanie procesów, chociaż nie są one w całości przechowywane w pamięci operacyjnej. Najbardziej widoczną zaletą takiego podejścia jest to, że programy mogą być większe niż pamięć fizyczna. Co więcej, pamięć wirtualna pozwala utworzyć abstrakcyjną pamięć główną w postaci olbrzymiej, jednolitej tablicy do magazynowania informacji i oddzielić pamięć logiczną, oglądaną przez użytkownika, od pamięci fizycznej. Uwalnia to programistów od trosk związanych z ograniczeniami ilościowymi pamięci. Jednakże nie jest łatwo implementować pamięć wirtualną, może też ona istotnie obniżyć wydajność programu w przypadku niestaranego jej użycia. W niniejszym rozdziale omawiamy pamięć wirtualną w postaci stronicowania na żądanie oraz analizujemy jej złożoność i koszt.

9.1 ■ Podstawy

Głównym powodem uzasadniającym konieczność stosowania algorytmów zarządzania pamięcią z rozdz. 8 jest to, że wykonywane rozkazy muszą znajdować się w pamięci fizycznej. Pierwszy sposób spełnienia tego wymagania polega na umieszczeniu całej logicznej przestrzeni adresowej w pamięci fizycz-

nej. Ograniczenie to może być osłabione za pomocą nakładek i ładowania dynamicznego, jednak z reguły wymagają one specjalnych środków ostrożności i dodatkowego wysiłku ze strony programisty. Ograniczenie to wydaje się zarówno konieczne, jak i sensowne, jest jednak także niefortunne, ponieważ ogranicza rozmiar programu do rozmiaru pamięci fizycznej.

Badania rzeczywistych programów wykazują jednak, że w wielu przypadkach cały program nie jest potrzebny. Na przykład:

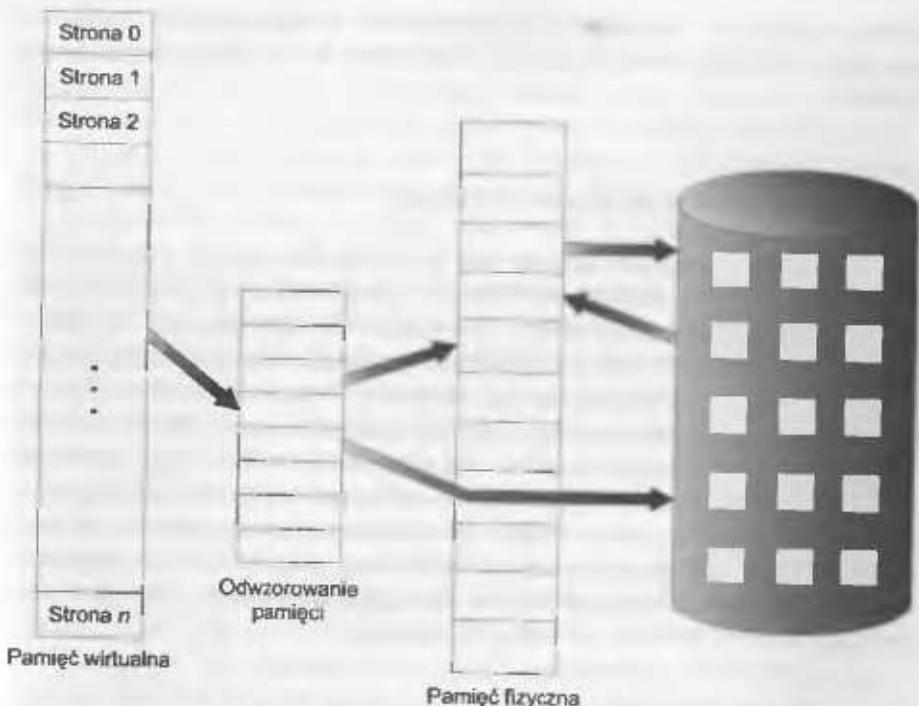
- Programy często zawierają fragmenty przeznaczone do obsługi wyjątkowo pojawiających się błędów. Ponieważ błędy takie w praktyce występują rzadko (jeśli występują w ogóle), kod ich obsługi prawie nigdy nie jest wykonywany.
- Tablice, listy i wykazy mają często więcej przydzielonej pamięci, niż naprawdę potrzebują. Bywa i tak, że tablica o zadeklarowanych wymiarach 100 na 100 rzadko kiedy ma więcej elementów niż 10 na 10. Wykaz symboli w asemblerze może mieć miejsce na 300 symboli, choć przeciętny program ma mniej niż 200 symboli.
- Pewne możliwości i właściwości programu mogą być rzadko stosowane. Na przykład amerykańskie rządowe komputery od lat nie wykonywały programów równoważenia budżetu.

Nawet wtedy, gdy trzeba, aby cały program znajdował się w pamięci, może on nie być potrzebny w całości w tym samym czasie (jest tak na przykład w przypadku nakładek).

Możliwość wykonywania programu, który tylko w części znajduje się w pamięci, ma wiele zalet:

- Program przestaje być ograniczany wielkością dostępnej pamięci fizycznej. Użytkownicy mogą pisać programy w olbrzymiej, wirtualnej przestrzeni adresowej, co upraszcza zadanie programowania.
- Każdy program użytkownika może zajmować mniejszy obszar pamięci fizycznej; można zatem w tym samym czasie wykonywać więcej programów, zwiększąc wykorzystanie procesora i przepustowość, bez zwiększania czasu odpowiedzi lub czasu cyklu przetwarzania.
- Mała liczba operacji wejścia-wyjścia koniecznych do załadunku lub wymiany programów użytkowych w pamięci, zatem każdy program użytkownika powinien wykonywać się szybciej.

Tak więc wykonywanie programu, który nie jest w całości w pamięci, powinno przynieść korzyści zarówno systemowi, jak i użytkownikowi.



Rys. 9.1 Schemat ukazujący pamięć wirtualną większą niż pamięć fizyczna

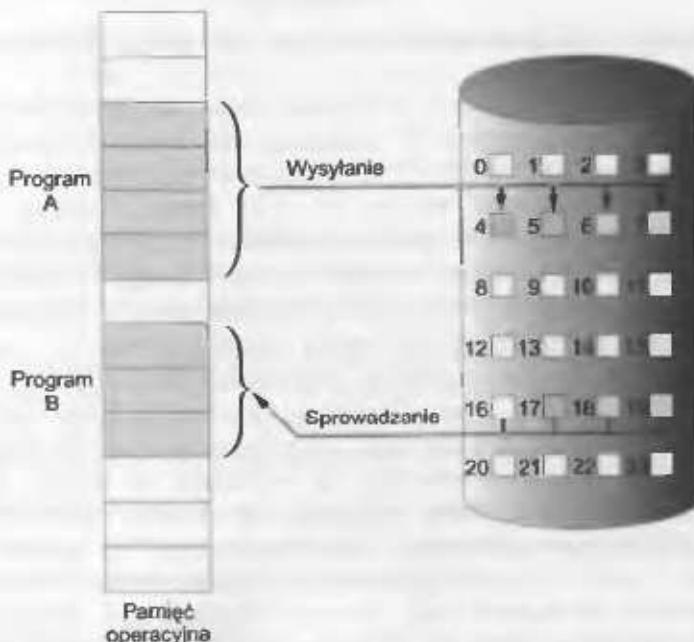
Pamięć wirtualna pozwala na odseparowanie pamięci logicznej użytkownika od pamięci fizycznej. To odseparowanie umożliwia programistom posługiwanie się olbrzymią pamięcią wirtualną nawet wtedy, gdy w pamięci fizycznej pozostaje niewiele miejsca (rys. 9.1). Pamięć wirtualna znacznie ułatwia proces programowania, gdyż programista nie musi się martwić o ilość dostępnej fizycznie pamięci ani o wybór kodu zaliczanego do nakładek, koncentrując się w zamian na programowanym zagadnieniu. W systemach realizujących pamięć wirtualną nakładki zanikły niemal zupełnie.

Pamięć wirtualna jest najczęściej implementowana w formie *stronicowania na żądanie* (ang. *demand paging*). Można ją także zrealizować w systemie segmentacji. W kilku systemach użyto schematu stronicowanego segmentowania, w którym segmenty są podzielone na strony. Wówczas z punktu widzenia użytkownika występuje segmentacja, natomiast system operacyjny może urzeczywistniać taki obraz pamięci za pomocą stronicowania na żądanie. Do realizacji pamięci wirtualnej może również posłużyć *segmentacja na żądanie* (ang. *demand segmentation*). Zastosowano ją w systemach komputerowych firmy Burroughs. Również w systemie operacyjnym IBM OS/2 stosuje się segmentację na żądanie. Algorytmy zastępo-

wania segmentów są jednak, w porównaniu z algorytmami zastępowania stron, bardziej skomplikowane ze względu na zmienne rozmiary segmentów.

9.2 ■ Stronicowanie na żądanie

System stronicowania na żądanie jest podobny do systemu stronicowania z wymianą (rys. 9.2). Procesy przebywają w pamięci pomocniczej (która zwykle jest dysk). Proces, który należy wykonać, zostaje sprowadzony do pamięci operacyjnej. Jednak zamiast przesyłania do pamięci całego procesu, stosuje się procedurę leniwej wymiany (ang. *lazy swapper*). Polega ona na tym, że nigdy nie dokonuje się wymiany strony w pamięci, jeśli nie jest to konieczne. Ponieważ traktujemy obecnie proces jako ciąg stron, a nie wielką i ciągłą przestrzeń adresową, używanie terminu „wymiana” staje się technicznie niepoprawne. Procedura wymiany operuje całymi procesami, a poszczególnymi stronami procesu zajmuje się procedura stronicująca (ang. *pager*). Toteż w kontekście wymiany stron na żądanie powinniśmy raczej używać terminu „procedura stronicująca” zamiast terminu „procedura wymiany”.



Rys. 9.2 Przesyłanie pamięci stronicowanej do ciągłego obszaru dyskowego

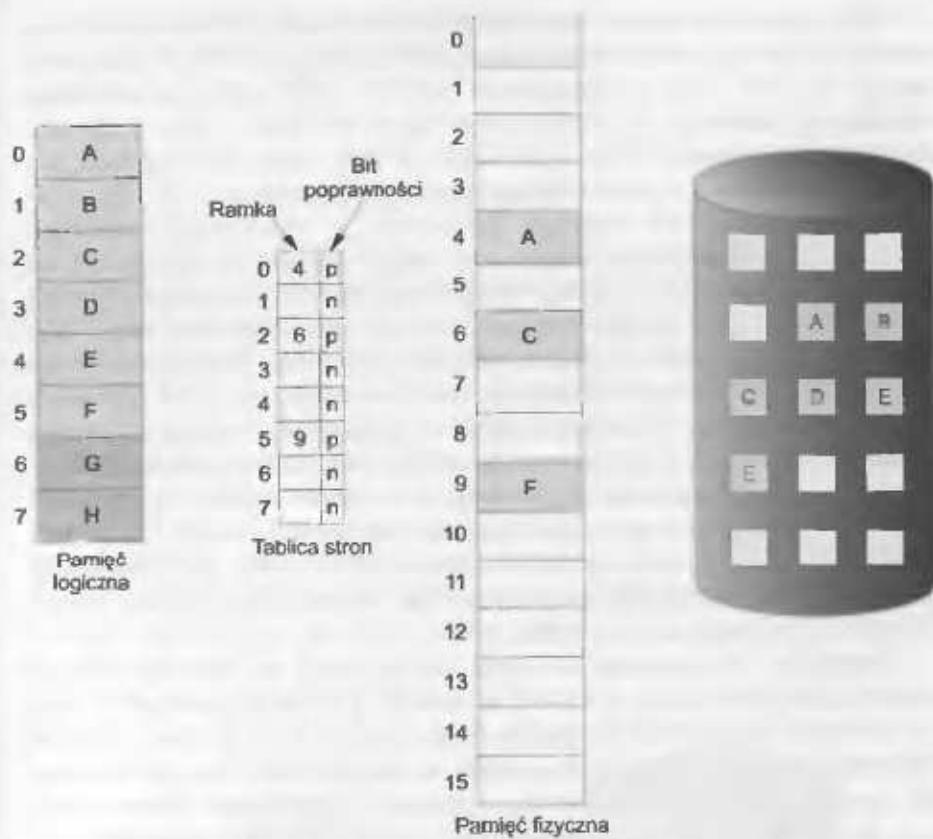
Gdy proces ma zostać wprowadzony do pamięci, wówczas procedura stronicująca zgaduje, które strony będą w użyciu przed ponownym wysłaniem procesu na dysk. Zamiast dokonywać wymiany całego procesu, procedura stronicująca sprowadza do pamięci tylko strony niezbędne. Unika w ten sposób czytania do pamięci stron, które nigdy nie będą użyte, skracając czas wymiany i zmniejszając zapotrzebowanie na pamięć fizyczną.

Postępowanie takie wymaga korzystania z odpowiednich środków sprzętowych, aby odróżniać strony pozostające w pamięci operacyjnej od stron przchwywających na dysku. W tym celu można zastosować schemat z udziałem *bitu poprawności* (ang. *valid-invalid bit*), omówiony w p. 8.5.2. Jednak tym razem, jeśli bit ten ma wartość „poprawne”, oznacza to, że odwołanie do danej strony jest dozwolone i strona znajduje się w pamięci operacyjnej. Jeśli bit poprawności ma wartość „niepoprawne”, to znaczy, że dana strona jest albo niedzwolona (tzn. nie należy do logicznej przestrzeni adresowej procesu), albo jest dozwolona, lecz w tej chwili znajduje się na dysku. Pozycja w tablicy stron, dotycząca strony sprowadzonej do pamięci, jest określona jak zwykle, natomiast pozycja odnosząca się do strony nie załadowanej, jest po prostu oznaczana jako niepoprawna lub zawiera adres dyskowy strony. Sytuacja ta jest przedstawiona na rys. 9.3.

Zauważmy, że oznaczenie strony jako niepoprawnej nie wywołuje żadnych skutków, jeśli proces nigdy się do niej nie odwoła. Toteż gdyby udawało się nam tak zgadywać, aby spośród wszystkich stron wybierać tylko te strony, które są faktycznie potrzebne, proces wykonywałby się dokładnie tak samo jak wówczas, gdy wszystkie jego strony przebywają w pamięci. Dopóki proces działa na *stronach pozostających w pamięci*, dopóty jego wykonanie przebiega normalnie.

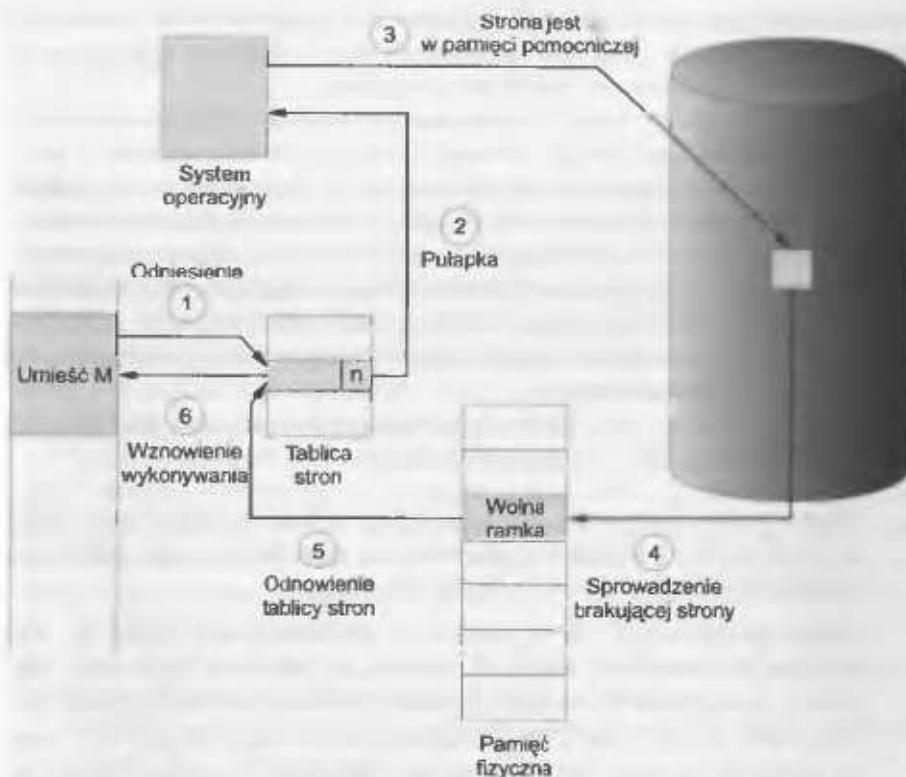
Co się jednak stanie, jeśli proces spróbuje użyć strony, której nie ma w pamięci? Próba dostępu do strony oznaczonej jako niepoprawna spowoduje błąd zwany *brakiem strony* (ang. *page-fault*). Sprzęt stronicujący, tłumacząc adres na podstawie tablicy stron, wykryje, że bit poprawności ma wartość „niepoprawne” i spowoduje awaryjne przejście do systemu operacyjnego. Omawiana pułapka jest wynikiem zawiinionego przez system operacyjny nieprzygotowania w pamięci potrzebnej strony (podyktowanego dążeniem do minimalizacji liczby operacji dyskowych i zużycia pamięci), nie oznacza natomiast błędu adresowania, powstającego wskutek usiłowania użycia niedzwolonego adresu pamięci (np. niewłaściwego wskaźnika do tablicy). Wobec tego należy skorygować to niedopatrzenie. Postępowanie związane z obsługą braku strony jest proste (rys. 9.4):

1. Sprawdzamy wewnętrzną tablicę (na ogół przechowywaną w bloku kontrolnym procesu), aby określić, czy odwołanie do pamięci było dozwolone czy też nie.



Rys. 9.3 Tablica stron z brakami stron w pamięci operacyjnej

- Jeżeli odwołanie było niedozwolone, kończymy proces. Jeśli było dozwolone, lecz zabrakło właściwej strony w pamięci, to sprawdzamy tę stronę.
- Znajdujemy wolną ramkę (np. biorąc jakąś ramkę z listy wolnych ramek).
- Zamawiamy przeczytanie z dysku potrzebnej strony do nowo przydzielonej ramki.
- Po zakończeniu czytania z dysku modyfikujemy przyporządkowaną procesowi tablicę wewnętrzną oraz tablicę stron, odnotowując w nich, że strona jest już w pamięci.
- Wznawiamy wykonanie rozkazu przerwanego wskutek tego, że zawierał odwołanie do niedozwolonego adresu. Proces może teraz sięgać do strony tak, jakby była ona zawsze w pamięci.



Rys. 9.4 Etapy obsługi braku strony

Jest ważne, aby uświadomić sobie, że po wystąpieniu braku strony, dzięki przechowyaniu stanu przerwanego procesu (rejestry, kody warunków, licznik rozkazów), jest możliwe wznowienie procesu dokładnie w tym samym miejscu i stanie, przy czym żądana strona znajduje się już w pamięci i jest dostępna. Postępując w ten sposób, można wykonywać proces nawet wtedy, gdy niektórych jego części (jeszcze) nie ma w pamięci. Gdy proces spróbuje sięgnąć do komórki, której nic ma w pamięci, wówczas sprzęt spowoduje przejście do systemu operacyjnego (brak strony). System operacyjny wczyta potrzebną stronę do pamięci i wznowi proces tak, jakby ta strona była stale w pamięci.

W skrajnym przypadku można rozpoczęć wykonywanie procesu bez żadnej strony w pamięci. Kiedy system operacyjny ustawi licznik rozkazów na pierwszy rozkaz procesu znajdujący się na stronie nieobecnej w pamięci, wtedy w procesie wystąpi natychmiast jej brak. Po sprowadzeniu tej strony do pamięci proces będzie wykonywany z ewentualnymi przerwami dopóty, dopóki wszystkie brakujące strony nie znajdą się w pamięci. Odtąd dalsze dzia-

łanie procesu może przebiegać bez zakłóceń. Tak przedstawia się *czyste stronicowanie na żądanie* (ang. *pure demand paging*) – nigdy nie sprowadza się strony do pamięci, zanim nie będzie ona potrzebna.

Niektóre programy mogą teoretycznie potrzebować kilku nowych stron pamięci do wykonania każdego rozkazu (jednej z powodu rozkazu i wielu z powodu danych), co grozi występowaniem wielu braków stron przypadających na jeden rozkaz. Sytuacja taka mogłaby prowadzić do niedopuszczalnego obniżenia sprawności systemu. Na szczęście analiza wykonywanych procesów wykazuje, że prawdopodobieństwo takiego zachowania jest niezwykle małe. W programach daje się zauważyc w właściwość *lokalności odniesień*, opisaną w p. 9.7.1, wskutek czego wyniki zastosowania stronicowania na żądanie pozostają zadowalające.

Sprzęt obsługujący stronicowanie na żądanie jest taki sam jak sprzęt do stronicowania i wymiany: w jego skład wchodzi:

- **Tablica stron:** Tablica ta jest wyposażona w wektor bitów poprawności odwołań lub też informacja o poprawności odwołań jest reprezentowana za pomocą specjalnej wartości bitów ochrony.
- **Pamięć pomocnicza:** W tej pamięci są przechowywane strony nie występujące w pamięci operacyjnej. Zazwyczaj służy do tego celu dysk. Pamięć pomocnicza bywa nazywana *urządzeniem* (do dokonywania) *wymiany* (ang. *swap device*), a część dysku przeznaczona do tego celu zwie się *obszarem wymiany* lub *pamięcią uzupełniającą*. Przydział obszaru do celów wymiany jest omówiony w rozdz. 13.

Oprócz tego wyposażenia sprzętowego jest jeszcze potrzebne odpowiednie oprogramowanie, o czym się wkrótce przekonamy.

Na sprzęt nakłada się pewne ograniczenia. Podstawowym zagadnieniem jest możliwość wznowiania wykonania rozkazu po wystąpieniu braku strony. W większości przypadków wymóg ten jest łatwy do spełnienia. Brak strony może wystąpić w dowolnym odwołaniu do pamięci. Jeśli brak strony wystąpi przy pobraniu rozkazu do wykonania, to wznowienie polega na ponownym pobraniu tego rozkazu. Jeśli brak strony wystąpi przy pobieraniu argumentu rozkazu, to należy pobrać ten rozkaz ponownie, powtórnie go zdekodować i znowu pobrać argument.

Jako najgorszy przypadek rozważmy trójadresowy rozkaz typu DODAJ A do B, umieszczając wynik w C. Etapy wykonania tego rozkazu wyglądają jak następuje:

1. Pobranie i zdekodowanie rozkazu (DODAJ).
2. Pobranie A.

3. Pobranie B.
4. Dodanie A i B.
5. Zapamiętanie sumy w C.

Jeśli niepowodzenie zdarzy się przy próbie zapamiętania w C (ponieważ C jest na stronie, której obecnie nie ma w pamięci), to trzeba będzie dotrzeć do wymaganej strony, sprowadzić ją, uaktualnić tablicę stron i rozkaz wykonać ponownie. Ponowne wykonanie rozkazu będzie wymagać powtórnego jego pobrania, zdekodowania, załadowania obu argumentów i powtórnego dodania. Niemniej jednak praca, którą trzeba powtórzyć, nie jest wielka (jest jej mniej niż potrzeba do wykonania całego rozkazu), powtórzenie zaś jest konieczne tylko w razie wystąpienia braku strony.

Zasadnicza trudność powstaje wtedy, gdy jakiś rozkaz zmienia kilka różnych komórek. Rozważmy na przykład rozkaz MVC (przemieszcz znaki) systemu IBM 360/370, służący do przesłania z jednego miejsca na drugie (być może zachodząc na siebie) do 256 B. Jeżeli którykolwiek blok bajtów (źródłowy lub docelowy) przekracza granicę strony, to brak strony może powstać po częściowym wykonaniu przesyłania. Na dodatek, jeżeli blok docelowy zachodzi na blok źródłowy, to dane w bloku źródłowym mogą ulec zniszczeniu, wykluczając proste wznowienie tego rozkazu.

Problem ten można rozwiązać na dwa różne sposoby. W jednym rozwiążaniu korzysta się z mikroprogramu, który oblicza położenie obu końców obu bloków i usiłuje do nich dotrzeć. Jeśli miałby się pojawić brak strony, to wystąpi on już w tym kroku, zanim jeszcze cokolwiek zostanie zmienione. Przesyłanie można wykonać wtedy, gdy już wiadomo, że brak strony nie wystąpi, bo wszystkie niezbędne strony znajdują się w pamięci. W drugim rozwiążaniu używa się rejestrów do chwilowego przechowywania wartości przesyłanych pól. Jeżeli zdarzy się brak strony, to wszystkie poprzednie wartości będą z powrotem przepisane do pamięci, zanim wystąpi pułapka. Działanie to odtwarza stan pamięci przed wykonaniem rozkazu, wobec czego może on być powtórzony.

Podobne kłopoty konstrukcyjne występują w maszynach, w których używa się specjalnych trybów adresowania, w tym trybów automatycznego zwiększania bądź zmniejszania adresu (np. w komputerze PDP-11). W takich trybach adresowania używa się rejestru jako wskaźnika i automatycznie zwiększa lub zmniejsza wartość w tym rejestrze według wymagań. Automatyczne zmniejszenie zawartości rejestru dokonuje się przed użyciem jej jako adresu argumentu; automatyczne zwiększenie następuje po użyciu zawartości rejestru jako adresu argumentu. Tak więc rozkaz

MOV (R2)+, -(R3)

powoduje skopiowanie zawartości komórki wskazanej przez rejestr 2 do komórki określonej przez rejestr 3. Rejestr 2 zostaje zwiększyony (o 2 dla jednego słowa, ponieważ pamięć w komputerze PDP-11 jest adresowana bajtowo) po użyciu go jako wskaźnika. Rejestr 3 jest zmniejszany (o 2) przed użyciem go jako wskaźnika. Rozważmy teraz, co się stanie, jeśli brak strony wystąpi podczas próby zapamiętania w komórce wskazywanej przez rejestr? Aby ponownie wykonać rozkaz, należałoby przywrócić obu rejestrów stan, jaki miały przed jego wykonaniem. Jedno rozwiążanie polega na wprowadzeniu specjalnego rejestru stanu, aby zapisywać w nim numer i zawartość każdego rejestru zmienianą podczas wykonywania rozkazu. Ów rejestr stanu pozwala systemowi operacyjnemu „anulować” skutki częściowo wykonanego rozkazu, który spowodował brak strony.

Nie są to jedyne problemy konstrukcyjne powstające na skutek uzupełniania istniejącej architektury sprzętu w mechanizmy stronicowania mające umożliwić stronicowanie na żądanie, ale ilustrują one część związkanych z tym trudności. Mechanizm stronicowania umieszcza się w systemie komputerowym między procesorem a pamięcią operacyjną. Powinno ono być całkowicie przezroczyste dla procesu użytkownika. Często uważa się zatem, że stronicowanie może być dodane do dowolnego systemu. Choć pogląd taki jest uzasadniony w środowisku, w którym nie ma stronicowania na żądanie, gdzie brak strony oznacza nieusuwalny błąd, nie jest on prawdziwy w sytuacji, gdy brak strony oznacza tylko, że należy sprowadzić do pamięci dodatkową stronę i wznowić proces.

9.3 ■ Sprawność stronicowania na żądanie

Stronicowanie na żądanie może mieć istotny wpływ na wydajność systemu komputerowego. Aby zrozumieć, dlaczego tak jest, obliczmy *efektywny czas dostępu* (ang. *effective access time*) do pamięci stronicowanej na żądanie. Czas dostępu do pamięci, który oznaczamy przez *cd*, w większości systemów komputerowych wynosi obecnie od 10 do 200 ns. Jeśli nie występują braki stron, to efektywny czas dostępu jest równy czasowi dostępu do pamięci. Kiedy jednak wystąpi brak strony, wtedy trzeba będzie najpierw przeczytać potrzebną stronę z dysku, a potem sięgnąć po wymagane słowo.

Niech *p* oznacza prawdopodobieństwo braku strony ($0 \leq p \leq 1$). Należałoby oczekiwać, że *p* jest bardzo bliskie zeru, tzn. że braki stron będą bardzo nieliczne. Efektywny czas dostępu wyniesie wtedy:

$$\text{efektywny czas dostępu} = (1 - p) \times cd + p \times \text{czas obsługi braku strony}$$

Aby obliczyć efektywny czas dostępu, musimy wiedzieć, ile czasu wymaga obsługa braku strony. Brak strony powoduje powstanie następującego ciągu zdarzeń:

1. Przejście do systemu operacyjnego.
2. Przechowanie rejestrów użytkownika i stanu procesu.
3. Określenie, że przerwanie było spowodowane brakiem strony.
4. Sprawdzenie, czy odniesienie do strony było dopuszczalne, oraz określenie położenia strony na dysku.
5. Wydanie polecenia czytania z dysku do wolnej ramki:
 - (a) czekanie w kolejce do tego urządzenia, aż będzie obsłużone zamówienie na czytanie;
 - (b) czekanie przez czas szukania informacji na urządzeniu i (lub) jej nadchodzenia;
 - (c) rozpoczęcie przesyłania strony do wolnej ramki.
6. Przydzielenie procesora innemu użytkownikowi na czas oczekiwania bieżącego użytkownika (nieobowiązkowe planowanie przydziału procesora).
7. Otrzymanie przerwania z dysku (zakończona operacja wejścia-wyjścia).
8. Przechowanie rejestrów i stanu procesu innego użytkownika (jeśli wykonoano krok 6).
9. Określenie, czy przerwanie pochodziło od dysku.
10. Skorygowanie zawartości tablicy stron i innych tablic w celu odnotowania, że strona jest obecnie w pamięci.
11. Czekanie na powtórne przydzielenie procesora danemu procesowi.
12. Odtworzenie stanu rejestrów użytkownika, stanu procesu i nowej tablicy stron, po czym wznowienie przerванego rozkazu.

Nie wszystkie z tych kroków są konieczne w każdym przypadku. Na przykład założyliśmy, że na czas wykonywania operacji wejścia-wyjścia procesor będzie przydzielony innemu procesowi (krok 6). Takie postępowanie pozwala na lepsze wykorzystanie procesora w trybie pracy wieloprogramowej, lecz wymaga dodatkowego czasu na wznowienie obsługi braku strony po zakończeniu operacji wejścia-wyjścia.

W każdym przypadku mamy do czynienia z trzema głównymi składnikami wpływającymi na czas obsługi braku strony:

1. Obsługą przerwania wywołanego brakiem strony.
2. Czytaniem strony.
3. Wznowieniem procesu.

Pierwsza i trzecia czynność może być zredukowana przy starannym kodowaniu do kilkuset rozkazów. Kazda z tych czynności może zająć od 1 do 100 μ s. Natomiast czas przełączenia strony będzie prawdopodobnie trwał około 24 ms. Średni czas zwłoki* w typowym dysku twardym wynosi 8 ms, czas szukania sektora jest rzędu 15 ms, czas przesyłania wynosi zaś około 1 ms. Zatem łączny czas pobrania strony (suma czasu działania sprzętu i oprogramowania) może wynosić około 25 ms**. Pamiętajmy też, że zwracamy uwagę tylko na czas obsługi urządzenia. Jeśli na dostęp do urządzenia czeka kolejka procesów (inne procesy, którym przytrafiły się braki stron), to trzeba będzie do naszego obliczenia dodać czas oczekiwania w kolejce na zwolnienie urządzenia stronicującego i przekazanie go do naszej dyspozycji, co dodatkowo wydłuży czas wymiany.

Jeśli przyjmiemy, że średni czas obsługi braku strony wynosi 25 ms, a czas dostępu do pamięci wynosi 100 ns, to efektywny czas dostępu wyniesie w nanosekundach:

$$\begin{aligned}\text{efektywny czas dostępu} &= (1 - p) \times (100) + p \times (25 \text{ ms}) \\ &= (1 - p) \times 100 + p \times 25\,000\,000 = \\ &= 100 + 24\,999\,900 \times p\end{aligned}$$

Jak widać, efektywny czas dostępu jest wprost proporcjonalny do częstotliwości braków stron. Jeśli jeden dostęp na 1000 będzie powodować brak strony, to efektywny czas dostępu wyniesie 25 μ s. Wskutek stronicowania na żądanie komputer zostanie spowolniony 250-krotnie! Jeśli chcielibyśmy, aby pogorszenie było mniejsze niż 10%, to musiałoby być

$$\begin{aligned}110 &> 100 + 25\,000\,000 \times p \\ 10 &> 25\,000\,000 \times p \\ p &< 0,0000004\end{aligned}$$

Oznacza to, że do utrzymania spowolnienia wynikającego ze stronicowania na możliwym do przyjęcia poziomie, należałoby dopuścić mniej niż 1 brak strony na każde 2 500 000 odniesień do pamięci.

* Czas pojawienia się sektora pod głowicą w ruchu obrotowym dysku. – Przyp. tłum.

** Nawet w najszybszych dyskach czas ten wynosi kilka milisekund. – Przyp. tłum.

Utrzymanie częstotliwości występowania braków stron na niskim poziomie jest ważne w systemie ze stronicowaniem na żądanie. W przeciwnym razie rośnie efektywny czas dostępu, drastycznie spowalniając wykonanie procesu.

Dodatkowym aspektem stronicowania na żądanie jest utrzymywanie obszaru wymiany i jego kompleksowe zagospodarowanie. Dyskowe operacje wejścia-wyjścia w odniesieniu do obszaru wymiany są na ogół szybsze niż wykonywane za pośrednictwem systemu plików. Ich większa szybkość wynika z tego, że bloki przydzielane w obszarze wymiany są znacznie większe oraz że nie stosuje się przeglądania plików ani metod przydziału pośredniego (zob. rozdz. 13). W celu zwiększenia przepustowości stronicowania system może więc na początku procesu przekopiować cały plik do obszaru wymiany, po czym dokonywać stronicowania na żądanie przy użyciu tego obszaru. W systemach, w których obszary wymiany są ograniczone, można w odniesieniu do plików binarnych zastosować inne postępowanie. Potrzebne strony takich plików są wprowadzane wprost z systemu plików. Kiedy dochodzi do zastępowania stron, wtedy strony takie mogą być swobodnie zapisywane przez nowe strony (ponieważ nigdy się ich nie zmienia), a w razie potrzeby – czytane z powrotem z systemu plików. Jeszcze inną możliwością jest pobieranie potrzebnych stron z systemu plików, a przy zastępowaniu posyłanie ich do obszaru wymiany. Ten sposób gwarantuje, że z systemu plików będą czytane tylko strony potrzebne, a dalsze stronicowanie będzie dokonywane za pomocą obszaru wymiany. Metoda ta wydaje się dobrym kompromisem; jest używana w systemie BSD UNIX.

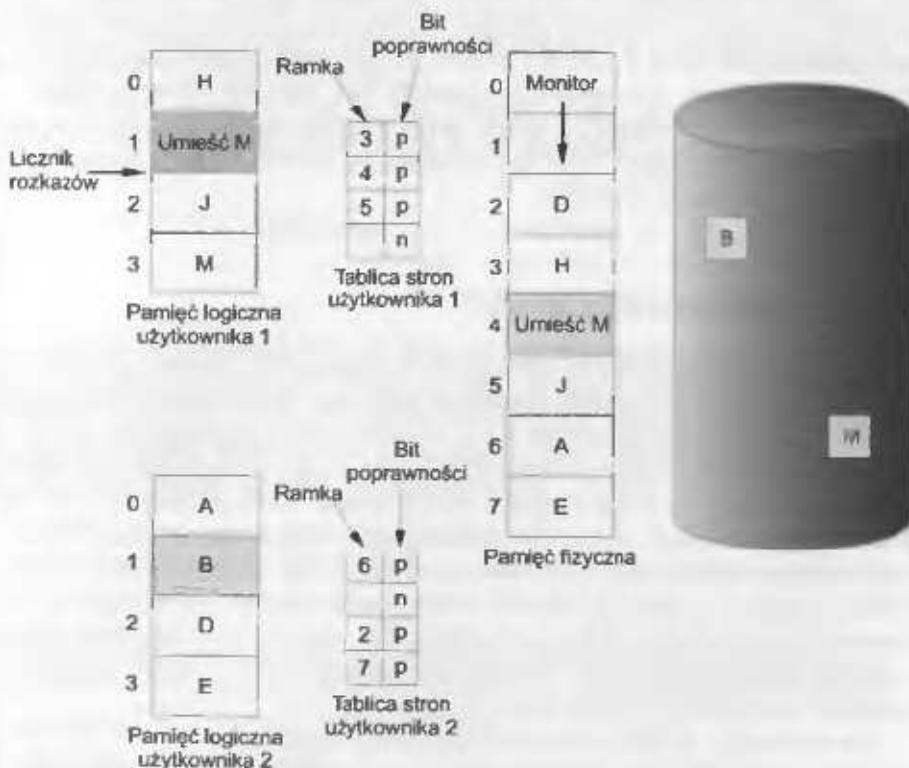
9.4 ■ Zastępowanie stron

W naszych dotychczasowych rozważaniach częstotliwość występowania braków stron nie stanowiła poważnego problemu, ponieważ każda strona mogła spowodować co najwyżej jeden taki błąd – wtedy, kiedy następowało do niej pierwsze odwołanie. Takie potraktowanie sprawy nie jest zupełnie dokładne. Zauważmy, że jeśli 10-stronicowy proces używa tylko połowy ze swoich stron, to stronicowanie na żądanie pozwala zaoszczędzić na operacjach wejścia-wyjścia, ponieważ nie wprowadza się 5 stron, które nigdy nie będą uzyte. Można również zwiększyć stopień wieloprogramowości, wykonując dwukrotnie więcej procesów. Jeśli zatem mamy 40 ramek, to zamiast 4 procesów, z których każdy zajmowałby 10 ramek (z których każde 5 ramek nigdy nie zostały użyt), moglibyśmy wykonać ich 8.

Powiększanie stopnia wieloprogramowości może doprowadzić do *nad-przydziału* (ang. *over-allocating*) pamięci. Jeśli będziemy wykonywali 6 procesów, z których każdy ma rozmiar 10 stron, lecz faktycznie używa 5 stron, to

zwiększymy zatrudnienie procesora i przepustowość, oszczędzając 10 ramek. Może się jednak zdarzyć, że każdy z tych procesów dla szczególnego zestawu danych nagle spróbuje użyć wszystkich 10 stron, co spowoduje zapotrzebowanie na 60 ramek, przy tylko 40 dostępnych. Szansa na wystąpienie tej mało prawdopodobnej sytuacji wzrośnie w miarę zwiększania stopnia wieloprogramowości, gdy średnie wykorzystanie pamięci zbliży się do ilości pamięci dostępnej fizycznie. (Dlaczego w naszym przykładzie mamy poprzestawać na poziomie wieloprogramowania równym 6, skoro można go zwiększyć do siedmiu lub ośmiu?)

Nadprzydział objawi się w następujący sposób. Podczas wykonywania procesu użytkownika wystąpi brak strony. Sprzęt zaalarmuje system operacyjny, który sprawdzi swoje wewnętrzne tablice, aby rozstrzygnąć, że to jest brak strony, a nie próba niedozwolonego dostępu do pamięci. System operacyjny określi miejsce na dysku, gdzie przebywa potrzebna strona, po czym okaże się, że na liście wolnych ramek nie ma ani jednej wolnej ramki – cała pamięć jest w użyciu (rys. 9.5).



Rys. 9.5 Zapotrzebowanie na zastąpienie strony

W tej sytuacji system operacyjny ma kilka możliwości. Mogliby zakończyć proces użytkownika. Jednak stronicowanie na żądanie ma być tym, co system operacyjny wykonuje w celu polepszenia wykorzystania systemu komputerowego oraz zwiększenia jego przepustowości. Użytkownicy nie powinni być świadomi, że ich procesy są wykonywane w systemie stronicowanym. Dla użytkownika stronicowanie powinno być logicznie przezroczyste. Zakończenie procesu nie byłoby więc najlepszym wyborem.

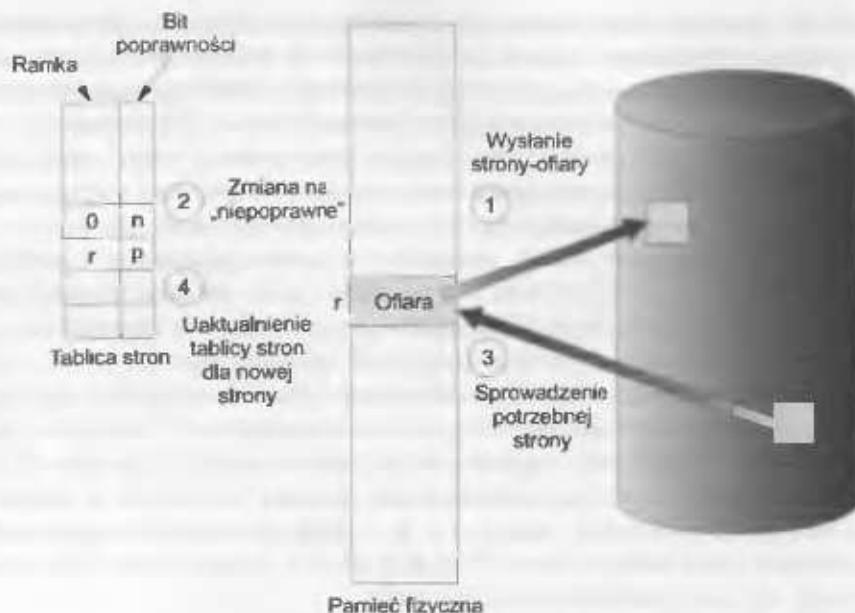
Można by wymienić proces, zwalniając wszystkie jego ramki i zmniejszając poziom wieloprogramowości. Ten pomysł jest niekiedy do przyjęcia i zajmiemy się nim dalej w p. 9.7, najpierw jednak omówimy bardziej frapującą możliwość: *zastępowanie stron* (ang. *page replacement*).

Zasada zastępowania stron jest następująca. Jeżeli wszystkie ramki są zajęte, to znajduje się taka, która nie jest właśnie używana i zwalnia się ją. Ramkę można zwolnić przez zapisanie jej zawartości na dysku i zmianę tablicy stron (i innych tablic) w celu wskazania, że strona nie jest już w pamięci (rys. 9.6). Zwolnioną ramkę można użyć do wprowadzenia strony, której brak spowodował przerwanie procesu. Procedura obsługi braków stron ulega modyfikacji, aby uwzględnić zastępowanie stron:

1. Zlokalizowanie potrzebnej strony na dysku.
2. Odnalezienie wolnej ramki:
 - (a) jeśli istnieje wolna ramka, to zostanie uzyta;
 - (b) w przeciwnym razie trzeba wykonać algorytm zastępowania stron w celu wytypowania *ramki-ofiary* (ang. *victim frame*);
 - (c) stronę-ofiarę zapisuje się na dysku; towarzyszy temu stosowna zmiana tablic stron i ramek.
3. Wczytanie potrzebnej strony do (świeżej) zwolnionej ramki i zmiana tablicy stron i ramek.
4. Wznowienie działania procesu.

Zauważmy, że gdy nie ma wolnych ramek, wówczas jest potrzebne dwukrotne przesyłanie stron (jedno na dysk i jedno z dysku). Powoduje to dwukrotne wydłużenie czasu obsługi braku strony, co w konsekwencji wydłuża efektywny czas dostępu.

Nakład ten można zmniejszyć przez zastosowanie *bitu modyfikacji*, nazywanego też *bitem zabrudzenia* (ang. *modify (dirty) bit*). Każda strona lub rama może być wyposażona w sprzętowy bit modyfikacji. Bit ten jest ustalany sprzętowo jako równy 1 przy zapisie dowolnego bajta lub słowa na



Rys. 9.6 Zastępowanie strony

stronie, aby wskazywał, że dana strona uległa zmianie. Przy wyborze strony do zastąpienia sprawdza się jej bit modyfikacji. Jeśli bit jest ustawiony, to wiadomo, że stan danej strony zmienił się od czasu jej przeczytania z dysku. W tym wypadku stronę należy zapisać na dysku. Natomiast nie ustawiony (wyzerowany) bit modyfikacji świadczy o tym, że zawartość strony nie była zmieniana od czasu przeczytania jej do pamięci. Jeśli więc kopia strony na dysku nie została zniszczona (np. przez jakąś inną stronę), to można uniknąć przesyłania takiej strony na dysk, ponieważ ona już się tam znajduje. Tę technikę stosuje się także do stron udostępnianych wyłącznie do czytania (np. stron kodu binarnego). Stron takich nie wolno zmieniać, więc w razie konieczności można się ich pozbyć. Takie postępowanie ze stronami może znacznie zmniejszyć czas obsługi braku strony, ponieważ zmniejsza się wówczas o połowę czas operacji wejścia-wyjścia – *pod warunkiem, że strona nie została zmieniona*.

Zastępowanie stron jest podstawą stronicowania na żądanie. Dopełnia ono rozdzielenia pamięci logicznej od pamięci fizycznej. Z pomocą tego mechanizmu programiści otrzymują wielką pamięć wirtualną przy użyciu mniejszej pamięci fizycznej. W stronicowaniu bez zgłoszania żądań adresy użytkownika były odwzorowywane na adresy fizyczne; dzięki temu oba zbiory

adresów były zupełnie różne. Wszystkie strony procesu musiały jednak stale znajdować się w pamięci fizycznej. W stronicowaniu na żądanie rozmiar przestrzeni adresów logicznych przestaje być ograniczany przez pamięć fizyczną. Na przykład dwudziestostronicowy proces użytkownika może być z powodzeniem wykonywany w 10 ramkach przy użyciu stronicowania na żądanie i algorytmu zastępowania w celu odnajdywania, w razie potrzeby, wolnej ramki. Zmieniona strona, która ma ulec zastąpieniu, jest kopiwana na dysk. Późniejsze odniesienie do tej strony spowoduje wykrycie jej braku. Wówczas zostanie ona z powrotem sprowadzona do pamięci, być może za cenę zastąpienia innej strony w procesie.

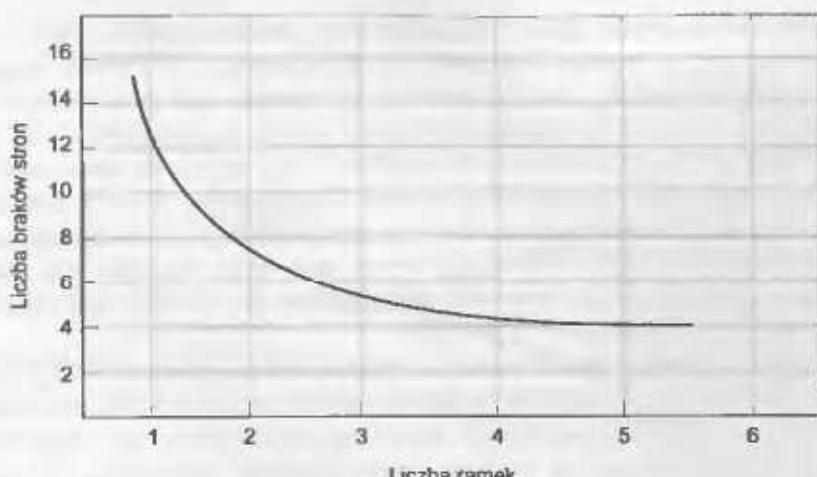
Aby zrealizować stronicowanie na żądanie, należy rozwiązać dwa główne problemy: opracować *algorytm przydzielu ramek* (ang. *frame-allocation algorithm*) oraz *algorytm zastępowania stron* (ang. *page-replacement algorithm*). Jeśli w pamięci znajduje się wiele procesów, to należy zdecydować, ile ramek zostanie przydzielonych do każdego procesu. Następnie, gdy powstanie konieczność zastąpienia strony, trzeba umieć wskazać ramkę do wymiany. Projektowanie odpowiednich algorytmów rozwiązujących te problemy jest ważnym zadaniem, gdyż operacje dyskowe są bardzo czasochłonne. Nawet niewielkie ulepszenie metody stronicowania na żądanie przynosi znaczny poprawę działania systemu.

9.5 ■ Algorytmy zastępowania stron

Istnieje wiele różnych algorytmów zastępowania stron. Bez mała każdy system operacyjny ma własny, unikatowy schemat zastępowania. Jak wybrać konkretny algorytm zastępowania? Zależy nam przede wszystkim na takim, który minimalizuje częstotliwość braków stron (ang. *page-fault rate*).

Algorytm ocenia się na podstawie wykonania go na pewnym ciągu odniesień do pamięci i zsumowania liczby braków stron. *Ciąg odniesień* (ang. *reference string*) można wytworzyć sztucznie (np. za pomocą generatora liczb losowych) lub na podstawie śledzenia danego systemu i zapisywania adresu każdego odwołania do pamięci. Ta druga metoda powoduje powstanie dużej liczby danych (rzędu milionów adresów na sekundę). Aby zredukować liczbę danych, odnotujmy dwa fakty.

Po pierwsze, dla danego rozmiaru strony (a rozmiar strony jest na ogół ustalony przez sprzęt lub system) musimy brać pod uwagę tylko numer strony, a nie cały adres. Po drugie, odwołanie do strony s , następujące bezpośrednio po odwołaniu do strony s , nigdy nie spowoduje braku strony. Strona s będzie w pamięci po pierwszym odniesieniu; następujące bezpośrednio po nim odniesienie nie spowoduje błędu.



Rys. 9.7 Wykres zależności braków stron od liczby ramek

Na przykład, śledząc pewien proces, moglibyśmy zanotować następujący ciąg adresów:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

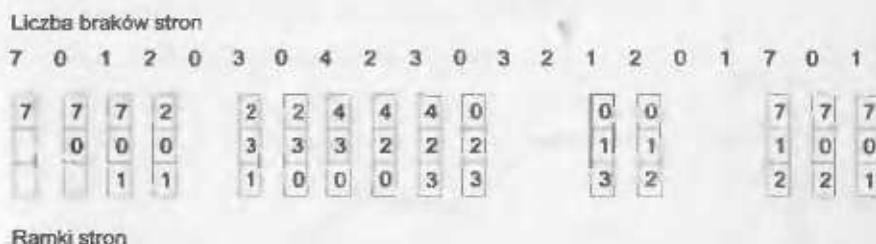
który – przy 100-bajtowej stronie – można zredukować do następującego ciągu odniesień:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Aby określić liczbę braków stron dla konkretnego ciągu odniesień i algorytmu zastępowania stron, należy także znać liczbę dostępnych ramek. Oczywiście, przy wzroście liczby wolnych ramek liczba braków stron będzie maleć. Jeśli na przykład mielibyśmy co najmniej 3 ramki, to dla powyższego ciągu odniesień wystąpiłyby tylko 3 braki stron – po jednym przy pierwszym odwołaniu do każdej strony. Gdyby natomiast istniała tylko jedna wolna rama, wtedy zastępowanie byłoby konieczne przy każdym odniesieniu, powodując 11 błędów. Na ogół oczekujemy zależności, którą widać na rys. 9.7. Wraz ze wzrostem liczby ramek maleje liczba braków stron do pewnego minimalnego poziomu.

Do ilustrowania algorytmów zastępowania stron będziemy w naszych rozważaniach używać następującego ciągu odniesień do pamięci z trzema ramkami:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



Rys. 9.8 Algorytm FIFO zastępowania stron

9.5.1 Algorytm FIFO

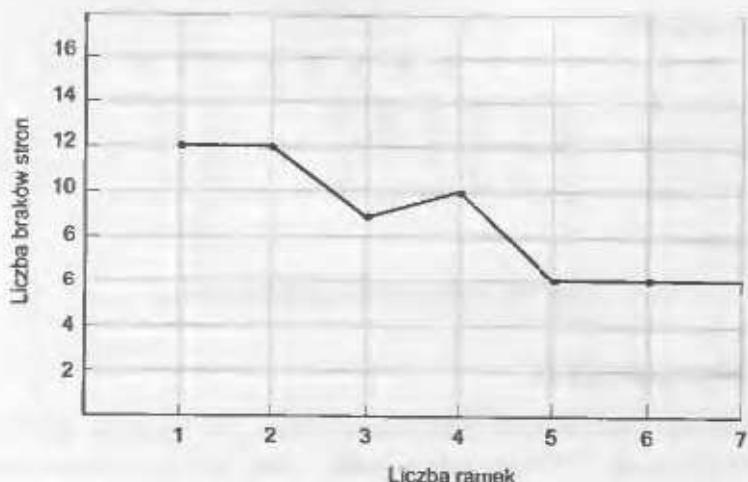
Najprostszym z algorytmów zastępowania stron jest algorytm FIFO*. Algorytm zastępowania FIFO kojarzy z każdą stroną jej czas wprowadzenia do pamięci. Kiedy trzeba zastąpić stronę, wtedy wybiera się stronę najstarszą. Zauważmy, że nie ma niezbędnej potrzeby odnotowywania czasu sprowadzenia strony. Można utworzyć kolejkę FIFO przechowującą wszystkie strony przebywające w pamięci. Do zastąpienia będzie delegowana strona z czoła kolejki. Strona wprowadzana do pamięci zostanie ulokowana na końcu kolejki.

Dla naszego przykładowego ciągu odniesień trzy ramki są początkowo puste. Pierwsze trzy odwołania (7, 0, 1) z powodu braków stron będą wymagały sprowadzenia odpowiednich stron do trzech pustych ramek. Następne odwołanie (2) spowoduje zastąpienie strony 7, gdyż strona 7 została wprowadzona jako pierwsza. Ponieważ kolejnym odniesieniem jest 0, a strona 0 jest już w pamięci, nie spowoduje ono błędu. Pierwsze odwołanie do strony 3 spowoduje zastąpienie strony 0, gdyż była ona pierwszą z trzech stron wprowadzonych do pamięci w ciągu (0, 1 i 2). To zastąpienie oznacza, że następne odniesienie do strony 0 spowoduje błąd. Wówczas strona 0 wejdzie na miejsce strony 1. Postępowanie to będzie kontynuowane, tak jak jest to pokazane na rys. 9.8. Przy każdym wystąpieniu braku pokazujemy, które strony znajdują się w naszych trzech ramkach. Liczba braków wyniesie łącznie 15.

Algorytm zastępowania FIFO jest łatwy do zrozumienia i zaprogramowania. Jednak jego zachowanie nie zawsze jest zadowalające. Strona zastępowana może zawierać wstępny moduł procesu, dawno używany i już niepotrzebny. Może jednak też zawierać zmienną będącą ciągle w użyciu, której wcześniej nadano wartość początkową.

Zauważmy także, że nawet po wybraniu do zastąpienia strony, która jest właśnie używana, wszystko działa poprawnie. Po wysłaniu z pamięci aktyw-

* Czyli „pierwszy na wejściu – pierwszy na wyjściu” (ang. *first-in first-out*). – Przyp. tłum.



Rys. 9.9 Krzywa braków stron dla zastępowania stron według algorytmu FIFO.
sporządzona na podstawie ciągu odniesień

nej strony w celu sprowadzenia nowej prawie natychmiast z powodu jej braku zostanie ona zamówiona na nowo. Aby sprowadzić z powrotem aktywną stronę do pamięci, trzeba będzie usunąć jakąś inną. Tym samym zły wybór przy zastępowaniu zwiększa liczbę braków stron i spowalnia wykonanie procesu, choć nie powoduje niepoprawnego działania.

Aby zilustrować kłopoty, które mogą się pojawiać podczas korzystania z algorytmu zastępowania FIFO, rozpatrzymy ciąg odniesień:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Na rysunku 9.9 jest pokazana krzywa braków stron w zależności od liczby dostępnych ramek. Zwraca uwagę fakt, iż liczba braków stron dla czterech ramek (10) jest większa od liczby braków stron dla trzech ramek (9)! Rezultat ten jest całkiem nieoczekiwany i nosi nazwę *anomalii Belady'ego* (ang. *Belady's anomaly*). Anomalia Belady'ego odzwierciedla fakt, że w niektórych algorytmach zastępowania stron współczynnik braków stron może wzrastać ze wzrostem wolnych ramek. Oczekiwaliśmy raczej, że zwiększenie pamięci procesu powinno poprawić jego zachowanie. Dość wcześnie zbadano, że założenie to nie zawsze jest prawdziwe. Wynikiem tych badań było odkrycie anomalii Belady'ego.

9.5.2 Algorytm optymalny

Jednym z następstw odkrycia anomalii Belady'ego było poszukiwanie optymalnego algorytmu zastępowania stron. Optymalny algorytm zastępowania stron cechuje najniższy współczynnik braków stron ze wszystkich algoryt-

Ciąg odniesień																			
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	0	0
0	0	0	0	0	0	0	4	0	0	0	0	0	1	0	0	0	0	0	0
1	1	1	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1

Ramki stron

Rys. 9.10 Optymalny algorytm zastępowania stron

mów. Algorytm optymalny nigdy nie jest dotknięty anomalią Belady'ego. Optymalny algorytm zastępowania stron istnieje i jest nazywany OPT lub MIN. Brzmi on po prostu:

Zastąp tę stronę, która najdłużej nie będzie używana.

Zastosowanie takiego algorytmu zastępowania stron gwarantuje najmniejszą z możliwych częstotliwość braków stron dla danej liczby ramek.

Na przykład dla naszego roboczego ciągu odniesień liczba braków stron przy zastosowaniu algorytmu optymalnego wynosi dziewięć, co widać na rys. 9.10. Pierwsze trzy odniesienia powodują wystąpienie braków stron, w wyniku których ulegają zapelnieniu trzy wolne ramki. Odwołanie do strony 2 powoduje zastąpienie strony 7, ponieważ strona 7 nie zostanie użyta aż do osiemnastego odwołania, podczas gdy strona 0 będzie użyta w piątym, a strona 1 – w czternastym odwołaniu. Odwołanie do strony 3 powoduje zastąpienie strony 1, gdyż strona 1 będzie ostatnią z aktualnie przebywających w pamięci stron, do których nastąpi powtórne odniesienie. Zaledwie dziewięcioma brakami stron zastępowanie optymalne jest znacznie lepsze od algorytmu FIFO, który powoduje 15 braków. (Jeśli pominać trzy pierwsze odwołania, które muszą wystąpić w każdym algorytmie, to zastępowanie optymalne okaże się dwukrotnie lepsze* od zastępowania metodą FIFO). Rzeczywiście, żaden inny algorytm zastępowania nie przetworzy w trzech ramkach danego ciągu odniesień z liczbą braków mniejszą niż dziewięć.

Niestety, optymalny algorytm zastępowania jest trudny do realizacji, ponieważ wymaga wiedzy o przyszłej postaci ciągu odniesień. (Podobną sytuację napotkaliśmy przy planowaniu procesora metodą SJF – „najpierw najkrótsze zadanie”, opisaną w p. 5.3.2). W rezultacie algorytm optymalny jest używany głównie w studiach porównawczych. Na przykład znajomość tego, że jakiś algorytm nie jest wprawdzie optymalny, lecz w najgorszym razie odbiega od optymalnego o 12,3%, a średnio jest od niego gorszy o 4,7% może okazać się dość cenna.

* W tym przykładzie. – Przyp. tłum.

9.5.3 Algorytm LRU

Jeśli algorytm optymalny jest nieosiągalny, to być może da się uzyskać jego przybliżenie. Podstawowa różnica między algorytmami FIFO a OPT (poza patrzeniem w przód lub wstecz) polega na tym, że w FIFO występuje czas po sprowadzeniu strony do pamięci, a w OPT czas, w którym strona ma być użyta. Jeśli do oszacowania najbliższej przyszłości używamy niedawnej przeszłości, to zastępujemy stronę, która nie była używana od najdłuższego czasu (rys. 9.11). Algorytm taki zwie się *zastępowaniem najdawniej używanych stron* (ang. *least recently used* – LRU).

Zastępowanie LRU kojarzy z każdą stroną czas jej ostatniego użycia. Gdy strona musi być zastąpiona inną, wówczas za pomocą algorytmu LRU wybiera się tę, która nie była używana od najdłuższego czasu. Jest to postępowanie według algorytmu optymalnego, jednak sięgające wstecz osi czasu, a nie w przód. (Zdziwić może, że jeśli przyjąć, że S'' jest odwróceniem ciągu odniesień S , to częstość braków stron wynikająca z algorytmu OPT wykonanego na ciągu S jest taka sama jak częstość braków stron dla algorytmu OPT zastosowanego do S'' . I podobnie – częstość braków stron w algorytmie LRU wykonanym na ciągu S jest taka sama jak częstość braków stron dla algorytmu LRU odniesionego do ciągu S'').

Wynik zastosowania algorytmu LRU do naszego przykładowego ciągu odniesień jest przedstawiony na rys. 9.11. Algorytm LRU powoduje 12 błędów. Zauważmy, że na początku mamy pięć takich samych braków jak przy zastępowaniu optymalnym. Jednak przy odwołaniu do strony 4 algorytm LRU rozstrzyga, że z trzech stron pozostających w pamięci strona 2 była używana najdawniej. Ostatnio używana była strona 0, a bezpośrednio przed nią – strona 3. Zatem algorytm LRU zastąpi stronę 2, nie mając danych o tym, że będzie ona za chwilę potrzebna. Gdy potem wystąpi brak strony 2, wówczas algorytm LRU zastąpi stronę 3, gdyż ze zbioru trzech stron w pamięci $\{0, 3, 4\}$, ona właśnie była używana najdawniej. Pomimo tych problemów zastępowanie metodą LRU z dwunastoma błędami jest znacznie lepsze niż zastępowanie FIFO, powodujące 15 błędów.

Liczba braków stron

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	4	4	4	0		1		1		1				
0	0	0		0	0	0	3	3		3	3	2	2	2	2	0	0	0	
1	1		3	3	2	2	2									7			

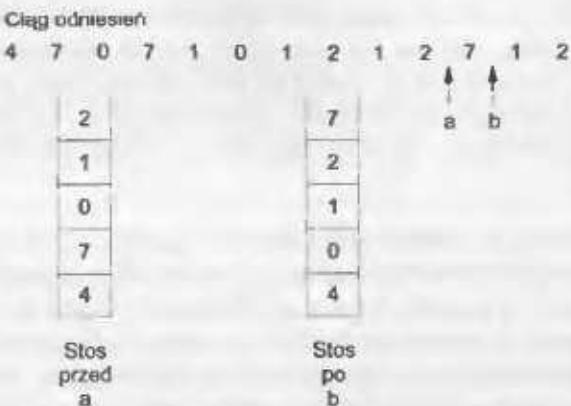
Ramki stron

Rys. 9.11 Algorytm LRU zastępowanie stron

Metodę LRU stosuje się często jako algorytm zastępowania stron i uważa się ją za dość dobrą. Główną trudnością jest sposób implementacji zastępowania według kolejności LRU. Algorytm LRU zastępowania stron może wymagać sporego zaplecza sprzętowego. Kłopot sprawia określenie porządku ramek na podstawie czasu ich ostatniego użycia. W praktyce stosuje się dwie implementacje:

- **Liczniki:** W najprostszym przypadku do każdej pozycji w tablicy stron dodajemy rejestr czasu użycia, do procesora zaś dodajemy zegar logiczny lub licznik. Wskazania zegara są zwiększone wraz z każdym odniesieniem do pamięci. Ilekroć występuje odniesienie do pamięci, tylekroć wartość rejestru zegara jest kopiwana do rejestru czasu użycia należącego do danej strony w tablicy stron. W ten sposób dla każdej strony dysponujemy zawsze „czasem” ostatniego do niej odniesienia. Zastępujemy stronę z najmniejszą wartością czasu. Schemat ten wymaga przeglądania tablicy stron w celu znalezienia strony zasługującej na miano najdawniej używanej (czyli LRU), jak również zapisywania w pamięci (pole czasu użycia w tablicy stron). Rejestry czasu użycia wymagają aktualnienia również przy wymianach tablic stron (powodowanych planowaniem przydziału procesora). Należy liczyć się także z możliwością powstania nadmiaru w rejestrze zegara.
- **Stos:** Inne podejście do realizacji algorytmu zastępowania LRU polega na utrzymywaniu stosu numerów stron. Przy każdym odwołaniu do strony jej numer wyjmuję się ze stosu i umieszcza na jego szczycie. W ten sposób na szczycie stosu jest zawsze strona użytą ostatnio, na spodzie zaś są strony najdawniej używane (rys. 9.12). Ponieważ trzeba wydobywać pozycje z wnętrza stosu, najlepsza implementacja polega na zastosowaniu dwukierunkowej listy ze wskaźnikami do czoła i do końca listy. Wyjęcie strony i umieszczenie jej na szczycie stosu wymaga w najgorszym razie zmiany szesziu wskaźników. Każde aktualnienie listy jest zatem nieco bardziej czasochłonne, ale do zastąpienia strony nie jest potrzebne przeszukiwanie listy, ponieważ wskaźnik końcowy listy, wskazujący na dno stosu, identyfikuje najdawniej używaną stronę. Opisana metoda jest szczególnie przydatna do implementacji zastępowania metodą LRU za pomocą oprogramowania systemowego lub mikroprogramu.

Ani zastępowanie optymalne, ani zastępowanie metodą LRU nie są dotknięte anomalią Belady’ego. Istnieje klasa algorytmów zastępowania stron, zwana *algorytmami stosowymi* (ang. *stack algorithms*), która nigdy nie wykazuje anomalii Belady’ego. Algorytmem stosowym nazywa się taki algorytm, dla którego można wykazać, że zbiór stron w pamięci w przypadku istnienia



Rys. 9.12 Użycie stosu do zapisywania ostatnich odwołań do stron

n ramek jest zawsze podzbiorem zbioru stron znajdującego się w pamięci przy $n + 1$ ramkach. W zastępowaniu metodą LRU zbiór stron w pamięci będzie się składał z n ostatnio używanych stron. Jeśli liczba ramek ulegnie zwiększeniu, to te n stron będzie nadal tworzyć zbiór ostatnio używanych, więc wciąż będzie pozostawać w pamięci.

Należy zauważyc, że żadna implementacja metody LRU nie jest do pomyślenia bez pomocy sprzętu bogatszego niż standardowe rejestrasy asocjacyjne. Uaktualnienie pól zegarowych lub stosu musi być wykonane przy każdym odniesieniu do pamięci. Powodowanie przy każdym odwołaniu do pamięci przerwania, aby za pomocą oprogramowania uaktualniać te struktury danych, spowodowałoby co najmniej 10-krotne spowolnienie kontaktów z pamięcią, a zatem 10-krotne spowolnienie każdego procesu użytkownika. Na takie koszty zarządzania pamięcią można by sobie pozwolić tylko w nielicznych systemach.

9.5.4 Algorytmy przybliżające metodę LRU

W niewielu systemach istnieje odpowiedni sprzęt do realizacji prawdziwego algorytmu LRU. Niektóre systemy nie mają żadnego takiego wyposażenia sprzętowego, co powoduje konieczność użycia innych algorytmów (w rodzaju FIFO). W wielu systemach są stosowane pewne ograniczone środki wspomagające – bity odniesienia (ang. *reference bit*). Bit odniesienia jest ustawiany dla danej strony przez sprzęt zawsze wtedy, gdy występuje do niej odniesienie (czyli czytanie lub zapisanie dowolnego bajta na stronie). Bity odniesienia są związane ze wszystkimi pozycjami tablicy stron.

Na początku wszystkie bity odniesienia są zerowane przez system operacyjny. W trakcie wykonywania procesu użytkownika każdy bit związany ze

stroną, do której następuje odwołanie, jest ustawiany (przyjmuje wartość 1) przez sprzęt. Po pewnym czasie można określić, które strony były używane, a które nie, na podstawie sprawdzenia wartości bitów odniesienia. Nic można w ten sposób poznać porządku użycia stron, ale wiadomo, które strony zostały użyte oraz do których się nie odwoływano. Informacja o takim częściowym porządku służy do budowy wielu algorytmów zastępowania stron przybliżających zastępowanie metodą LRU.

9.5.4.1 Algorytm dodatkowych bitów odwołań

Przez odnotowywanie stanu bitów odniesienia w regularnych odstępach czasu można pozyskać dodatkowe informacje. Dla każdej strony można przeznaczyć 8-bitowy bajt w tablicy w pamięci operacyjnej. W regularnych okresach (powiedzmy, co 100 ms) przerwanie zegarowe powoduje przekazanie sterowania do systemu operacyjnego. Dla każdej strony system operacyjny wprowadza bit odniesienia na najbardziej znaczącą pozycję jej 8-bitowego bajta, przesuwając pozostałe bity o jedną pozycję w prawo, z utratą bitu na pozycji najmniej znaczącej. Takie 8-bitowe rejestrysty przesuwające zawierają historię użycia strony w ostatnich osmiu okresach. Jeśli rejestr przesuwający zawiera wartość 00000000, to odpowiadająca mu strona nie była używana przez osiem okresów*. Strona, której przynajmniej jednokrotnie użyto w każdym z tych okresów, będzie miała w rejestrze przesuwającym wartość 11111111.

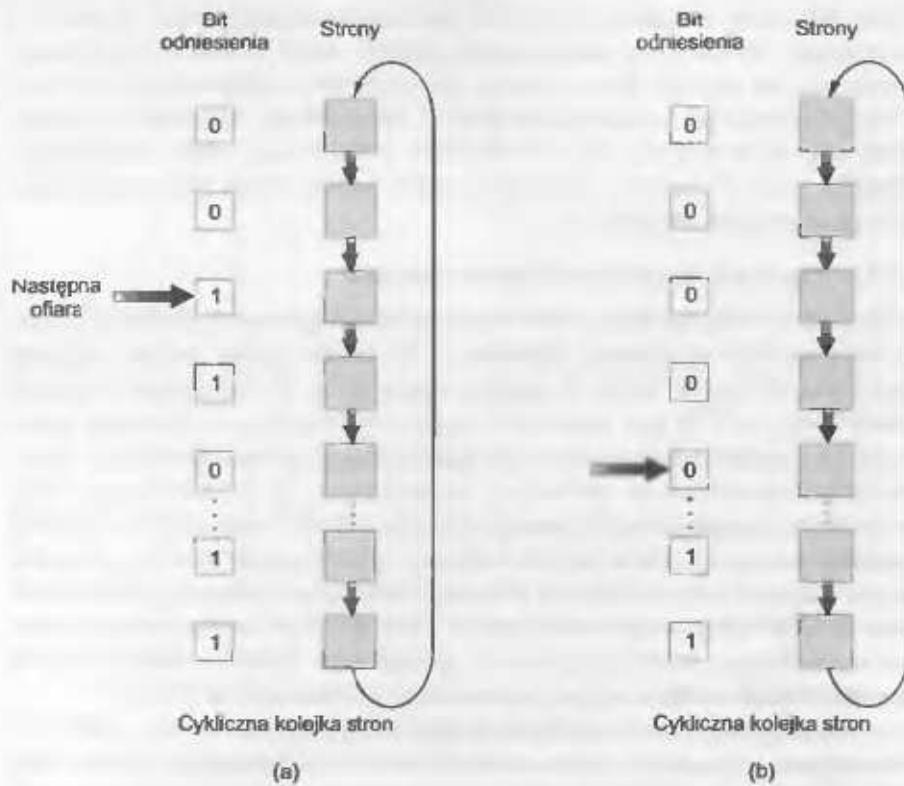
Strona, której historia w rejestrze przesuwającym ma postać 11000100, była używana później niż strona o historii 01110111. Interpretując omawiane 8-bitowe bajty jako liczby bez znaku, za stronę najdawniej używaną i kandydującą do zastąpienia przyjmuje się stronę, której odpowiada najmniejsza liczba. Zauważmy, że nie ma gwarancji na to, że takie liczby będą parami różne. Można zatem zastąpić (wymienić) albo wszystkie strony, którym odpowiadają najmniejsze wartości, albo do wyboru spośród nich zastosować algorytm FIFO.

Liczba bitów historii stron może, oczywiście, być różna i dobrana tak, by – w zależności od posiadanego sprzętu – możliwe przyspieszyć aktualnianie. W skrajnym przypadku liczba ta może zostać zredukowana do zera, pozostaje więc wtedy tylko sam bit odniesienia. Algorytm tego rodzaju nazywa się algorymem zastępowania *stron na zasadzie drugiej szansy*.

9.5.4.2 Algorytm drugiej szansy

Podstawą algorytmu zastępowania na zasadzie drugiej szansy jest algorytm FIFO. Po wybraniu strony sprawdza się dodatkowo jej bit odniesienia. Jeśli wartość bitu wynosi 0, to strona zostaje zastąpiona. Natomiast gdy bit odnie-

* Co najmniej. – Przyp. tłum.



Rys. 9.13 Algorytm zastępowania stron na zasadzie drugiej szansy (zegarowy)

sienia jest równy 1, dana strona dostaje drugą szansę*, a pod uwagę bierze się następną stronę wynikającą z porządku FIFO. Bit odniesienia strony, której daje się drugą szansę, jest zerowany, a czas jej przybycia ustawia się na czas bieżący. W ten sposób strona, której dano drugą szansę, nie będzie uwzględniona przy zastępowaniu dopóty, dopóki nie zostaną zastąpione inne strony (lub otrzymają drugą szansę). Co więcej, jeśli strona jest eksploatowana wystarczająco często, by utrzymać swój bit odniesienia ustawiony na 1, to nigdy nie zostanie zastąpiona.

Jednym ze sposobów na zaimplementowanie algorytmu drugiej szansy (nazywanego też zegarowym) jest kolejka cykliczna. Stronę typowaną do zastąpienia w następnej kolejności pokazuje wskaźnik. Przy zapotrzebowaniu na ramkę wskaźnik przemieszcza się naprzód, aż zostanie znaleziona strona

* Na pobyt w pamięci. – Przyp. tłum.

z wyzerowanym bitem odniesienia. Podczas przemieszczania bity odniesienia są zerowane (rys. 9.13). W najgorszym przypadku, jeśli wszystkie bity odniesienia były ustawione, wskaźnik obiegnie całą kolejkę, dając każdej stronie drugą szansę. Przed wybrianiem następnej strony do zastąpienia zostaną wyzerowane wszystkie bity odniesienia. Gdy wszystkie bity odniesienia są ustawione, wówczas zastępowanie metodą drugiej szansy sprowadza się do zastępowania zgodnie z algorytmem FIFO.

9.5.4.3 Ulepszony algorytm drugiej szansy

Omówiony wyżej algorytm drugiej szansy można ulepszyć, biorąc pod uwagę zarówno bit odniesienia, jak i bit modyfikacji (zob. p. 9.4) i traktując je jako uporządkowaną parę. Te dwa bity umożliwiają nam rozpatrywanie następujących czterech klas:

1. (0,0) – nie używana ostatnio i nie zmieniona – najlepsza strona do zastąpienia.
2. (0,1) – ostatnio nie używana, ale zmieniona – nieco gorsza kandydatka, ponieważ stronę taką trzeba będzie zapisać na dysku przed zastąpieniem.
3. (1,0) – używana ostatnio, lecz czysta* – prawdopodobnie za chwilę będzie znów używana.
4. (1,1) – używana ostatnio i zmieniona – prawdopodobnie niedługo znów będzie potrzebna, w przypadku zastępowania należy ją najpierw wykopować z pamięci.

Gdy zachodzi potrzeba zastąpienia strony, każda ze stron znajduje się w jednej z tych czterech klas. Postępujemy tak samo jak w algorytmie zegarowym, lecz zamiast sprawdzać, czy bit odniesienia wskazanej przez nas strony ma wartość 1, sprawdzamy, do której z klas dana strona należy. Postępujemy pierwszą napotkaną stroną z najniższej, niepustej klasy. Zauważmy, że zanim znajdziemy stronę do zastąpienia, możemy być zmuszeni obejść kolejkę cykliczną kilka razy.

9.5.5 Algorytmy zliczające

Istnieje wiele innych algorytmów, których można użyć przy zastępowaniu stron. Można na przykład wprowadzić liczniki odwołań do każdej ze stron i postępować na jeden z dwu poniższych sposobów.

* To znaczy – nic zmieniona – Przyp. tłum.

- **Algorytm LFU:** Algorytm zastępowania strony *najrzadziej używanej* (ang. *least frequently used* – LFU) typuje do zastąpienia stronę o najmniejszej wartości licznika odwołań. Powodem takiego wyboru jest spostrzeżenie, że strona aktywnie użytkowana winna mieć duży licznik odwołań. Algorytm ten może być obarczony błędami wynikającymi z sytuacji, w której strona jest intensywnie użytkowana w początkowej fazie procesu, a potem przestaje być używana w ogóle. Z czasów intensywnego użytkowania ma ona wysoką wartość licznika i pozostaje w pamięci, choć nie jest dłużej potrzebna. Jednym z rozwiązań może tu być przesuwanie liczników w prawo o 1 bit w regularnych odstępach czasu, powodujące wykładnicze zmniejszanie wartości licznika przeciennego użycia.
- **Algorytm MFU:** Algorytm zastępowania strony *najczęściej używanej* (ang. *most frequently used* – MFU) uzasadnia się tym, że strona z najmniejszą wartością licznika została prawdopodobnie dopiero co sprowadzona i będzie jeszcze używana.

Jak można oczekiwać, oba algorytmy zastępowania, MFU oraz LFU, nie są popularne. Implementacja tych algorytmów jest dość kosztowna, a nie przybliżają one dość dobrze algorytmu OPT.

9.5.6 Algorytm buforowania stron

Jako uzupełnienie konkretnych algorytmów zastępowania stron stosuje się często różne procedury. Na przykład systemy często przechowują *pulę* wolnych ramek. Po wystąpieniu błędu strony ramkę-ofiarę wybiera się jak uprzednio. Niemniej jednak, zanim usunie się z pamięci stronę-ofiarę, potrzebną stronę czyta się do którejś z wolnych ramek puli. Postępowanie takie pozwala na maksymalnie szybkie wznowienie procesu, bez oczekiwania na przepisanie strony-ofiary do pamięci operacyjnej. Kiedy w późniejszym czasie strona-ofiara zostanie przepisana do pamięci zewnętrznej, wtedy zajmowaną przez nią ramkę dołączy się do puli wolnych ramek.

Rozwinięciem tego pomysłu jest utrzymywanie listy zmienionych stron. Gdy tylko urządzenie stronicujące jest bezczynne, wybiera się zmienioną stronę i zapisuje ją na dysku. Jej bit modyfikacji jest wówczas zerowany. W takim schemacie większe jest prawdopodobieństwo, że strona wybrana do zastąpienia będzie czysta i jej zawartości nie trzeba będzie zapisywać na dysku.

Inna odmiana tej metody polega na utrzymywaniu puli wolnych ramek i pamiętaniu, które strony rezydowały w każdej ramce. Ponieważ zawartość ramki nie ulega zmianie wskutek jej zapisania na dysku, więc póki dana ramka nie zostanie użyta na nowo, pół pozostająca w niej strona może być użyta

powtórnie wprost z puli wolnych ramek. W tym przypadku nie jest potrzebna żadna operacja wejścia-wyjścia. Po wystąpieniu braku strony sprawdza się najpierw, czy potrzebna strona jest w puli wolnych ramek. Jeśli jej tam nie ma, to należy wybrać wolną ramkę i przeczytać do niej brakującą stronę.

Opisaną technikę zastosowano w systemie VAX/VMS wraz z algorytmem zastępowania FIFO. Gdy w wyniku zastępowania według algorytmu FIFO zdarzy sięomyłkowe usunięcie strony, która jest właśnie używana, wówczas stronę tę szybko odzyskuje się z bufora wolnych ramek bez potrzeby wykonywania operacji wejścia-wyjścia. Bufor wolnych ramek chroni przed względnie kiepskim, ale prostym algorymem zastępowania FIFO. Metoda ta okazała się niezbędna, ponieważ we wczesnych wersjach komputerów VAX implementacja bitu odniesienia była niewłaściwa.

9.6 ■ Przydział ramek

Jak rozdzielać stałą ilość wolnej pamięci między różne procesy? Jeśli mamy 93 wolne ramki i dwa procesy, to ile ramek ma otrzymać każdy z nich?

Najprostszym przypadkiem pamięci wirtualnej jest system z jednym użytkownikiem. Rozważmy system mikrokomputerowy z jednym użytkownikiem, mający 128 KB pamięci złożonej ze stron wielkości 1 KB. Mamy więc 128 ramek. System operacyjny może zająć 35 KB, pozostawiając 93 ramki procesowi użytkownika. Przy czystym stronicowaniu na żądanie początkowo wszystkie 93 ramki trafiają na listę ramek wolnych. Gdy proces użytkownika rozpoczyna działanie, generuje ciąg braków stron. Pierwsze 93 braki stron zostaną skwitowane przez przydzielanie stron z listy wolnych ramek. Gdy lista wolnych ramek się wyczerpie, trzeba będzie zastosować algorytm zastępowania stron, aby spośród 93 stron pozostałych w pamięci wybrać jedną stronę, która zostanie zastąpiona przez dziewięćdziesiątą czwartą itd. Po zakończeniu działania procesu 93 ramki wróćą znów na listę wolnych ramek.

Ta prosta strategia może mieć wiele odmian. Można wymagać od systemu operacyjnego, aby wszystkie swoje bufore i miejsca na tablice przydzielali w obrębie listy pustych ramek. Gdy system operacyjny przestanie korzystać z tych obszarów, wówczas mogą się one przydać do stronicowania procesów użytkowych. Można spróbować rezerwować trzy wolne ramki na liście wolnych ramek przez cały czas. Jeśli wystąpi brak strony, to zawsze będzie wolna ramka do wynajęcia. Kiedy dochodzi do wymiany stron, wtedy można wybrać takie zastępowanie, aby zapisywanie strony na dysku odbywało się równocześnie z działaniem procesu użytkownika.

Możliwe są i inne warianty, ale podstawowa strategia jest przejrzysta: proces użytkownika dostaje dowolną z wolnych ramek.

Połączenie stronicowania na żądanie z wieloprogramowaniem jest przyczyną powstawania innych problemów. Wskutek wieloprogramowania w pamięci pojawiają się w tym samym czasie co najmniej dwa procesy.

9.6.1 Minimalna liczba ramek

Na przyjmowaną strategię przydzielu ramek nakłada się, co oczywiste, wiele ograniczeń. Nie można przydzielić więcej ramek niż wynosi łączna liczba dostępnych ramek (chyba że istnieje możliwość wspólnego użytkowania stron). Istnieje również minimalna liczba ramek, które powinny być przydzielone. Jest jasne, że wraz ze zmniejszaniem się liczby ramek przydzielonych do każdego procesu wzrasta częstość występowania braków stron, spowalniając wykonanie procesów.

Niezależnie od niekorzystnych objawów w działaniu systemu, w którym przydziela się za mało ramek, istnieje minimalna liczba ramek, które muszą być przydzielone. Ta minimalna liczba jest określona przez zbiór rozkazów w architekturze komputera. Zważmy, że jeśli brak strony wystąpi przed dokończeniem wykonania rozkazu, to rozkaz musi być powtórzony. Wynika z tego, że należy mieć wystarczającą liczbę ramek do przechowania wszystkich stron, do których może się odnosić pojedynczy rozkaz.

Weźmy na przykład pod uwagę maszynę, w której wszystkie rozkazy dotyczące się do pamięci mają tylko po jednym adresie. Potrzebujemy w związku z tym przynajmniej jednej ramki na rozkaz i jednej ramki na jego odniesienie się do pamięci. Na dodatek, jeśli dopuszcza się jednopoziomowe adresowanie pośrednie (np. rozkaz pobrania znajdujący się na stronie 16 może się odwoływać do adresu na stronie 0, pod którym jest pośrednie odwołanie do strony 23), to stronicowanie wymaga przynajmniej trzech stron na proces. Warto się zastanowić, co mogłoby się stać, gdyby proces miał tylko dwie ramki.

Minimalna liczba ramek jest zdefiniowana przez architekturę komputera. Na przykład rozkaz przesyłania w komputerze PDP-11 w niektórych trybach adresowania składa się z więcej niż jednego słowa, rozkaz ten może więc znaleźć się na dwóch sąsiednich stronach. Ponadto każdy z jego dwóch argumentów może być adresowany pośrednio, co stawia w pogotowiu łącznie sześć stron. Najgorszym przypadkiem w systemie IBM 370 jest prawdopodobnie rozkaz przemieszczania znaków (MVC). Ponieważ jest to rozkaz typu „z pamięci do pamięci”, zajmuje więc 6 bajtów i może znajdować się na dwóch sąsiednich stronach. Blok znaków przeznaczonych do przesłania i obszar, w którym należy je umieścić, również mogą leżeć na dwóch stronach. Sytuacja ta może powodować zapotrzebowanie na sześć ramek. (W rzeczywistości, w najgorszym przypadku, gdy rozkaz MVC jest argumentem rozkazu EXE-

CUTE (wykonaj) znajdującego się na dwu stronach, potrzeba aż ośmiu ramek).

Najgorsze przypadki takich scenariuszy występują w komputerach, których architektura dopuszcza wielokrotne poziomy adresowania pośredniego (np. 16-bitowe słowo może zawierać 15-bitowy adres oraz 1-bitowy wskaźnik adresowania pośredniego). Teoretycznie zwykły rozkaz pobrania może zawsze odniesienie do adresu pośredniego, które może powodować kolejne odniesienie do adresu pośredniego (na innej stronie), które też może powodować odniesienie do adresu pośredniego (na jeszcze innej stronie) itd., aż w końcu będzie on odnosić się do każdej strony w pamięci wirtualnej. Zatem w najgorszym przypadku cała pamięć wirtualna powinna być w pamięci fizycznej. Aby pokonać tę trudność, należy ograniczyć liczbę poziomów adresowania pośredniego (np. ograniczyć adresowanie pośrednie w rozkazie do 16 poziomów). Przy wystąpieniu pierwszego poziomu adresowania pośredniego ustawia się licznik na 16, który następnie zmniejsza się o 1 przy każdym kolejnym adresowaniu pośrednim danego rozkazu. Jeśli licznik osiągnie wartość 0, to wystąpi pulapka (przekroczenie ograniczenia adresowania pośredniego). Takie ograniczenie redukuje do 17 maksymalną liczbę odniesień do pamięci przypadających na jeden rozkaz, powodując tak samo ograniczone zapotrzebowanie na ramki.

Minimalna liczba ramek przypadających na proces jest zdefiniowana przez architekturę logiczną komputera, a maksymalna ich liczba wynika z ilości dostępnej pamięci fizycznej. W tych granicach pozostaje nam ciągle jeszcze wiele możliwości do wyboru.

9.6.2 Algorytmy przydziału

Najprostszym sposobem rozdzielenia m ramek między n procesów jest danie każdemu jednakowej porcji m/n ramek. Na przykład: jeśli mamy 93 ramki i 5 procesów, to każdy proces może dostać 18 ramek. Pozostałą nadwyżkę 3 ramek można potraktować jako bufor wolnych ramek. Schemat taki zwie się *przydziałem równym* (ang. *equal allocation*).

Można też podejść do tego inaczej – uwzględnić rozmaite zapotrzebowania procesów na ramki. Jeśli mały proces studencki zajmujący 10 KB pamięci oraz interakcyjna implementacja bazy danych wielkości 127 KB są wykonywane jako jedynie procesy w systemie z 62 wolnymi ramkami, to nie byłoby zbyt sensowne przydzielać obu procesom po 31 ramek. Proces studencki nie potrzebowałby więcej niż 10 ramek, zatem pozostałe 21 ramek marnowałoby się w oczywisty sposób.

Problem ten rozwiązuje się, stosując *przydział proporcjonalny* (ang. *proportional allocation*). Każdemu procesowi przydziela się dostępną pamięć

odpowiednio do jego rozmiaru. Niech s_i oznacza wielkość pamięci wirtualnej procesu p_i . Sumę pamięci wirtualnej wszystkich procesów określamy jako

$$S = \sum s_i$$

Wówczas, gdy ogólna liczba ramek wynosi m , procesowi p_i przydzielamy a_i ramek, przy czym a_i wynosi w przybliżeniu

$$a_i = s_i / S \times m$$

Oczywiście, wielkości a_i muszą być zaokrąglone do liczb całkowitych większych od minimalnej liczby ramek wymaganej przez zbiór rozkazów komputera, a ich suma nie może przekraczać m .

Stosując przydział proporcjonalny, 62 ramki dzielimy między proces z 10 stronami i proces z 127 stronami w ten sposób, że pierwszemu z nich dajemy 4 ramki, drugiemu zaś 57 ramek, co wynika z obliczenia

$$10/137 \times 62 \approx 4$$

$$27/137 \times 62 \approx 57$$

W ten sposób, zamiast po równo, oba procesy wykorzystują dostępne ramki zgodnie ze swymi „potrzebami”.

Tak w przydziale równym, jak i proporcjonalnym wielkość przydziału dla każdego procesu waha się, rzecz jasna, zależnie od stopnia wieloprogramowości. Przy wzroście poziomu wieloprogramowości każdy proces utraci nieco ramek, aby zapewnić pamięć potrzebną dla nowego procesu. Z kolei, gdy poziom wieloprogramowości obniży się, wtedy ramki, które należały do zakończonych procesów, można rozdzielić między pozostałe procesy.

Zauważmy, że w obu przydziałach: równym i proporcjonalnym proces o wyższym priorytecie jest traktowany tak samo jak proces niskopriorytetowy. Jednakże na mocy definicji mamy prawo wymagać, aby proces o wyższym priorytecie otrzymał więcej pamięci w celu przyspieszenia swojego działania – nawet za cenę pogorszenia działania procesów niskopriorytetowych.

Jedną z metod jest zastosowanie przydziału proporcjonalnego, w którym liczba ramek zależy nie od względnych rozmiarów procesów, lecz od priorytetów procesów albo od kombinacji rozmiaru i priorytetu.

9.6.3 Porównanie przydziału globalnego i lokalnego

Innym, ważnym aspektem przydzielania ramek różnym procesom jest zastępowanie stron. W przypadku mnogości procesów rywalizujących o ramki w pamięci algorytmy zastępowania stron można zakwalifikować do dwóch obszernych kategorii: zastępowania globalnego (ang. *global replacement*) i za-

zastępowania lokalnego (ang. *local replacement*). Zastępowanie globalne umożliwia procesom wybór ramki ze zbioru wszystkich ramek, nawet gdy ramka jest w danej chwili przydzielona do innego procesu – jeden proces może zabrać ramkę drugiemu procesowi. Zastępowanie lokalne ogranicza wybór do zbioru ramek przydzielonych do danego procesu.

Rozważmy na przykład schemat przydziału, w którym pozwalamy procesom wysokopriorytetowym wybierać ramki do zastępowania z obszarów procesów niskopriorytetowych. Proces może wyselekcyjnować ramkę do zastąpienia spośród własnych ramek lub ramek dowolnego procesu o niższym priorytecie. W takim podejściu proces o wyższym priorytecie może zwiększać liczbę swoich ramek kosztem procesu o niższym priorytecie.

W strategii zastępowania lokalnego liczba ramek przydzielonych do procesu nie zmienia się. Przy zastępowaniu globalnym może się natomiast zdarzyć, że proces będzie wybierał jedynie ramki przydzielone innym procesom, zwiększając liczbę przydzielonych mu ramek (przy założeniu, że inne procesy nie będą wybierały jego ramek do zastępowania).

W algorytmie zastępowania globalnego proces nie może kontrolować własnej częstotliwości występowania braków stron, co stanowi pewien kłopot. Zbiór stron procesu w pamięci zależy nie tylko od zachowania się danego procesu przy stronicowaniu, ale również od sposobu stronicowania innych procesów. Wskutek tego ten sam proces może zachowywać się zupełnie różnie (zajmując przy jednym wykonaniu 0,5 s, a przy następnym – 10,3 s); zależy to od warunków zewnętrznych. Nie dzieje się tak w przypadku algorytmu zastępowania lokalnego. Przy zastępowaniu lokalnym zbiór stron procesu w pamięci zależy tylko od stronicowania odnoszącego się do danego procesu. Ze względu na swą wycinkowość zastępowanie lokalne może hamować proces, czyniąc niedostępny dla niego inne, mniej używane strony pamięci. Toteż lepszą przepustowość systemu na ogół daje zastępowanie globalne i dlatego jest częściej stosowaną metodą.

9.7 ■ Szamotanie

Gdy liczba ramek przydzielonych do niskopriorytetowego procesu zmniejsza się poniżej minimum wymaganego przez architekturę komputera, wówczas wykonanie takiego procesu musi zostać zawieszone. Należy wtedy usunąć pozostałe strony procesu, zwalniając wszystkie ramki, które były mu przydzielone. Odpowiadają za to procedury wymiany (sprawdzania i wyprowadzania) ze średniego poziomu planowania przydziału procesora.

Rzeczywiście, przyjrzyjmy się dowolnemu procesowi, który nie ma „dość” ramek. Chociaż technicznie jest możliwe zmniejszenie liczby przy-

dzielonych ramek do minimum, zawsze będzie istniała pewna (większa) liczba stron aktywnie używanych. Jeśli proces nie będzie miał takiej liczby ramek, to szybko wystąpi brak strony. Wtedy któraś ze stron będzie musiała być zastąpiona. Ponieważ jednak wszystkie strony są aktywnie używane, trzeba więc będzie zastąpić jakąś stronę, która za chwilę okaże się potrzebna. W konsekwencji, w procesie bardzo szybko będą następowaly po sobie kolejne braki stron. Proces będzie ciągle wykazywał brak strony, wymieniając jakąś stronę, po czym – z powodu jej braku – sprowadzając ją z powrotem.

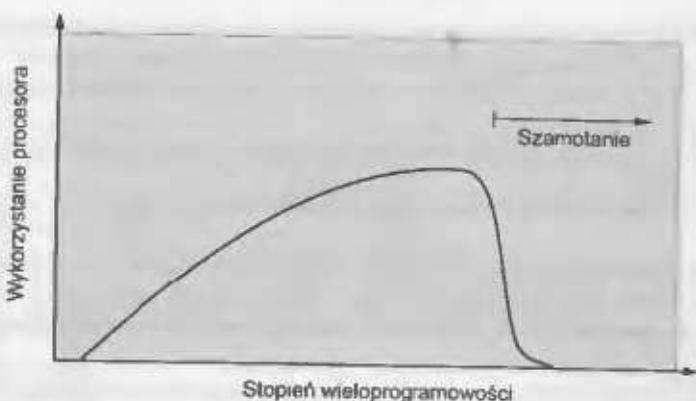
Taką bardzo dużą aktywność stronicowania określa się mianem *szamotania* (ang. *thrashing*). Proces szamocze się, jeśli spędza więcej czasu na stronicowaniu niż na wykonaniu.

9.7.1 Przyczyna szamotania

Szamotanie powoduje poważne zaburzenia wydajności. Rozważmy następujący scenariusz, oparty na rzeczywistym zachowaniu wczesnych systemów stronicujących.

System operacyjny nadzoruje wykorzystanie jednostki centralnej. Jeśli jest ono za małe, to zwiększa się stopień wieloprogramowości, wprowadzając nowy proces do systemu. Strony są zastępowane według globalnego algorytmu zastępowania stron, bez brania pod uwagę, do jakich procesów należą. Założymy teraz, że proces wchodzi w nową fazę działania i potrzebuje więcej ramek. Zaczyna wykazywać braki stron i powoduje utratę stron przez inne procesy. Te z kolei procesy potrzebują tych stron, więc wykazują ich braki i znów przyczyniają się do odbierania stron innym procesom. Tak zachowujące się procesy muszą używać urządzenia stronicującego w celu dokonywania wymiany stron. Ustawiają się w kolejce do urządzenia stronicującego, a jednocześnie opróżnia się kolejka procesów gotowych do wykonywania. Wskutek oczekiwania procesów na urządzenie stronicujące zmniejsza się wykorzystanie procesora.

Planista przydziału procesora dostrzega spadek wykorzystania procesora, więc zwiększa stopień wieloprogramowości. Nowy proces, próbując wystartować, zabiera ramki wykonywanym procesom, czym powoduje jeszcze więcej braków stron i jeszcze większą kolejkę do urządzenia stronicującego. W rezultacie wykorzystanie procesora spada jeszcze bardziej i planista przydziału procesora próbuje zwiększyć stopień wieloprogramowości po raz kolejny. Powstaje szamotanina, przepustowość systemu gwałtownie maleje i równie gwałtownie wzrasta częstość występowania braków stron. Wskutek tego wzrasta czas efektywnego dostępu do pamięci. Nie można zakończyć żadnej pracy, ponieważ procesy spędzają cały czas na stronicowaniu.



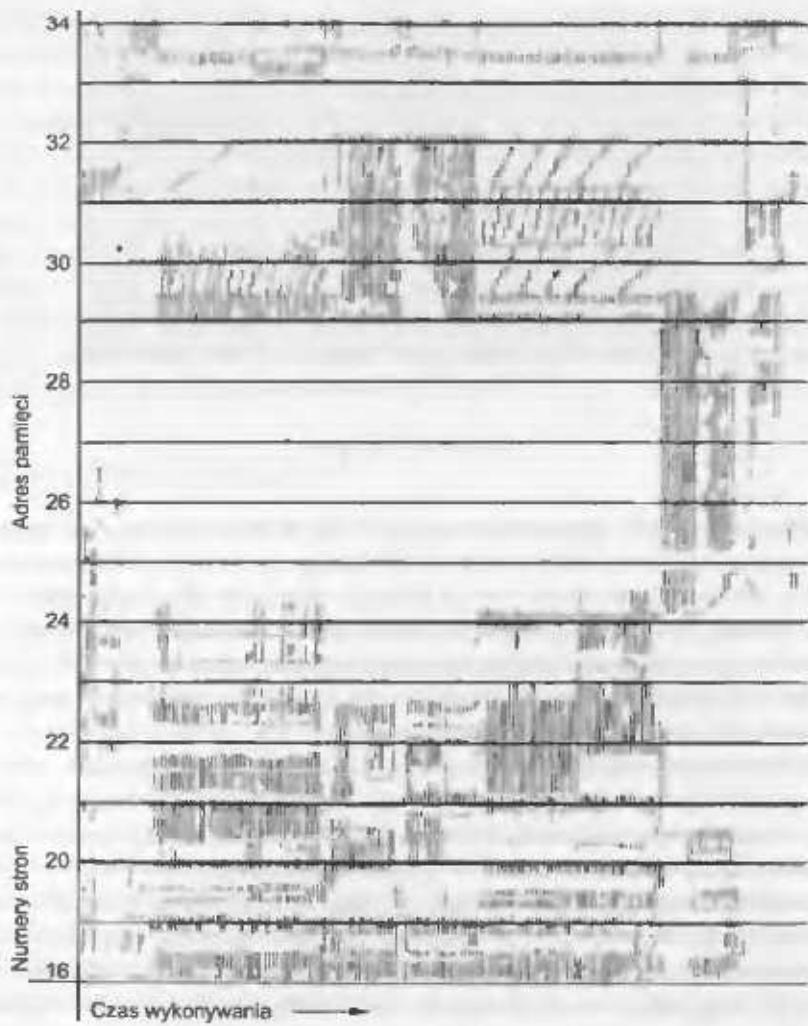
Rys. 9.14 Szamotanie

Zjawisko to jest zilustrowane na rys. 9.14. Wykreślono na nim wykorzystanie procesora jako funkcję stopnia wieloprogramowości. W miarę wzrostu stopnia wieloprogramowości wykorzystanie procesora również rośnie, choć coraz wolniej, aż osiąga maksimum. Dalsze zwiększanie wieloprogramowości prowadzi do szamotania i wykorzystanie procesora ostro maleje. W tej sytuacji, by zwiększyć wykorzystanie procesora i powstrzymać szamotanie, należy zmniejszyć stopień wieloprogramowości.

Efekt szamotania można ograniczyć za pomocą *lokalnego* (lub *priorytetowego*) (!) algorytmu zastępowania. Przy zastępowaniu lokalnym, gdy jakiś proces zaczyna się szamotać, wówczas nie wolno mu krasić ramek innego procesu i doprowadzać go także do szamotania. Zastępowaniu stron towarzyszy zwracanie uwagi na to, którego procesu są one częściami. Jeśli jednak jakieś procesy się szamocą, to będą one pozostawać przez większość czasu w kolejce do urządzenia stronicującego. Rośnie średni czas obsługi braku strony ze względu na wydłużanie się kolejki do urządzenia stronicującego. Wskutek tego czas efektywnego dostępu wzrasta nawet dla procesów, które się nie szamocą.

Aby zapobiec szamotaniu, należy dostarczyć procesowi tyle ramek, ile potrzebuje. Ale skąd możemy się dowiedzieć, ile ramek proces będzie „potrzebować”? Istnieje kilka sposobów. Przyjmując strategię tworzenia zbioru roboczego (zob. p. 9.7.2), rozpoczyna się od sprawdzenia, z ilu ramek proces w danej chwili korzysta. Podejście to określa model strefowy wykonania procesu.

Model strefowy (ang. *locality model*) zakłada, że w trakcie wykonania proces przechodzi z jednej strefy programu do innej. Przez strefę programu rozumie się zbiór stron pozostających we wspólnym użyciu (rys. 9.15).



Rys. 9.15 Strefy lokalne we wzorze odwołań do pamięci

Ujmując ogólnie, program składa się z wielu różnych stref, które mogą na siebie zachodzić.

Na przykład wywoływany podprogram definiuje nową strefę. W tej strefie odniesienia do pamięci dotyczą rozkazów danego podprogramu, jego zmiennych lokalnych i podzbioru zmiennych globalnych. Wychodząc z podprogramu, proces opuszcza daną strefę, gdyż zmienne lokalne i rozkazy podprogramu przestają być dalej aktywnie używane. Do strefy tej może później nastąpić powrót. Widać zatem, że strefy programu są określone przez jego

strukturę i struktury danych. W modelu strefowym przyjmuje się założenie, że wszystkie programy będą charakteryzować się taką podstawową strukturą odniesień do pamięci. Zwróciły uwagę na to, że model strefowy w niejawnny sposób daje podstawę temu, co powiedzieliśmy dotychczas w tej książce na temat pamięci podręcznych. Gdyby dostęp do danych dowolnego typu były losowe, nie dając się ująć w żaden szablon, to pamiętanie podręczne należałyby uznać za bezcelowe.

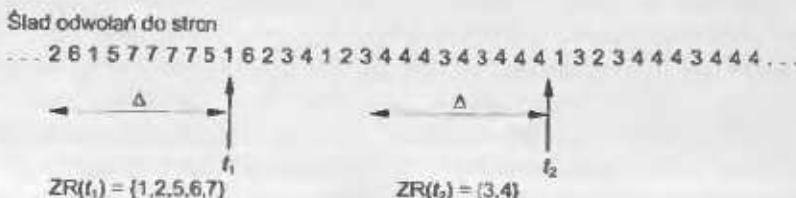
Załóżmy, że procesowi przydzielono dość ramek, aby mógł w nich pomieścić swoją bieżącą strefę. Zanim wszystkie strony danej strefy nie znajdą się w pamięci, będzie on wykazywał braki stron; następnie braki te zanikną, aż do chwili, gdy proces zmieni strefę. Gdyby przydzielono mniej ramek, niż wynosi rozmiar bieżącej strefy, wówczas proces zacząłby się szamotać, ponieważ nie mógłby utrzymać w pamięci tych wszystkich stron, których aktywnie używa.

9.7.2 Model zbioru roboczego

Model zbioru roboczego (ang. *working-set model*) opiera się na założeniu, że program ma charakterystykę strefową. W modelu tym używa się parametru Δ do definiowania *okna zbioru roboczego* (ang. *working-set window*). Pomyśl polega na sprawdzaniu ostatnich Δ odniesień do stron. Za *zbior roboczy* (ang. *working-set*) przyjmuje się zbiór stron, do których nastąpiło ostatnich Δ odniesień (rys. 9.16). Jeśli strona jest aktywnie używana, to będzie znajdować się w zbiorze roboczym. Gdy strona przestanie być używana, wówczas wpadnie ze zbioru roboczego po Δ jednostkach czasu odliczonych od ostatniego do niej odwołania. W ten sposób zbiór roboczy przybliża strefę programu.

Na przykład przy ciągu odniesień do pamięci takim jak na rys. 9.16, przyjawszy $\Delta = 10$ odniesień do pamięci, zbiór roboczy w chwili t_1 będzie równy $\{1, 2, 5, 6, 7\}$. Do chwili t_2 zbiór roboczy zmieni się na $\{3, 4\}$.

Dokładność zbioru roboczego zależy od wyboru parametru Δ . Jeśli parametr Δ będzie za mały, to nie obejmie całego zbioru roboczego; jeśli ten parametr będzie za duży, to może zachodzić na kilka stref programu. W skraj-



Rys. 9.16 Model zbioru roboczego

nym przypadku, gdy parametr Δ jest nieskończony, wtedy zbiorem roboczym staje się zbiór stron, z którymi kontaktowano się podczas wykonania procesu.

Najważniejszą cechą zbioru roboczego jest jego rozmiar. Jeśli dla każdego procesu w systemie obliczymy rozmiar zbioru roboczego RZR_i , to możemy określić ogólne zapotrzebowanie na ramki (Z) wzorem

$$Z = \sum RZR_i$$

Każdy proces używa aktywnie stron ze swojego zbioru roboczego. Proces i potrzebuje RZR_i ramek. Jeśli łączne zapotrzebowanie jest większe niż ogólna liczba dostępnych ramek ($Z > m$), to powstanie szamotanie, gdyż niektóre procesy nie otrzymają wystarczającej liczby ramek.

Zastosowanie modelu zbioru roboczego jest więc całkiem proste. System operacyjny dogląda zbioru roboczego w każdym procesie i przydziela każdemu procesowi tyle ramek, ile wymaga rozmiar jego zbioru roboczego. Jeśli są jeszcze dodatkowe ramki, to można rozpocząć nowy proces. Gdy suma rozmiarów zbiorów roboczych wzrasta i zaczyna przekraczać łączną liczbę dostępnych ramek, wówczas system operacyjny wybiera proces, którego wykonanie trzeba będzie wstrzymać. Strony procesu są usuwane z pamięci, a jego ramki przydziela się innym procesom. Wstrzymany proces może być później wznowiony.

Stosując strategię zbioru roboczego, zapobiega się szamotaniu i utrzymuje stopień wieloprogramowości na możliwie wysokim poziomie – czyli optymalizuje wykorzystanie procesora.

Utrudnieniem przy zastosowaniu modelu zbioru roboczego jest utrzymanie śladu zbioru roboczego. Okno zbioru roboczego jest ruchome. Przy każdym odniesieniu do pamięci na jednym końcu okna pojawia się nowe odniesienie, a z drugiego jego końca wypada odniesienie najstarsze. Strona należy do zbioru roboczego, jeśli występuje gdziekolwiek w oknie roboczym. Model zbioru roboczego można przybliżać za pomocą zegara generującego przerwania w stałych odstępach czasu i bitu odniesienia.

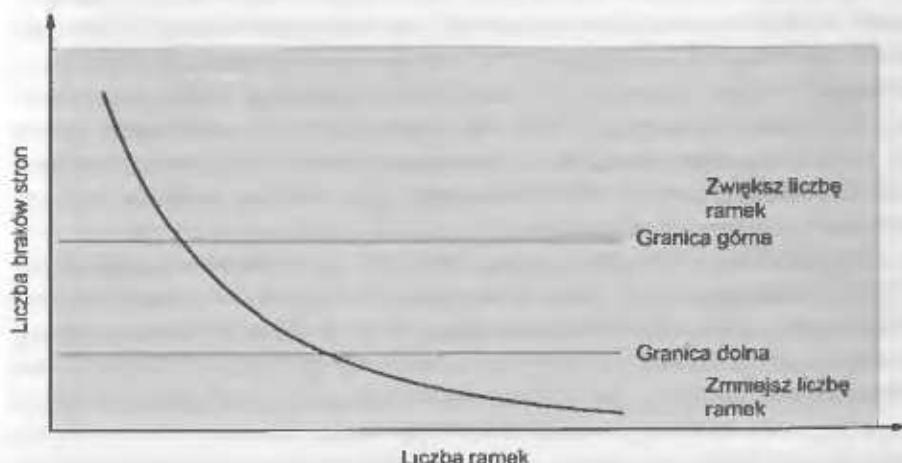
Załóżmy na przykład, że Δ wynosi 10 000 odniesień, a przerwania zegarowe występują co 5000 odniesień. Po wystąpieniu przerwania kopiuje się i zeruje wartości bitów odniesienia wszystkich stron. Jeśli więc powstanie brak strony, to można zbadać bieżący stan bitu odniesienia i dwa bity przechowywane w pamięci, aby określić, czy strona była używana w przedziale ostatnich od 10 000 do 15 000 odniesień. Jeśli była używana, to przytajmniej jeden z tych bitów będzie ustawiony. Jeśli strona nie była używana, to bity te będą wyzerowane. Do zbioru roboczego zalicza się strony, o których użyciu zaświadczał co najmniej jeden ustawiony bit. Zauważmy, że takie określenie

nie jest całkowicie dokładne, gdyż nie można powiedzieć, w którym miejscu przedziału 5000 odniesień wystąpiło dane odniesienie. Stopień tej niepewności można zmniejszyć przez powiększenie liczby bitów przechowujących historię odniesień i liczby przerwań (np. 10 bitów i przerwanie co każde 1000 odniesień). Wszelako koszt obsługi owych częstszych przerwań będzie odpowiednio wyższy.

9.7.3 Częstość braków stron

Model zbioru roboczego daje całkiem zadowalające rezultaty, a znajomość zbioru roboczego przydaje się przy wstępny stronicowaniu (zob. p. 9.8.1). Niemniej jednak nie jest to wygodna metoda nadzorowania szamotania. Prostszym postępowaniem jest mierzenie *częstości braków stron* (ang. *page-fault frequency* – PFF).

Zagadnienie polega na znalezieniu sposobu zapobiegania szamotaniu. Szamotanie odznacza się dużą częstością występowania braków stron. Dąży się zatem do nadzorowania częstości tych braków. Za dużą jej wartość świadczy o tym, że proces potrzebuje więcej ramek, natomiast zbyt mała liczba braków może oznaczać, że proces ma zbyt wiele ramek. Można ustalić górną i dolną granicę pożądanego poziomu braków stron (rys. 9.17). Procesowi, w którym częstość braków stron przekracza górną granicę, przydziela się dodatkową ramkę; jeśli zaś częstość występowania braków stron spada poniżej dolnej granicy, to usuwa się ramkę z procesu, którego ten objaw dotyczy. Tak więc można bezpośrednio mierzyć częstość braków stron i sterować nią w celu unikania szamotania.



Rys. 9.17 Częstość braków stron

Tak jak w strategii zbioru roboczego, w tej metodzie istnieje możliwość wstrzymywania procesu. Jeśli wzrasta liczba braków stron i występuje niedobór wolnych ramek, to trzeba wybrać pewien proces i wstrzymać jego wykonanie. Wolne ramki rozdziela się wtedy między procesy z najwyższymi częstotliwościami braków stron.

9.8 ■ Inne rozważania

Wybór algorytmu zastępowania i polityki przydziału to główne decyzje, które trzeba podjąć przy budowie systemu stronicowania. Istnieje jednak wiele innych spraw godnych uwagi.

9.8.1 Stronicowanie wstępne

Naturalną właściwością stronicowania na żądanie jest występowanie na początku działania procesu dużej liczby braków stron. Sytuacja taka powstaje jako wynik działań zmierzających do otrzymania w pamięci początkowej strefy procesu. Podobnie może się działać w innych sytuacjach. Na przykład, gdy wznawia się proces usunięty z pamięci operacyjnej, wówczas wszystkie jego strony znajdują się na dysku i powrót każdej z nich może nastąpić dopiero po wykryciu jej braku. Wysokiej aktywności stronicowania we wstępnej fazie procesu usiłuje się zapobiegać za pomocą *stronicowania wstępnego* (ang. *prepaging*). Strategia ta polega na jednorazowym wprowadzeniu do pamięci wszystkich stron, o których wiadomo, że będą potrzebne.

Na przykład w systemie, w którym zastosowano model zbioru roboczego, razem z każdym procesem przechowuje się listę stron należących do jego zbioru roboczego. Kiedy pojawi się konieczność wstrzymania procesu (z powodu czekania na wejście-wyjście lub niedoboru ramek), wtedy zapamiętuje się zbiór roboczy danego procesu. Gdy proces ma zostać wznowiony (operacja wejścia-wyjścia zakończyła się lub zapas ramek stał się wystarczający), wówczas przed wznowieniem procesu cały zbiór roboczy przenosi się automatycznie z powrotem do pamięci.

Stronicowanie wstępne przynosi korzyści w niektórych przypadkach. Trzeba rozstrzygnąć, czy koszt stronicowania wstępnego jest mniejszy niż koszt obsługi odpowiednich braków stron. Równie dobrze może się zdarzyć, że strony sprowadzone do pamięci wskutek stronicowania wstępnego nie zostaną użyte. Założmy, że s stron zostało wstępnie sprowadzonych do pamięci oraz że z tej liczby ułamek α stron pozostaje w użyciu ($0 \leq \alpha \leq 1$). Pytanie polega na tym, czy koszt α unikniętych braków stron jest większy czy mniejszy niż koszt wstępnego sprowadzenia do pamięci $(1 - \alpha)$ stron zbęd-

nich. Jeśli α jest bliskie zeru, to stronicowanie wstępne mija się z celem. Natomiast gdy α jest bliskie jedności, zastosowanie stronicowania wstępnego jest uzasadnione.

9.8.2 Rozmiar strony

Projektanci systemu operacyjnego dla już istniejącej maszyny rzadko mogą wybierać rozmiar strony. Jednak przy projektowaniu nowych maszyn należy podejmować decyzje dotyczące najlepszego rozmiaru strony. Jak można oczekwać, nie ma raz na zawsze określonego, jednego i najlepszego rozmiaru strony. Można natomiast mówić o zbiorze czynników przemawiających za różnymi rozmiarami. Rozmiary stron są z reguły potęgami dwójki: ungólniąc, wahają się w przedziale od $512 (2^9)$ do $16\,384 (2^{14})$ bajtów.

Jak wybrać rozmiar strony? Jednym z aspektów jest wielkość tablicy stron. Zmniejszanie rozmiaru strony powoduje, przy danym obszarze pamięci wirtualnej, zwiększenie liczby stron, a zatem i rozmiaru tablicy stron. W pamięci wirtualnej o rozmiarze 4 MB (2^{22}) będzie 4096 stron 1024-bajtowych, podczas gdy stron 8192-bajtowych byłoby tylko 512. Ponieważ każdy aktywny proces musi mieć własną kopię tablicy stron, widzimy, że jest wskazane przyjęcie dużego rozmiaru strony.

Z kolei przy małych stronach polepsza się wykorzystanie pamięci. Jeśli proces ma przydzieloną pamięć, której rozmiar, począwszy od adresu 00000, jest taki, żeby zabezpieczyć potrzeby tego procesu, to jest bardzo prawdopodobne, że pamięć procesu nie zakończy się równo na granicy którejś strony. Zatem część ostatniej strony, choć przydzielona (bo strony są jednostkami przydziału), pozostanie bezużyteczna (fragmentacja wewnętrzna). Zakładając niezależność rozmiaru procesu od rozmiaru strony, można oczekwać, że – biorąc średnio – połowa ostatniej strony każdego procesu będzie marnowana. Strata ta wyniesie tylko 256 B dla strony 512-bajtowej, lecz w przypadku strony zawierającej 8192 B sięgnie już 4096 B. Aby zminimalizować wewnętrzną fragmentację, należałoby stosować mniejsze rozmiary stron.

Inny problem stanowi czas potrzebny na odczytanie i zapisanie strony. Czas operacji wejścia-wyjścia składa się z czasu wyszukiwania, wykrywania* i samego przesyłania. Czas przesyłania jest proporcjonalny do ilości przesyłanych informacji (a więc do rozmiaru strony), co jest argumentem na rzecz stron o małych rozmiarach. Przy szybkości przesyłania wynoszącej 2 MB/s przesłanie 512 B zabiera tylko 0,2 ms. Zwłoka na wykrycie sektora wyniesie przypuszczalnie 8 ms, a czas wyszukiwania – 20 ms. Zatem od długości stro-

* Czasu wyszukiwania ścieżki na dysku (ang. seek time) i czasu wykrywania sektora na ścieżce (ang. latency time). – Przyp. tłum.

ny zależy tylko 1% ogólnego czasu przesyłania (28,2 ms). Podwojenie rozmiaru strony zwiększyłoby czas przesyłania zaledwie do 28,4 ms. Przeczytanie strony 1024-bajtowej zabierze tylko 28,4 ms, podczas gdy przeczytanie tej samej liczby bajtów w postaci dwu stron 512-bajtowych potrwałoby 56,4 ms. Tak więc dążenie do minimalizacji czasu operacji wejścia-wyjścia przemawia na rzecz wyboru większego rozmiaru strony.

Jednakże przy mniejszych rozmiarach stron powinna się zmniejszyć ogólna ilość przesyłanych informacji, gdyż można je lepiej lokalizować. Dzięki mniejszym rozmiarom strony dają się dokładniej dopasowywać do stref programu. Rozważmy na przykład proces zajmujący 200 KB, którego tylko połowa (100 KB) jest na bieżąco potrzebna i wykonywana. Gdybyśmy mieli tylko jedną wielką stronę, musielibyśmy sprowadzić ją w całości, przesyłając do pamięci i przydzielając w niej 200 KB. Gdyby strony były 1-bajtowe, wystarczyłoby sprowadzić tylko 100 KB faktycznie używanej części procesu, w wyniku czego przesłano by i przydzielono tylko 100 KB. Stosując mniejsze strony, otrzymuje się lepszą rozdzielczość, co pozwala na wydzielanie tylko tych fragmentów pamięci, które są rzeczywiście potrzebne. Przy większych stronach konieczne staje się przydzielanie i przesyłanie nie tylko fragmentów niezbędnych, lecz także wszystkiego innego, co wypadło na danej stronie – niezależnie od tego, czy jest to potrzebne czy nie. Wobec tego mniejszy rozmiar strony winien prowadzić do mniejszej ilości przekazywanych na wejściu-wyjściu informacji i zmniejszyć ogólny rozmiar przydzielanej pamięci.

Nietrudno jednak zauważyć, że w przypadku stron o rozmiarze 1 B brak strony pojawiłby się przy *kazdem* bajcie. Proces zajmujący 200 KB po otrzymaniu tylko połowy tej pamięci wygenerowałby tylko jeden brak strony w przypadku strony o rozmiarze 200 KB i 102 400 braków stron, gdyby strona była 1-bajtowa. Każdy brak strony wymaga dużego nakładu pracy na przechowanie rejestrów, zastąpienie strony, ustawienie procesu w kolejce do urządzenia stronicującego i aktualnienie tablic. Aby minimalizować liczbę braków stron, powinno się używać stron o duzych rozmiarach.

Z historycznego punktu widzenia obserwuje się tendencję do wybierania większych stron. W istocie, w pierwszym wydaniu tej książki, z 1983 r. górną granicą rozmiaru strony było 4096 B, a w 1990 r. był to najpopularniejszy rozmiar strony. Procesor Intel 80386 ma rozmiar strony równy 4 KB; w Motorola 68030 dopuszczone są strony o rozmiarach od 256 B do 32 KB. Ewolucja w kierunku większych rozmiarów stron jest prawdopodobnie wynikiem tego, że wzrost szybkości procesorów i pojemności pamięci jest szybszy niż wzrost szybkości dysków. Obecnie w ogólnym rachunku wydajności systemu braki stron są kosztowniejsze niż przedtem. Oplaca się zatem zwiększać rozmiary stron, aby zmniejszaćczęstość ich obsługi. Oczywiście zwiększa się wówczas wewnętrzna fragmentacja.

Istnieją inne czynniki, które można brać pod uwagę (w rodzaju zależności między rozmiarem strony a rozmiarem sektora na urządzeniu stronicującym). Zagadnienie to nie ma najlepszego rozwiązania. Pewne czynniki (wewnętrzna fragmentacja, strefy programu) przemawiają za małymi rozmiarami stron, podczas gdy inne – jak rozmiar tablic lub czas zajmowany przez operacje wejścia-wyjścia – uzasadniają stosowanie stron o większych rozmiarach. Co najmniej dwa systemy pozwalały na używanie dwóch różnych rozmiarów stron. Sprzęt, na którym pracuje system MULTICS (komputer GE 645), pozwala na używanie stron 64-słowowych lub 1024-słowowych. W komputerach IBM/370 można używać stron o rozmiarach 2 KB lub 4 KB. Trudność w wyborze rozmiaru strony ilustruje fakt, że system MVS dla komputera IBM/370 używa stron wielkości 4 KB, podczas gdy w systemie VS/I obrano za rozmiar strony 2 KB.

9.8.3 Odwrócona tablica stron

W punkcie 8.5.4 wprowadziliśmy pojęcie odwróconej tablicy stron. Celem tego rodzaju zarządzania stronami było zmniejszenie ilości pamięci fizycznej potrzebnej do tłumaczenia adresów wirtualnych na fizyczne. Uzyskaliśmy oszczędności wskutek utworzenia tablicy mającej po jednej pozycji na każdą stronę fizyczną, indeksowanej za pomocą pary *(identyfikator procesu, numer strony)*.

Odwrócona tablica stron pozwala zaoszczędzić miejsca w pamięci fizycznej na pamiętanie informacji o tym, której stronę pamięci wirtualnej przechowuje każda z ramek fizycznych^{*}. Jednakże odwrócona tablica stron nie zawiera pełnych informacji o logicznej przestrzeni adresowej procesu, potrzebnych wówczas, gdy strona, do której następuje właśnie odwołanie, nie przebywa w pamięci operacyjnej. W stronicowaniu na żądanie informacje te są potrzebne do obsługi braków stron. Aby je udostępnić, należy utrzymywać dla każdego procesu zewnętrzną tablicę stron. Każda taka tablica jest zbudowana tak jak konwencjonalna tablica stron procesu i zawiera dane o położeniu poszczególnych stron wirtualnych.

Czy jednak zewnętrzne tablice stron niweczą pozytek płynący z zastosowania odwróconej tablicy stron? Odwołania do tych tablic występują tylko wskutek występowania braków stron, więc dostęp do nich nie musi być szybki. One same podlegają natomiast stronicowaniu i stosownie do potrzeb są umieszczane w pamięci lub ją opuszczają. Niestety, brak strony może w tych warunkach spowodować, że zarządcą pamięci wirtualnej doprowadzi do innego braku strony na skutek wprowadzenia do pamięci operacyjnej zewnętrznej

* Ponieważ informacje te występują w niej jednorazowo; nie są powielane w tablicach związanych z poszczególnymi procesami. – Przyp. tłum.

tablicy stron, której potrzebuje, aby zlokalizować stronę wirtualną w pamięci pomocniczej. Ten specjalny przypadek wymaga bardzo starannej obsługi w jądrze systemu i opóźnienia procesu wyszukiwania stron.

9.8.4 Struktura programu

Stronicowanie na żądanie projektuje się w ten sposób, aby było przezroczyste dla programu użytkownika. W wielu przypadkach użytkownik jest zupełnie nieświadomy stronicowanej natury pamięci. Jednak kiedy indziej wiedząc, co kryje się za stronicowaniem na żądanie, można poprawić działanie systemu.

Weźmy pod uwagę nienaturalny, lecz komunikatywny przykład i założmy, że strony mają po 128 słów. Rozważmy program w Pascalu, który ma za zadanie wyzerować elementy tablicy o wymiarach 128 na 128. Zapewne można to zakodować jak poniżej:

```
var A: array [1..128] of array [1..128] of integer;
for j := 1 to 128
  do for i := 1 to 128
    do A[i][j] := 0;
```

Zauważmy, że tablica jest umieszczona w pamięci wierszami. To znaczy, że elementy tablicy są zapamiętane w porządku $A[1][1], A[1][2], \dots, A[1][128], A[2][1], A[2][2], \dots, A[2][128], \dots, A[128][128]$. Dla stron wielkości 128 słów każdy wiersz zajmuje więc jedną stronę. Zatem powyższa instrukcja zeruje jedno słowo na każdej stronie, po czym następne słowo na każdej stronie itd. Jeśli system operacyjny przydzieli całemu programowi mniej niż 128 ramek, to podczas wykonania programu brak stron wystąpi $128 \times 128 = 16\,384$ razy. Niewielka zmiana w kodowaniu:

```
var A: array [1..128] of array [1..128] of integer;
for i := 1 to 128
  do for j := 1 to 128
    do A[i][j] := 0;
```

spowoduje natomiast wyzerowanie wszystkich słów na stronie przed rozpoczęciem zerowania następnej strony, co zmniejszy liczbę braków stron do 128.

Staranny dobór struktur danych i struktur programowania może spowodować, że strefy programu zacieśniają się. Może to więc wpływać na zmniejszenie częstotliwości braków stron i liczby stron w zbiorze roboczym. Dobra lokalność cechuje stos, gdyż dostęp do niego jest dokonywany zawsze od szczytu. Z kolei tablica haszowania, w której założeniu leży rozpraszanie

odniesień, charakteryzuje się złą lokalnością. Oczywiście, lokalność odwołań jest tylko jedną z mier efektywności użytej struktury danych. Do innych czynników o dużej wadze należy szybkość przeszukiwania, ogólna liczba odwołań do pamięci i ogólna liczba zgłoszeń do stron.

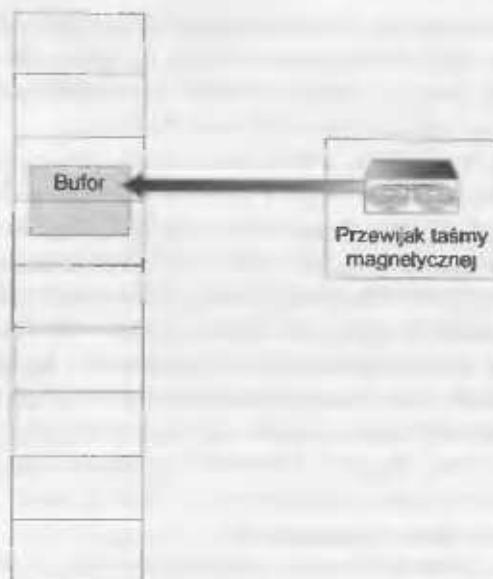
Końcowe fazy komplikacji i ładowania mogą mieć znaczący wpływ na stronicowanie. Odseparowanie kodu i danych oraz generowanie kodu wznowialnego oznacza, że strony takiego kodu mogą być chronione przed zapisem, a zatem nigdy nie zmienią swej zawartości. Czystych stron nie trzeba wysyłać z pamięci na dysk przed ich zastąpieniem. Program ładowający może unikać umieszczania procedur na granicach stron, dbając, aby każda z nich leżała w całości na jednej stronie. Procedury, które często wywołują się wzajemnie, mogą być upakowane na tej samej stronie. Upakowywanie takie jest wariantem zagadnienia plecakowego z badań operacyjnych: spróbować tak upakować segmenty o różnej długości ładowane na strony stałego rozmiaru, aby zminimalizować odwołania do różnych stron. Takie podejście jest szczególnie użyteczne dla stron o dużych rozmiarach.

Wybór języka programowania może też wpływać na stronicowanie. Na przykład często używane wskaźniki w języku Lisp wykazują tendencję do losowego dostępu do pamięci. W porównaniu z Lispelem wykorzystanie wskaźników w języku Pascal jest nikt. W systemach pamięci wirtualnej programy pascalowe będą miały lepszą lokalność odwołań i z reguły będą działały szybciej niż programy wyrażone w Lispie.

9.8.5 Powiązanie stron z operacjami wejścia-wyjścia

W stronicowaniu na żądanie przydaje się czasami *blokowanie* (ang. *locking*) niektórych stron w pamięci. Z sytuacją taką mamy do czynienia wówczas, gdy operacja wejścia-wyjścia odnosi się do pamięci (wirtualnej) użytkownika. Operacje wejścia-wyjścia są często implementowane za pomocą osobnego procesora wejścia-wyjścia. Na przykład sterownik taśmy magnetycznej otrzymuje informację o liczbie bajtów do przesłania oraz adres bufora w pamięci (rys. 9.18). Po zakończeniu operacji do procesora dociera odpowiednie przerwanie.

Należy zapewnić, by nie doszło do opisanego poniżej ciągu zdarzeń. Proces wysyła zamówienie na operację wejścia-wyjścia i zostaje ustawiony w kolejce do danego urządzenia. W międzyczasie procesor zostaje przydzielony innym procesom. W procesach tych pojawiają się braki stron i – w wyniku działania algorytmu zastępowania globalnego – jeden z procesów zastępuje stronę zawierającą bufor pamięci procesu czekającego na operację wejścia-wyjścia. Następuje wysłanie strony do pamięci pomocniczej. Po pewnym czasie, gdy zamówienie na wejście-wyjście przesunie się na czoło kolejki do



Rys. 9.18 Diagram ukazujący, dlaczego ramki używane do operacji wejścia-wyjścia powinny znajdować się w pamięci operacyjnej

danego urządzenia, zamówiona operacja prześle dane pod wskazany uprzednio adres. Jednakże z ramki, która zawierała bufor, korzysta teraz inna strona należącą do innego procesu.

Najczęściej spotyka się dwa rozwiązania tego problemu. Jeden polega na zakazie wykonywania operacji wejścia-wyjścia wprost do pamięci użytkownika. Zamiast tego dane podlegają ustawicznemu kopiowaniu między pamięcią systemu a pamięcią użytkownika. Operacje wejścia-wyjścia są wykonywane tylko między pamięcią systemu a urządzeniami wejścia-wyjścia. Aby zapisać blok danych na taśmie, najpierw kopiuje się go do pamięci systemu i dopiero stamtąd zapisuje na taśmie.

Nakłady ponoszone na dodatkowe kopowanie mogą być trudne do zaakceptowania. Inne rozwiązanie polega na umożliwieniu *blokowania stron* w pamięci. Każdej ramce przyporządkowuje się bit blokowania. Jeśli ramka jest zablokowana, to nie bierze udziału w zastępowaniu stron. W tej metodzie przed zapisaniem bloku na taśmie blokuje się w pamięci zawierające go strony. Dalsze działanie systemu przebiega bez zmian. Stron zablokowanych nie można zastępować. Zablokowane strony zwalnia się dopiero po zakończeniu operacji.

Inne zastosowanie bitu blokowania ma miejsce przy zwyczajnym zastępowaniu stron. Rozważmy następujący ciąg zdarzeń. W niskopriorytetowym procesie zatrzymano strony. Wybrany ramkę do zastąpienia, system stronicując

jący wczytuje niezbędną stronę do pamięci. Gotowy do dalszego działania niskopriorytetowy proces ustawia się w kolejce procesów gotowych i czeka na procesor. Ponieważ priorytet procesu jest niski, planista przydziału procesora może omijać ten proces przez jakiś czas. Podczas gdy niskopriorytetowy proces czeka na przydział procesora, inny proces – o wysokim priorytecie – wykazuje brak strony. Szukając strony do zastąpienia, system stronicowania typuje stronę pozostającą w pamięci, do której nie było odniesień i która nie została zmieniona. Może się nią okazać strona niskopriorytetowego procesu, dopiero co sprowadzona. Strona ta sprawia wrażenie idealnej kandydatki do zastąpienia: jest czysta, nie trzeba jej odsyłać na zewnątrz i zdaje się być nie używana od dłuższego czasu.

Decyzja o tym, czy procesowi wysokopriorytetowemu wolno zastąpić stronę procesu o niskim priorytecie, należy do sfery polityki. W końcu opóżnia się proces niskopriorytetowy na rzecz procesu wysokopriorytetowego. Z drugiej strony dochodzi tu do marnowania wysiłku poniesionego na sprowadzenie do pamięci strony procesu niskopriorytetowego. Jeśli zdecydujemy się chronić nowo sprowadzoną stronę przed zastąpieniem do czasu przynajmniej jednokrotnego jej użycia, to do implementacji tego mechanizmu można posłużyć się bitem blokowania. Po wybraniu strony do zastąpienia jej bit blokowania otrzyma wartość 1 i zachowa ją tak długo, aż oczekującemu procesowi zostanie znów przydzielony procesor.

Używanie bitu blokowania może być niebezpieczne wówczas, gdy po jego ustawieniu nigdy nie nastąpi jego wyzerowanie. W takiej sytuacji (spowodowanej np. błędem w systemie operacyjnym) zablokowana ramka staje się bezużyteczna. W systemie operacyjnym komputera Macintosh stosuje się blokowanie stron, gdyż jest to system dla jednego użytkownika i nadmiar zablokowanych stron zaszkodziłby tylko wynajmującemu je użytkownikowi. Systemy dla wielu użytkowników muszą okazywać mniej zaufania użytkownikom. Na przykład w systemie SunOS dopuszcza się „zalecenia” blokowania, lecz system może je odrzucić, jeśli pula wolnych stron staje się zbyt mała lub jeśli dany proces próbuje zablokować zbyt wiele stron w pamięci.

9.8.6 Przetwarzanie w czasie rzeczywistym

W tym rozdziale skoncentrowaliśmy się na metodach jak najlepszego, ogólnego wykorzystania systemu komputerowego przez optymalizowanie posługiwania się pamięcią. Przechowując w pamięci potrzebne na bieżąco dane i przenosząc nie używane dane na dysk, powiększamy ogólną przepustowość systemu. Jednak poszczególne procesy mogą być wskutek tego dyskryminowane z powodu dodatkowych braków stron występujących podczas wykonywania tych procesów.

Rozważmy proces lub wątek działający w czasie rzeczywistym, tak jak to opisaliśmy w rozdz. 4. Proces taki powinien niezwłocznie otrzymywać procesor i przebiegać aż do zakończenia z minimalnymi opóźnieniami. Pamięć wirtualna nie sprzyja obliczeniom w czasie rzeczywistym, ponieważ może wprowadzać nieoczekiwane, długotrwałe przestoje w wykonywaniu procesu wskutek sprowadzania brakujących stron. Dlatego systemy czasu rzeczywistego prawie nigdy nie mają pamięci wirtualnej.

W założeniach systemu Solaris 2 firmy Sun leży zarówno podział czasu, jak i obliczenia w czasie rzeczywistym. W celu rozwiązania problemu braków stron system Solaris 2 umożliwia procesowi powiadomienie go o ważnych stronach. Oprócz wspomnianych „wskazówek” dotyczących korzystania ze stron, system operacyjny umożliwia uprzywilejowanym użytkownikom blokowanie stron w pamięci. Nadużywanie tego mechanizmu może prowadzić do uniemożliwiania wszystkim innym procesom kontaktu z systemem. Jest niezbędne, aby procesy czasu rzeczywistego miały ograniczone i małe czasy oczekiwania.

9.9 ■ Segmentacja na żądanie

Chociaż stronicowanie na żądanie uważa się na ogół za najwydajniejszy sposób organizacji pamięci wirtualnej, do jego zastosowania są potrzebne spore ilości sprzętu. Wobec niedostatku sprzętu do realizacji pamięci wirtualnej stosuje się czasami mniej wydajne środki. Należy do nich *segmentacja na żądanie* (ang. *demand segmentation*). Procesor Intel 80286 nie ma możliwości stronicowania, ale operuje segmentami. Pracujący na tym procesorze system operacyjny OS/2 wykorzystuje zawarte w sprzęcie możliwości segmentacji do implementowania segmentacji na żądanie jako jedynego możliwego przybliżenia stronicowania na żądanie.

System OS/2, zamiast stronami, przydziela pamięć segmentami. Segmente są opisane za pomocą *deskryptorów segmentów* (ang. *segment descriptors*), które zawierają informacje o długości segmentów, trybie ich ochrony i umiejscowieniu. Wykonywany proces nie musi mieć w pamięci wszystkich swoich segmentów. W zamian za to deskryptor segmentu zawiera *bit poprawności* wskazujący dla każdego segmentu, czy znajduje się on w danej chwili w pamięci. Gdy proces odniesie się do segmentu zawierającego kod lub dane, wówczas sprzęt sprawdzi bit poprawności. Jeśli segment znajduje się w pamięci głównej, to dostęp odbywa się bez przestojów. Gdy segmentu nie ma w pamięci, wtedy następuje przejście do systemu operacyjnego (brak segmentu) – zupełnie tak, jak przy implementacji stronicowania na żądanie. System OS/2 wysyła wtedy jakiś segment do pamięci pomocniczej

i sprowadza cały żądany segment. Przerwany rozkaz (któremu zabrakło segmentu) jest potem kontynuowany.

Do określania segmentu, który ma być zastąpiony w razie braku segmentu, system OS/2 używa innego bitu w deskryptorze segmentu, zwanego *bitem udostępnienia* (ang. *accessed bit*). Bit udostępnienia pełni tę samą funkcję co bit odniesienia w środowisku stronicowania na żądanie – jest ustawiany wówczas, gdy jakkolwiek bajt segmentu zostanie przeczytany lub zapisany. Utrzymuje się kolejkę złożoną z deskryptorów wszystkich segmentów w pamięci. Po każdym kwancie czasu system operacyjny wysuwa na czoło kolejki segmenty z ustawionymi bitami udostępnienia. Bity udostępnienia są wtedy zerowane. W ten sposób kolejka tworzy uporządkowanie, w którym segmenty ostatnio używane znajdują się na początku. Ponadto w systemie OS/2 są funkcje umożliwiające procesom informowanie systemu o segmentach usuwalnych lub takich, które muszą stale pozostawać w pamięci. Informacja ta służy do przedstawiania pozycji w kolejce. W przypadku wystąpienia pułapki kwitującej niewłaściwe odniesienie do segmentu, procedury zarządzania pamięcią określają najpierw, czy dostępna wolna pamięć pomieszczy segment. Można wykonać upakowanie pamięci w celu pozbicia się zewnętrznej fragmentacji. Jeśli nawet po upakowaniu nie ma wystarczającej ilości wolnej pamięci, to trzeba któryś segment zastąpić. W tym celu wybiera się segment z końca kolejki i przesyła go do obszaru wymiany. Jeśli nowo otrzymany, wolny obszar pamięci wystarcza na pomieszczenie żadanego segmentu, to potrzebny segment zostaje przeczytany na miejsce usuniętego segmentu. Następnie aktualnia się deskryptor segmentu i umieszcza go na czele kolejki. W przeciwnym razie, jeśli jeszcze brakuje pamięci, dokonuje się jej upakowania i powtarza się całe postępowanie.

Segmentacja na żądanie pociąga za sobą – co jest widoczne – pewne koszty, toteż nie jest ona najlepszym sposobem wykorzystania zasobów systemu komputerowego. Jednak w przypadku uboższego sprzętu pozostaje do wyboru tylko całkowity brak pamięci wirtualnej. Trudności związane z systemami pozbawionymi pamięci wirtualnej, jak te opisane w rozdz. 8. dowodzą, że takie rozwiązanie ma również braki. Dlatego też segmentacja na żądanie jest rozsądnym kompromisem na rzecz funkcjonalności w warunkach ograniczeń sprzętowych uniemożliwiających stronicowanie na żądanie.

9.10 ■ Podsumowanie

Jest wskazane, aby istniała możliwość wykonywania procesów, których logiczna przestrzeń adresowa jest większa od dostępnej przestrzeni adresów fizycznych. Można uczynić taki proces wykonywalnym przez jego restruktura-

ryzację za pomocą nakładek, jednak jest to na ogół trudne zadanie do zaprogramowania. Pamięć wirtualna jest techniką pozwalającą na odwzorowywanie wielkiej logicznej przestrzeni adresowej w mniejszej pamięci fizycznej. Pamięć wirtualna umożliwia wykonywanie niezwykle dużych procesów, a także pozwala na podniesienie stopnia wieloprogramowości, co polepsza wykorzystanie procesora. Co więcej, zwalnia ona programistę aplikacji od kłopotów związanych z dostępnością pamięci.

W czystym stronicowaniu na żądanie strony nie sprawdza się dopóty, dopóki nie ma do niej odniesienia. Pierwsze odniesienie powoduje wystąpienie braku strony, o którym jest powiadamiana rezydująca na stałe w pamięci część systemu operacyjnego. System za pomocą informacji zapisanych w wewnętrznej tablicy określa miejsce pobytu strony w pamięci pomocniczej. Następnie znajduje wolną ramkę i czyta do niej stronę z pamięci pomocniczej. Tablica stron zostaje uaktualniona dla odzwierciedlenia tej zmiany, a rozkaz przerwany z powodu braku strony zostaje wznowiony. Podejście takie pozwala na pracę nawet takich procesów, których całkowity obraz pamięci nie znajduje się od razu w pamięci głównej. Póki częstotliwość braków stron jest względnie niska, poty działanie systemu jest do zaakceptowania.

Stronicowania na żądanie można użyć w celu zmniejszenia liczby ramek przydzielonych procesowi. Może to wpływać na podniesienie stopnia wieloprogramowości (bo umożliwia dopuszczenie do działania większej liczby procesów w tym samym czasie) i – przynajmniej w teorii – na polepszenie wykorzystania procesora przez system. Pozwala ono również na wykonywanie procesów nawet wtedy, gdy ich wymagania pamięciowe przekraczają całkowitą ilość pamięci dostępnej fizycznie. Procesy takie działają w pamięci wirtualnej.

Jeśli łączne zapotrzebowanie na pamięć jest większe niż obszar pamięci fizycznej, to staje się niezbędné zastępowanie stron w pamięci, tj. zwalnianie ramek na nowe strony. Stosuje się rozmaite algorytmy zastępowania stron. Algorytm FIFO zastępowania stron jest łatwy do zaprogramowania, lecz jest obciążony anomalią Belady'ego. Optymalne zastępowanie stron wymaga wiedzy o przyszłości. Algorytm LRU zastępowania stron jest przybliżeniem algorytmu optymalnego, ale i on może być trudny do zrealizowania. Większość algorytmów zastępowania stron, jak na przykład algorytm drugiej szansy, stanowi przybliżenie algorytmu LRU.

Algorytm zastępowania stron wymaga uzupełnienia o jakąś politykę przydziału ramek. Przydział może być ustalony, polegający raczej na lokalnym zastępowaniu stron, lub dynamiczny – umożliwiający zastępowanie globalne. W modelu zbioru roboczego zakłada się, że wykonywanie procesów charakteryzuje się strefowością. Zbior roboczy tworzą strony należące do bieżącej strefy. Każdy proces powinien mieć przydzieloną liczbę ramek wystarczającą dla bieżącego zbioru roboczego.

Proces, który nie ma wystarczającej liczby ramek na swój zbiór roboczy, zaczyna się szamotać. Dostarczenie każdemu procesowi liczby ramek wystarczającej do uniknięcia szamotania może wymagać wymiany i planowania procesów.

Oprócz rozwiązymania głównych problemów zastępowania stron i przydziału ramek, właściwie zaprojektowany systemu stronicowania wymaga jeszcze rozważenia rozmiaru strony, blokowania stron w pamięci na czas operacji wejścia-wyjścia, stronicowania wstępnego, odpowiedniej struktury programu i innych zagadnień. Pamięć wirtualną można rozumieć jako jeden poziom w hierarchii poziomów pamięci w systemie komputerowym. Każdy poziom ma własny czas dostępu, rozmiar i parametry kosztów. Pełny przykład systemu hybrydowej, funkcjonalnej pamięci wirtualnej zaprezentowano w rozdziale dotyczącym systemu Mach, dostępnym na stronach WWW w sieci Internet.

■ Ćwiczenia

- 9.1 Kiedy występują braki stron? Opisz działania podejmowane przez system operacyjny, gdy wystąpi brak strony.
- 9.2 Założmy, że dysponujemy ciągiem odniesień dla procesu z m ramkami (początkowo wszystkie są puste). Ciąg odniesień ma długość p i występuje w nim n różnych numerów stron. Dla dowolnego algorytmu zastępowania stron określ:
 - (a) dolną granicę braków stron;
 - (b) górną granicę braków stron.
- 9.3 Pewien komputer dostarcza swoim użytkownikom pamięć wirtualną wielkości 2^{32} B. Komputer ten ma 2^{16} B pamięci fizycznej. Pamięć wirtualna jest zrealizowana za pomocą stronicowania, a rozmiar strony wynosi 4096 B. Proces użytkownika wytworzył adres wirtualny 11123456. Wyjaśnij, w jaki sposób system ustala odpowiadający temu adresowi wirtualnemu adres fizyczny. Dokonaj rozróżnienia między operacjami programowymi a sprzętowymi.
- 9.4 Które z następujących technik i struktur są „dobre” dla środowiska stronicowania na żądanie, które zaś są „niedobre”:
 - (a) stos;
 - (b) haszowana tablica symboli (ang. *hashed symbol table*);

- (c) przeszukiwanie sekwencyjne;
- (d) przeszukiwanie binarne;
- (e) czysty kod;
- (f) operacje wektorowe;
- (g) adresowanie pośrednie.

9.5 Założmy, że mamy pamięć stronicowaną na żądanie. Tablica stron jest przechowywana w rejestrach. Obsługa braku strony zabiera 8 ms wtedy, kiedy jest dostępna pusta strona lub strona zastępowana jest nie zmieniona, oraz 20 ms, jeśli zastępowana strona jest zmieniona. Czas dostępu do pamięci wynosi 100 ns.

Przyjmijmy, że strona do zastąpienia jest zmieniona w 70 przypadkach na 100. Ile wynosi maksymalna akceptowalna częstość braków stron, jeśli efektywny czas dostępu ma nie być dłuższy niż 200 ns?

- 9.6** Rozważ następujące algorytmy zastępowania stron. Ustaw je w kolejności od „złego” do „znakomitego” według ich częstości braków stron. Oddziel algorytmy podane na anomalię Belady’ego od tych, które jej nie ulegają:
- (a) algorytm LRU;
 - (b) algorytm FIFO;
 - (c) zastępowanie optymalne;
 - (d) zastępowanie metodą drugiej szansy.

- 9.7** Zaimplementowanie w systemie komputerowym techniki pamięci wirtualnej pociąga za sobą pewne koszty i pewne korzyści. Wymień owe koszty i korzyści. Czy jest możliwe, aby koszty przewyższały korzyści? Jeśli tak, to jakie zastosować miary, żeby do tego nie dopuścić?

- 9.8** W pewnym systemie operacyjnym zaimplementowano stronicowaną pamięć wirtualną przy użyciu procesora z czasem cyklu wynoszącym 1 μ s. Dostęp do strony innej niż bieżąca zabiera dodatkowo 1 μ s. Strony mają po 1000 słów, a urządzeniem stronicującym jest bęben wirujący z prędkością 3000 obrotów na minutę i transmitujący 1 milion słów na sekundę. W systemie tym wykonano następujące pomiary statystyczne:

- Ze wszystkich wykonanych rozkazów z inną niż bieżącą stroną kontaktowało się 0,1% rozkazów.

- Sposród rozkazów dotyczących innych stron 80% odnajdywało stronę wprost w pamięci operacyjnej.
- Gdy była potrzebna nowa strona, wówczas w 50% przypadków okażywało się, że strona zastępowana była zmieniona.

Oblicz efektywny czas rozkazu w tym systemie, przy założeniu że system wykonuje tylko jeden proces, a procesor pozostaje bezczynny podczas operacji bębnowych.

- 9.9** Rozważ system stronicowania na żądanie, w którym zmierzone w czasie parametry użytkowania przedstawiają się następująco:

wykorzystanie procesora	20%
dysk stronicujący	97,7%
inne urządzenia wejścia-wyjścia	5%

Które (jeśli w ogóle) z następujących propozycji poprawiłyby (prawdopodobnie) wykorzystanie procesora? Odpowiedź uzasadnij.

- (a) zainstalowanie szybszego procesora;
- (b) zainstalowanie większego dysku do stronicowania;
- (c) zwiększenie stopnia wieloprogramowości;
- (d) zmniejszenie stopnia wieloprogramowości;
- (e) zainstalowanie większej ilości pamięci operacyjnej;
- (f) zainstalowanie szybszego dysku twardego lub kilku sterowników kilku dysków twardych;
- (g) dodanie stronicowania wstępnego do algorytmów sprowadzania stron;
- (h) zwiększenie rozmiaru strony.

- 9.10** Rozważmy dwuwymiarową tablicę A :

var A : array [1..100] of array [1..100] of integer;

gdzie $A[1][1]$ znajduje się w komórce 200 systemu stronicowanej pamięci, której strony mają rozmiar 200 komórek. Na stronie 0 znajduje się mały proces (komórki od 0 do 199) manipulujący tą tablicą, tak więc każdy rozkaz będzie pobierany ze strony 0.

Ile braków stron powstanie podczas wykonywania następujących pętli nadawania wartości początkowych tablicy, jeśli założymy, że działamy na 3 ramkach i stosujemy zastępcowanie LRU, przy czym ramka 1 zawiera proces, a dwie pozostałe są początkowo puste:

- (a) **for** $j := 1$ **to** 100 **do**
 for $i := 1$ **to** 100 **do**
 $A[i][j] := 0;$
- (b) **for** $i := 1$ **to** 100 **do**
 for $j := 1$ **to** 100 **do**
 $A[i][j] := 0;$

9.11 Rozważmy następujący ciąg odniesień do stron:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

ile braków stron wystąpi dla następujących algorytmów zastępowania przy założeniu, że mamy jedną, dwie, trzy, cztery, pięć, sześć lub siedem ramek? Pamiętajmy, że wszystkie ramki są początkowo puste, tak że pierwsze kontakty z nowymi stronami będą zawsze powodowały po jednym braku strony.

- algorytm LRU;
- zastępowanie FIFO;
- zastępowanie optymalne.

- 9.12 Założymy, że chcemy użyć algorytmu stronicowania korzystającego z bitu odniesienia (np. zastępowania na zasadzie drugiej szansy lub modelu zbioru roboczego), ale sprzęt nie ma takiego bitu. Naszkicuj, jak można zasymulować bit odniesienia wtedy, gdy brak go w sprzęcie, lub wyjaśnij, dlaczego wykonanie tego nie jest możliwe. Jeśli możliwość taka istnieje, to oblicz jej koszt.
- 9.13 Obmyślono nowy algorytm zastępowania stron, o którym sądzi się, że jest optymalny. W jednym z zaaplikowanych mu testów pojawiła się anomalia Belady'ego. Czy nowy algorytm jest optymalny? Odpowiedź uzasadnij.
- 9.14 Założymy, że jako politykę zastępowania (w systemie stronicowanym) obrano regularne sprawdzanie każdej strony i usuwanie tej strony, która nie została użyta od czasu ostatniego sprawdzenia. Jakie korzyści i jakie straty przyniosą te zasady w porównaniu z zastosowaniem algorytmu LRU lub zastępowania według drugiej szansy?
- 9.15 Segmentacja jest podobna do stronicowania, lecz używa się w niej „stron” o zmiennej długości. Zdefiniuj dwa algorytmy zastępowania segmentów oparte na schematach FIFO i LRU. Pamiętaj, że ze względu na różne rozmiary segmentów segment, który został wytypowany do zastąpienia, może nie pozostawić dostatecznie dużego, spójnego obszaru

dla potrzebnego segmentu. Rozważ strategie dla systemów, w których segmentów nie wolno przemieszczać, i dla systemów z segmentami przemieszczalnymi.

- 9.16** Algorytm zastępowania stron powinien minimalizować liczbę braków stron. Minimalizację tę można otrzymać, rozpraszając intensywnie używane strony równomiernie po całej pamięci, zamiast depuszczać, aby rywalizowały ze sobą o niewielką liczbę ramek pamięci. Z każdą ramką można powiązać licznik stron przypisywanych danej ramce. Wówczas w celu zastąpienia strony, szukać się będzie ramki z najmniejszym licznikiem.
- (a) Korzystając z tego pomysłu, określ algorytm zastępowania stron. Zwróć szczególną uwagę na kwestie: (1) początkowych wartości liczników, (2) decydowania o zwiększeniu liczników, (3) decydowania o zmniejszaniu liczników oraz (4) sposobu wybierania strony do zastąpienia.
 - (b) Dla czterech ramek i poniższego ciągu odniesień określ, ile braków stron wystąpi w Twoim algorytmie?
- 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2
- (c) Mając dane: ciąg odniesień jak w części (b), cztery ramki i strategię zastępowania optymalnego, określ, ile wyniesie minimalna liczba błędów stron.
- 9.17** Rozważmy system stronicowania na żądanie z dyskiem stronicującym, którego średni czas dostępu i przesyłania wynosi 20 ms. Tłumaczenie adresów za pomocą tablicy stron odbywa się w pamięci głównej o czasie dostępu 1 μ s. Każde odwołanie do pamięci za pośrednictwem tablicy wymaga zatem dwu dostępów. Aby skrócić ten czas, dodano pamięć asocjacyjną, która redukuje czas dostępu do jednego odniesienia do pamięci, jeśli wpis z tablicy stron jest w pamięci asocjacyjnej.
- Załóżmy, że 80% dostępów kończy się na pamięci asocjacyjnej, oraz że 10% pozostałych dostępów (lub 2% całości) powoduje brak strony. Jaki będzie efektywny czas dostępu do pamięci?
- 9.18** Rozważamy system komputerowy ze stronicowaniem na żądanie, w którym bieżący stopień wieloprogramowości wynosi 4. W systemie niedawno wykonano pomiary w celu określenia wykorzystania procesora i dysku stronicującego. W ich wyniku otrzymano następujące zależności. Co w każdym przypadku się wydarzyło? Czy aby zwiększyć wykorzystanie procesora, można zwiększyć stopień wieloprogramowości? Czy stronicowanie można ocenić jako pomocne?

- (a) Wykorzystanie procesora – 13%; wykorzystanie dysku – 97%.
 - (b) Wykorzystanie procesora – 87%; wykorzystanie dysku – 3%.
 - (c) Wykorzystanie procesora – 13%; wykorzystanie dysku – 3%.
- 9.19 Dysponujemy systemem operacyjnym dla maszyny, która miała rejestr bazowy i rejestr graniczny, ale dostosowaliśmy ją do celów obsługi tablicy stron. Czy tablice stron mogą posłużyć do symulacji rejestru bazowego i rejestru granicznego? W jaki sposób, a jeśli nie, to dlaczego?
- 9.20 Co jest przyczyną szamotania? Jak system wykrywa szamotanie? Co system może zrobić po wykryciu szamotania, aby się go pozbyć?

Uwagi bibliograficzne

Stronicowanie na żądanie zostało po raz pierwszy użyte w systemie Atlas, zrealizowanym w komputerze MUSE w Manchester University około 1960 r. (artykuł Kilburna i in. [213]). Innym wczesnym systemem stronicowania na żądanie był MULTICS, zaimplementowany w komputerze GE 645, opisany w książce Organicka [317].

Belady i inni współautorzy artykułu [28] jako pierwsi zaobserwowali, że strategię zastępowania FIFO może cechować anomalia, która wzięła nazwę od jego nazwiska. Mattson i in. [273] wykazali, że algorytmy stosowane nie ulegają anomalii Belady'ego.

Algorytm zastępowania optymalnego pochodzi od Belady'ego [28]. Jego optymalności dowiedli Mattson i in. w pracy [273]. Optymalny algorytm Belady'ego odnosi się do stałego przydziału pamięci. Prieve i Fabry [335] opracowali algorytm optymalny w sytuacjach, gdy przydział jest zmienny.

Ulepszony algorytm zegarowy omówili Carr i Hennessy [63]. Zastosowano go w schemacie zarządzania pamięcią wirtualną w komputerze Macintosh, co opisał Goldman [152].

Szamotanie zostało omówione przez Denninga [98]. Również model zbioru roboczego opracował Denning [98]; on także omówił model zbioru roboczego w artykule [100].

Schemat nadzorowania częstości braków stron pochodzi od Wulfa [445], który skutecznie zastosował tę technikę w systemie komputerowym Burroughs B5500. Chu i Opderbeck [1977] omówili w artykule [74] zachowanie programu i algorytm zastępowania stron według częstości braków stron. Gupta i Franklin [158] porównali wydajność schematu zbioru roboczego i zastępowania według częstości braków stron.

Segmentację na żądanie i szczegóły systemu OS/2 opisał Iacobucci [185]. Inne informacje dotyczące systemu OS/2 można znaleźć w podręczniku [289] wydanym przez firmę Microsoft.

Stronicowanie w mikroprocesorze Intel 80386 opisano w dokumentacji [191], a sprzęt stronicujący Motorola 68030 omówiono w książce [301]. Zarządzanie pamięcią wirtualną w systemie operacyjnym VAX/VMS przedstawili Levy i Lipman w artykule [253]. Omówienia systemów operacyjnych stacji roboczych i pamięci wirtualnej dokonał Hagmann [160].



Rozdział 10

INTERFEJS SYSTEMU PLIKÓW

Dla większości użytkowników system plików jest najbardziej widocznym aspektem systemu operacyjnego. Tworzy on mechanizm bezpośredniego przechowywania informacji i bezpośredniego dostępu zarówno do danych, jak i do programów systemu operacyjnego oraz wszystkich użytkowników systemu komputerowego. System plików składa się z dwu wyraźnie wyodrębnionych części: zbioru *plików*, z których każdy zawiera powiązane ze sobą informacje, i *struktury katalogów*, za pomocą której organizuje się i udostępnia informacje o wszystkich plikach w systemie. W niektórych systemach plików występuje trzecia część – *strefy* (ang. *partitions*), które służą do wyodrębniania fizycznie lub logicznie wielkich zbiorów katalogów. W tym rozdziale zajmujemy się różnorodnymi aspektami plików oraz różnymi strukturami katalogowymi. Przedstawiamy też sposoby ochrony plików niezbędne w środowisku, w którym dostęp do plików ma wielu użytkowników i w którym zazwyczaj jest wskazane nadzorowanie tego, kto i w jaki sposób może korzystać z plików. Na zakończenie omawiamy semantykę dzielenia plików między wiele procesów.

10.1 ■ Pojęcie pliku

Komputery mogą przechowywać informacje na wielu różnych nośnikach, takich jak magnetyczne dyski i taśmy lub dyski optyczne. W celu wygodnego korzystania z systemu komputerowego system operacyjny dostarcza jednolitego pod względem logicznym obrazu przechowywanych informacji. System operacyjny, w oderwaniu od cech fizycznych urządzeń magazynowania infor-

macji, definiuje logiczną jednostkę magazynowania informacji – *plik* (ang. *file*). Za pomocą systemu operacyjnego pliki są odwzorowywane na urządzeniach fizycznych. Takie urządzenia pamięci charakteryzują się zazwyczaj *nieudotnością* (ang. *nonvolatile devices*), co oznacza, że ich zawartość jest w stanie przetrwać awarie zasilania i powtarzanie rozruchu systemu.

Plik jest nazwanym zbiorem powiązanych ze sobą informacji, zapisanym w pamięci pomocniczej. Z punktu widzenia użytkownika plik jest najmniejszym przydziałem logicznej pamięci pomocniczej, tzn. dane nie mogą być zapisywane w pamięci pomocniczej inaczej niż w obrębie pliku. Najczęściej pliki reprezentują programy (zarówno źródłowe, jak i wynikowe) oraz dane. Pliki danych mogą być liczbowe, literowe, alfanumeryczne lub binarne. Pliki mogą mieć format swobodny, jak na przykład pliki tekstowe, lub ścisłe określony. Mówiąc ogólnie, plik jest ciągiem bitów, bajtów, wierszy lub rekordów, których znaczenie określa twórca pliku i jego użytkownik. Pojęcie pliku jest zatem bardzo ogólne.

Informacje zawarte w pliku są określone przez jego twórcę. W pliku można przechowywać informacje różnego rodzaju: programy źródłowe, programy wynikowe, programy wykonywalne, dane liczbowe, teksty, listy płac, obrazy grafiki komputerowej, nagrania dźwiękowe itd. Plik ma określoną strukturę, stosownie do swojego typu. *Plik tekstowy* (ang. *text file*) jest ciągiem znaków składających się na wiersze (i być może strony). *Plik źródłowy* (ang. *source file*) jest ciągiem podprogramów i funkcji, które mają własną organizację – na przykład deklaracje poprzedzające wykonywalne instrukcje. *Plik wynikowy* (ang. *object file*) jest ciągiem bajtów połączonych w bloki rekordów interpretowanych przez procedury ładujące. *Plik wykonywalny* (ang. *executable file*) składa się z ciągu sekcji kodu, które można załadować do pamięci operacyjnej i wykonać. Wewnętrzna budowa plików jest omówiona w p. 10.4.1.

10.1.1 Atrybuty pliku

Dla wygody osób używających komputery plik zaopatruje się w nazwę, za pomocą której można się do niego odwoływać. Nazwa jest na ogół ciągiem znaków, takim jak *example.c*. Niektóre systemy rozróżniają wielkie i małe litery w nazwach, inne zaś je utożsamiają. Z chwilą gdy plik otrzyma nazwę, staje się niezależny od procesu, użytkownika, a nawet od systemu, który go utworzył. Na przykład pewien użytkownik może utworzyć plik o nazwie *example.c*, drugi użytkownik może ten plik edytować, podając jego nazwę. Właściciel pliku może zapisać plik na dyskietce lub na taśmie magnetycznej i może odczytać go w innym systemie, przy czym plik wciąż będzie się nazywał *example.c*.

Pliki mają także pewne inne atrybuty, różne w poszczególnych systemach operacyjnych, lecz na ogół są wśród nich następujące:

- **Nazwa:** Symboliczna nazwa pliku jest jedyną informacją przechowywaną w postaci czytelnej dla człowieka.
- **Typ:** Ta informacja jest potrzebna w tych systemach, w których rozróżnia się typy plików.
- **Położenie:** Jest to wskaźnik do urządzenia i położenia pliku na tym urządzeniu.
- **Rozmiar:** Atrybut ten zawiera bieżący rozmiar pliku (w bajtach, słowach lub blokach), może też zawierać maksymalny dopuszczalny rozmiar pliku.
- **Ochrona:** Informacje kontroli dostępu służą do sprawdzania, kto może plik czytać, zapisywać, wykonywać itd.
- **Czas, data i identyfikator użytkownika:** Informacje te mogą obejmować: (1) czas utworzenia pliku, (2) ostatnią jego zmianę i (3) ostatnie użycie pliku. Takie dane mogą być użyteczne ze względów ochronnych, bezpieczeństwa i w celu doglądania użycia plików.

Informacje o wszystkich plikach są przechowywane w strukturze katalogowej, która również rezyduje w pamięci pomocniczej. Zapisanie takich informacji wodniesieniu do jednego pliku może zajść od 16 do 1000 bajtów. W systemie z dużą liczbą plików rozmiar samego katalogu może wynosić megabajty. Ponieważ katalogi, tak jak pliki, muszą być nieulotne, należy je przechowywać na urządzeniach i sprowadzać do pamięci operacyjnej kawałkami, stosownie do potrzeb. Organizację struktury katalogowej omówimy w p. 10.3.

10.1.2 Operacje plikowe

Plik jest *abstrakcyjnym typem danych*. Do właściwego zdefiniowania pliku jest niezbędne rozważenie operacji, które można na plikach wykonywać. System operacyjny udostępnia funkcje systemowe do tworzenia, zapisywania, czytania, zmiany położenia w pliku, usuwania i skracania plików. Przesłedźmy, co system operacyjny musi robić w przypadku każdej z tych szesziu podstawowych operacji plikowych. Latwo będzie wtedy dostrzec, jak powinny być realizowane operacje podobne, na przykład przemianowanie pliku.

- **Tworzenie pliku:** Do utworzenia (ang. *creating*) pliku są niezbędne dwa kroki. Po pierwsze, w systemie plików musi zostać znalezione miejsce na plik. Sposoby przydzielenia plikom obszarów omówimy w rozdz. 11. Po

drugie, w katalogu należy utworzyć wpis pliku. Wpis katalogowy zawiera nazwę pliku i informację o jego położeniu w systemie plików.

- **Zapisywanie pliku:** Aby pisać (ang. *write*) do pliku, wywołuje się funkcję systemową, podając zarówno nazwę pliku, jak i informację, która ma być w pliku zapisana. Znając nazwę pliku, system przeszukuje katalog, aby znaleźć położenie pliku. System musi przechowywać wskaźnik pisania określający miejsce w pliku, do którego będzie się odnosić kolejna operacja pisania. Wskaźnik pisania musi być uaktualniany podczas każdego pisania.
- **Czytanie pliku:** Do czytania (ang. *reading*) pliku służy funkcja systemowa, w której wywołaniu określa się nazwę pliku oraz miejsce (w pamięci operacyjnej), gdzie ma być umieszczony następny blok pliku. Znowu następuje przeszukanie katalogu w celu odnalezienia stosownego wpisu katalogowego, a system musi utrzymywać *wskaźnik czytania* określający miejsce w pliku, od którego nastąpi kolejne czytanie. Po wykonaniu czytania uaktualnia się wartość wskaźnika czytania. Ponieważ, ogólnie biorąc, plik jest albo czytany, albo zapisywany, większość systemów utrzymuje tylko *wskaźnik bieżącego położenia w pliku* (ang. *current-file-position*). Zarówno operacje czytania, jak i pisania korzystają z tego samego wskaźnika; oszczędza się w ten sposób miejsce i upraszcza system.
- **Zmiana pozycji w pliku:** Odnajduje się odpowiedni wpis w katalogu i nadaje określoną wartość wskaźnikowi bieżącego położenia w pliku. Zmiana pozycji (ang. *repositioning*) w pliku nie wymaga w istocie zadnej operacji wejścia-wyjścia. Tę operację plikową nazywa się również *przeszukiem* (ang. *seek*) pliku.
- **Usuwanie pliku:** W celu usunięcia (ang. *deleting*) pliku odnajduje się jego nazwę w katalogu. Po odnalezieniu odpowiadającego jej wpisu katalogowego zwalnia się całą przestrzeń zajmowaną przez plik (aby mogły jej używać inne pliki) i likwiduje się dany wpis katalogowy*.
- **Skracanie pliku:** Zdarzają się sytuacje, w których użytkownik chce zachować niezmienione atrybuty pliku, lecz życzy sobie usunięcia jego wartości. Zamiast zmuszać użytkownika do usunięcia pliku i jego ponownego tworzenia, funkcja skracania pliku (ang. *truncating*) umożliwia pozostawienie niezmienionych atrybutów (z wyjątkiem długości), a jednocześnie ponowne ustalenie zerowej długości pliku**.

* Niektóre systemy nie usuwają od razu wpisu katalogowego pliku, co umożliwia podjęcie działań naprawczych w przypadku omyłkowego usunięcia pliku. – Przyp. tłum.

** Często skracanie pliku można odnieść do dowolnego jego miejsca. – Przyp. tłum.

Sześć opisanych wyżej operacji, to – rzecz oczywista – minimalny zbiór wymaganych działań na plikach. Do innych typowych operacji należą: *dopisywanie* (ang. *appending*) nowych informacji na końcu istniejącego pliku i *przemianowywanie* (ang. *renaming*) istniejącego pliku. Te elementarne operacje można łączyć w celu wykonywania innych operacji plikowych. Na przykład utworzenie kopii pliku lub przekopiowanie pliku na inne urządzenia wejścia-wyjścia, powiedzmy – na drukarkę lub ekran monitora, można uzyskać, tworząc nowy plik i czytając go z jednoczesnym zapisaniem do nowego pliku. Potrzebujemy też operacji pozwalających użytkownikowi na pobieranie i określanie różnych atrybutów pliku. Możemy na przykład potrzebować operacji umożliwiającej użytkownikowi określenie stanu pliku, na przykład jego długości, oraz operacji pozwalającej użytkownikowi ustalać atrybuty pliku, na przykład określać właściciela pliku.

Większość wymienionych operacji zawiera przeszukiwanie katalogu w celu odnalezienia w nim wpisu skojarzonego z plikiem. Aby uniknąć tego ciągłego przeszukiwania, wiele systemów *otwiera* plik, gdy ma on być użyty po raz pierwszy. System operacyjny przechowuje małą tablicę zawierającą informacje o wszystkich otwartych plikach (*tablicę otwartych plików*). Gdy trzeba wykonać operację plikową, wówczas używa się indeksu do tej tablicy, dzięki czemu nie ma żadnego szukania. Kiedy plik przestaje być aktywnie użytkowany, wtedy zostaje *zamknięty* przez proces i system operacyjny usuwa jego wpis z tablicy otwartych plików.

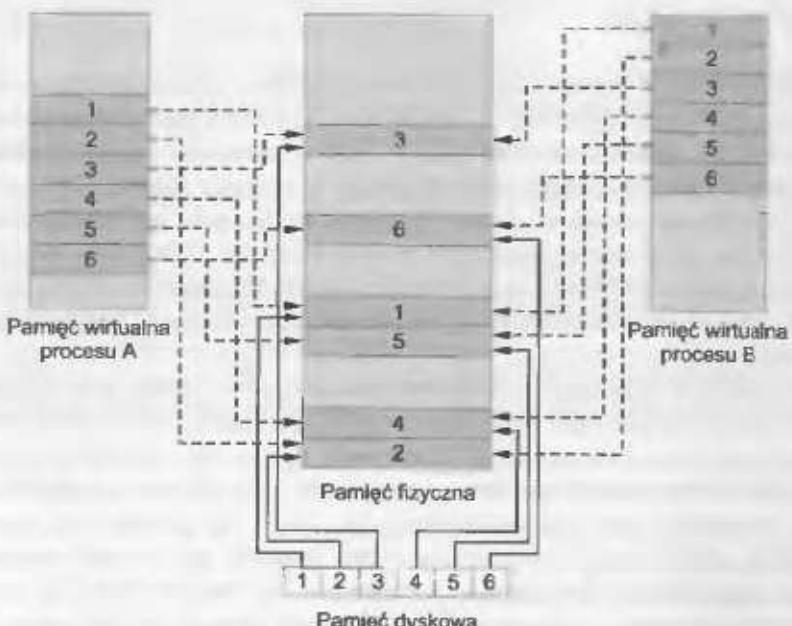
Niektóre systemy niejawnie otwierają plik przy pierwszym do niego odwołaniu. Po zakończeniu zadania lub programu, w którym otwarto plik, jest on jest automatycznie zamknięty. Jednak większość systemów wymaga jawnego otwierania plików przez programistę za pomocą funkcji systemowej *otwórz* (ang. *open*), która powinna być wywołana przed pierwszym użyciem pliku. Operacja otwierania pliku pobiera nazwę pliku, przegląda katalog i kopiuje odpowiedni wpis katalogowy do tablicy otwartych plików, jeśli ochrona pliku na to zezwala. Funkcja otwierania pliku przekazuje na ogół wskaźnik do wpisu w tablicy otwartych plików. Wszystkie operacje wejścia-wyjścia używają tego wskaźnika zamiast faktycznej nazwy pliku, dzięki czemu unika się dalszych przeszukiwań i upraszcza interfejs funkcji systemowych.

Implementacja operacji otwierania i zamknięcia (ang. *close*) pliku w środowisku z wieloma użytkownikami, jak na przykład w systemie UNIX, jest bardziej skomplikowana. W takim systemie plik może być otwarty przez kilku użytkowników jednocześnie. W systemie operacyjnym występują na ogół dwa poziomy tablic wewnętrznych. Procesowa tablica wszystkich plików otwartych w procesie zawiera informacje o sposobie korzystania z plików przez proces. Można w niej na przykład znaleźć bieżący wskaźnik dla każdego pliku, pokazujący miejsce w pliku, do którego będzie się odnosić następna operacja czytania lub pisania.

Każdy wpis w tablicy procesowej wskazuje z kolei na ogólnosystemową tablicę otwartych plików. Ogólnosystemowa tablica plików zawiera informacje niezależne od procesów, takie jak położenie pliku na dysku, daty dostępu i rozmiar pliku. Po otwarciu pliku w jednym procesie wykonanie przez inny proces funkcji *otwórz* spowoduje po prostu dodanie nowego wpisu do procesowej tablicy otwartych plików, z nowym wskaźnikiem bieżącym pliku i wskaźnikiem do odpowiedniej pozycji w tablicy ogólnosystemowej. Tablica otwartych plików zawiera na ogół licznik otwarć (ang. *open count*) skojarzony z każdym plikiem i pokazujący, w ilu procesach dany plik został otwarty. Każda operacja *zamknij* zmniejsza ten licznik, a gdy nastąpi jego wyzerowanie, plik przestaje być potrzebny i jego wpis zostaje usunięty z tablicy otwartych plików. Podsumowując, możemy powiedzieć, że z każdym otwartym plikiem jest związanych kilka elementów.

- **Wskaźnik plikowy:** Systemy, w których funkcje czytania i zapisywania pliku nie zawierają odległości w pliku, muszą śledzić miejsca ostatniego czytania lub pisania, utrzymując tzw. wskaźnik bieżącego położenia w pliku. Wskaźnik ten jest inny dla każdego procesu działającego na pliku, dlatego musi być przechowywany oddzielnie od dyskowych atrybutów pliku.
- **Licznik otwarć pliku:** Z chwilą zamknięcia pliku system operacyjny powinien powtórnie skorzystać z jego pozycji w tablicy otwartych plików, gdyż w przeciwnym razie mogłoby mu zabraknąć miejsca w tablicy. Ponieważ plik może być otwarty przez wiele procesów, system musi czekać z usunięciem wpisu w tablicy otwartych plików do ostatniego zamknięcia pliku. Licznik ten służy do nadzorowania liczby otwarć i zamknięć, przy czym staje się równy zeru po ostatnim zamknięciu. System może wtedy usunąć wpis.
- **Położenie pliku na dysku:** Większość operacji wymaga od systemu zmieniania danych wewnętrz pliku. Informacje potrzebne do zlokalizowania pliku na dysku są przechowywane w pamięci operacyjnej w celu unikania konieczności czytania ich z dysku podczas każdej operacji.

Niektóre systemy operacyjne umożliwiają *blokowanie* (ang. *locking*) części otwartego pliku w celu umożliwienia korzystania z niego wielu procesom: wspólnego użytkowania fragmentów pliku przez kilka procesów, a nawet odwzorowywania części pliku w pamięci operacyjnej w systemach pamięci wirtualnej. Ta ostatnia operacja, nazywana *odwzorowaniem pliku w pamięci* (ang. *memory mapping*), umożliwia logiczne przyporządkowanie części wirtualnej przestrzeni adresowej do danej sekcji pliku. Operacje czytania i zapi-



Rys. 10.1 Pliki odwzorowane w pamięci

sywania takiego obszaru pamięci traktuje się wówczas jako operacje czytania i zapisywania pliku, co znacznie upraszcza korzystanie z pliku. Zamknięcie pliku powoduje, że wszystkie dane odwzorowane w pamięci zostaną z powrotem zapisane na dysku i usunięte z pamięci wirtualnej procesu. Wiele procesów może odwzorować ten sam plik w swoich pamięciach wirtualnych w celu dzielenia danych. Operacje pisania wykonywane przez którykolwiek z takich procesów zmieniają dane w pamięci wirtualnej, a ich skutki mogą być oglądane przez wszystkie inne procesy, które odwzorowują ten sam fragment pliku. Powolując się na naszą wiedzę z rozdz. 9, powinniśmy bez trudu określić, jak implementuje się wspólne korzystanie z części pliku odwzorowanych w pamięci. Odwzorowanie pamięci wirtualnej w każdym procesie biorącym udział w dzieleniu odnosi się do tej samej strony pamięci fizycznej – takiej, która zawiera kopię bloku dyskowego. Tego rodzaju dzielenie pamięci jest zilustrowane na rys. 10.1. Aby koordynować dostęp do danych dzielonych, zainteresowane procesy mogą użyć któregoś z mechanizmów uzyskiwania wzajemnego wykluczania, omówionego w rozdz. 6.

10.1.3 Typy plików

Jednym z ważnych zagadnień projektowania systemu plików i całego systemu operacyjnego jest odpowiedź na pytanie, czy system operacyjny powinien rozpoznawać i obsługiwać typy plików. Jeśli system operacyjny rozpoznaje pliki różnych typów, to może na nich działać w rozsądny sposób. Ze zwykłym błędem mamy na przykład do czynienia wówczas, gdy użytkownik usiłuje wydrukować program wynikowy w postaci binarnej. Próba taka powoduje zazwyczaj wyprowadzanie śmieci, czemu można by zapobiec, gdyby system operacyjny był poinformowany o tym, że plik jest binarnym programem wynikowym.

Popularnym sposobem implementowania typów plików jest włączenie typu do nazwy pliku jako jej części. Nazwę pliku dzieli się na dwie części: nazwę i jej *rozszerzenie* (ang. *extension*), zazwyczaj oddzielone kropką (rys. 10.2). W ten sposób zarówno użytkownik, jak i system operacyjny mogą na podstawie samej nazwy określić typ pliku. Na przykład w systemie MS-DOS nazwa może zawierać do ośmiu znaków, po których występuje kropka i co najwyżej trzyznakowe rozszerzenie. System wykorzystuje to rozszerzenie jako godło, czyli typ pliku, wskazujące, jakiego rodzaju operacje są na danym pliku dozwolone. Wykonywać można na przykład tylko pliki mające rozszerzenia *com*, *exe* lub *bat*. Pliki z rozszerzeniami *com* lub *exe* są dwiema odmianami binarnych plików wykonywalnych, natomiast plik z rozszerzeniem *bat* jest makrodefinicją, czyli plikiem wsadowym (ang. *batch*), zawierającą polecenia dla systemu operacyjnego, zapisane w kodzie ASCII. System MS-DOS rozpoznaje tylko kilka rozszerzeń, jednak korzystają z nich również programy użytkowe do wskazywania typów używanych przez siebie plików. Na przykład asemblerzy przyjmują, że ich pliki źródłowe będą miały rozszerzenia *asm*, a procesor tekstu WordPerfect oczekuje plików z rozszerzeniami *wp*. Rozszerzenia takie nie są obowiązkowe: użytkownik może określić plik, nie podając rozszerzenia, a wówczas aplikacja będzie poszukiwać pliku o podanej nazwie i oczekiwany przez siebie rozszerzeniu. Ponieważ rozszerzenia takie nie są rozpoznawane przez system operacyjny, można je uważać za „wskazówki” dla korzystających z nich aplikacji.

Inny przykład użyteczności typów plików pochodzi z systemu operacyjnego TOPS-20. Jeśli użytkownik tego systemu spróbuje wykonać program wynikowy, którego plik źródłowy zmieniono (przeredagowano) już po utworzeniu pliku wynikowego, to plik źródłowy zostanie automatycznie na nowo skompilowany. Postępowanie takie zapewnia, że użytkownik zawsze będzie działać na aktualnych plikach wynikowych. W przeciwnym razie użytkownik mógłby marnować sporo czasu, wykonując program zawarty w starym pliku wynikowym. Zauważmy, że aby to było możliwe, system operacyjny musi

Typ pliku	Rozszerzenie nazwy	Funkcja
Wykonywalny	exe, com, bin lub brak rozszerzenia	gotowy do wykonania program w języku maszynowym
Wynikowy	obj, o	plik skompilowany, w języku maszynowym, nie skonsolidowany
Kod źródłowy	c, p, pas, f77, asm, a	kod źródłowy wyrażony w różnych językach
Wsadowy (makrodefinicja)	bat, sh	polecenia dla interpretera poleceń
Tekstowy	txt, doc	dane i dokumenty tekstowe
Plik edytora tekstu	wp, tex, rtf itp.	formaty plików różnych edytorów tekstu
Biblioteka	lib, a	biblioteki podprogramów dla programistów
Druk lub obraz	ps, dvi, gif	plik binarny lub ASCII w formacie zdatnym do drukowania lub oglądania
Archiwalny	arc, zip, tar	grupa powiązanych ze sobą plików zmagazynowana w celach archiwalnych w jednym pliku, niekiedy o zmniejszonej objętości

Rys. 10.2 Popularne typy plików

umieć odróżnić plik źródłowy od wynikowego, sprawdzić czas ostatniej zmiany lub utworzenia każdego z tych plików oraz określić język programu źródłowego (w celu zastosowania odpowiedniego kompilatora).

Weźmy pod uwagę system operacyjny Apple Macintosh. W systemie tym każdy plik ma typ, na przykład *text* lub *pict*^{*}. Każdy plik jest także zaopatrzony w atrybut swojego twórcy, zawierający nazwę programu, który go utworzył. Atrybut ten jest określany przez system operacyjny podeczas tworzenia pliku, jego zastosowanie zależy więc od systemu i podlega jego kontroli. Na przykład plik utworzony przez procesor tekstu ma jako swojego twórcę oznaczoną nazwę danego procesora tekstu. Gdy użytkownik otwiera ten plik przez wskazanie reprezentującej go ikony i dwukrotne naciśnięcie przycisku myszki, wówczas następuje automatyczne wywołanie procesora tekstu i załadowanie pliku gotowego do redagowania.

* Tekst lub obraz. – Przyp. tłum.

sektorów. Wszystkie dyskowe operacje wejścia-wyjścia są wykonywane w jednostkach różnych pojedynczemu blokowi (rekord fizyczny), a wszystkie bloki są tego samego rozmiaru. Rzadko kiedy rozmiar rekordu fizycznego pasuje dokładnie do długości potrzbnego rekordu logicznego. Rekordy logiczne mogą być ponadto różnej długości. *Upakowywanie* (ang. *packing*) pewnej liczby rekordów logicznych w fizycznych blokach jest typowym rozwiązaniem tego problemu.

Na przykład w systemie operacyjnym UNIX wszystkie pliki są definiowane po prostu jako strumienie bajtów. Każdy bajt jest indywidualnie adresowalny za pomocą jego odległości od początku (lub końca) pliku. W tym przypadku rekord logiczny ma długość 1 bajta. System plików wedle potrzeby automatycznie pakuje i rozpakowuje bajty w blokach fizycznych (powiedzmy, zawierających po 512 B).

Rozmiar rekordu logicznego, rozmiar bloku fizycznego oraz sposób pakowania przesądzają o liczbie rekordów logicznych mieszczących się w każdym bloku fizycznym. Upakowywanie może być wykonywane przez aplikację użytkownika lub przez system operacyjny.

W każdym przypadku plik można rozpatrywać jako ciąg bloków. Wszystkie podstawowe operacje wejścia-wyjścia działają na blokach. Zamiana rekordów logicznych na bloki fizyczne jest względnie prostym zadaniem do zaprogramowania.

Zwrócić uwagę na to, że konsekwentne przydzielanie przestrzeni dyskowej w postaci bloków, ogólnie biorąc, może prowadzić do marnowania części ostatniego bloku w każdym pliku. Jeśli bloki mają po 512 B, to plik o wielkości 1949 B otrzyma 4 bloki (2048 B), przy czym ostatnie 99 B zostanie zmarnowanych. Należy zatem przydzielać bajty będące wynikiem konieczności utrzymywania wszystkiego w blokach (zamiast w bajtach) są określane jako *fragmentacja wewnętrzna* (ang. *internal fragmentation*). Wszystkie systemy plików są obciążone fragmentacją wewnętrzną; im większe są bloki, tym większa jest fragmentacja wewnętrzna.

10.2 ■ Metody dostępu

Pliki przechowują informacje. Kiedy informacje ma być uzyta, wtedy trzeba uzyskać do niej dostęp, a następnie przeczytać ją do pamięci operacyjnej komputera. Istnieje kilka sposobów uzyskiwania dostępu do informacji zawartej w pliku. Niektóre systemy umożliwiają tylko jeden rodzaj dostępu do plików. W innych systemach, jak na przykład tych wywodzących się z IBM, istnieje wiele różnych metod dostępu i wybór właściwego dla danego zastosowania jest ważnym zagadnieniem projektowym.



Rys. 10.3 Plik o dostępie sekwencyjnym

10.2.1 Dostęp sekwencyjny

Najprostszą metodą dostępu jest *dostęp sekwencyjny* (ang. *sequential access*). Informacje w pliku są przetwarzane po kolei, jeden rekord za drugim. Ten rodzaj dostępu jest zapewne najpowszechniejszy; na przykład edytory i kompilatory zazwyczaj używają plików w ten sposób.

Ogromna większość operacji wykonywanych na plikach to czytanie i pisanie. Operacja czytania pobiera następną porcję pliku i automatycznie przesuwa do przodu wskaźnik położenia w pliku, określający miejsce następnej operacji wejścia-wyjścia. Z kolei operacja pisania umieszcza dane na końcu pliku i ustawia wskaźnik za nowo zapisanymi danymi (nowy koniec pliku). Wskaźnik położenia w takim pliku można ustawić na jego początku, a w niektórych systemach – program może przeskakiwać w przód lub wstecz n rekordów, dla pewnej wartości n (być może tylko dla $n = 1$). Dostęp sekwencyjny jest zilustrowany na rys. 10.3. Dostęp sekwencyjny jest oparty na taśmowym modelu pliku i jest odpowiedni zarówno w odniesieniu do urządzeń działających sekwencyjnie, jak i tych, które wykonują dostępy swobodne.

10.2.2 Dostęp bezpośredni

Inną metodą dostępu jest *dostęp bezpośredni** (ang. *direct access*), inaczej – *dostęp względny* (ang. *relative access*). Plik składa się z rekordów logicznych o stałej długości, które mogą być natychmiast czytane i zapisywane przez programy, bez zachowywania jakiegokolwiek szczególnego porządku. Metoda dostępu bezpośredniego opiera się na dyskowym modelu pliku, ponieważ dysk umożliwia swobodny dostęp do dowolnego bloku pliku. W dostępie bezpośrednim plik traktuje się jako ciąg ponumerowanych bloków lub rekordów. Plik o dostępie bezpośredniem pozwala na czytanie lub zapisywanie dowolnych bloków. Można zatem przeczytać blok 14 i zaraz potem blok 53, po czym zapisać blok 7. W pliku o dostępie bezpośredniem nie ma żadnych ograniczeń co do kolejności operacji czytania lub pisania.

* Nazywany również *swobodnym* – Przyp. tłum.

Pliki o dostępie bezpośredniem są szeroko stosowane ze względu na natychmiastowość dostępu do wielkich ilości informacji. Często używa się ich w bazach danych. Kiedy pojawia się zapytanie dotyczące konkretnego tematu, wtedy oblicza się, który blok zawiera odpowiedź, a następnie czyta bezpośrednio ten blok, udostępniając potrzebną informację.

Na przykład w systemie rezerwacji miejsc w liniach lotniczych całą informację dotyczącą konkretnego lotu (np. lotu 713) można zapamiętać w bloku identyfikowanym za pomocą numeru lotu. Tak więc liczba wolnych miejsc na lot 713 jest zapamiętana w 713 bloku głiku rezerwacji. Do zapamiętywania informacji o większym zbiorze, takim jak ludność, można obliczać wartość funkcji haszującej określonej na zbiorze nazwisk lub przeszukiwać mały indeks, przechowywany na stałe w pamięci operacyjnej, w celu określenia bloku, który trzeba przeczytać i przeszukać.

Operacje plikowe muszą być tak zmienione, aby zawierały jako parametr numer bloku. Mamy więc *czytaj n*, przy czym *n* jest numerem bloku, zamiast *czytaj dalej*, i *pisz n* zamiast *pisz dalej*. Można też pozostawić operacje *czytaj dalej* i *pisz dalej* – jak w dostępie sekwencyjnym – oraz dodać operację *przejdi do n*, w której *n* jest numerem bloku w pliku. Wówczas, by uzyskać efekt operacji *czytaj n*, wykonuje się operację *przejdi do n*, a potem *czytaj dalej*.

Numer bloku przekazywany przez użytkownika do systemu operacyjnego jest *zarwyczaj numerem względnym bloku* (ang. *relative block number*). Numer względny bloku jest indeksem względem początku pliku. Zatem pierwszy blok względny w pliku ma numer 0, następny 1 itd., nawet jeśli faktyczny, dyskowy adres bezwzględny danego bloku wynosi 14 703 dla pierwszego bloku i 3 192 dla drugiego. Użycie względnych numerów bloków pozwala systemowi operacyjnemu decydować o tym, gdzie umieścić plik (co zwie się problemem przydziału i jest omówione w rozdz. 11) i zapobiega ślepninie przez użytkownika do fragmentów systemu plików nie będących częścią jego pliku. W niektórych systemach numery względne bloków rozpoczynają się od 0, w innych zaś od 1.

Jeśli jest znana długość *L* rekordu logicznego, to zamówienie na rekord *N* jest przekształcane na zamówienie wejścia-wyjścia dotyczące *L* bajtów od adresu *L * (N - 1)* wewnątrz pliku. Ponieważ rekordy logiczne mają stały rozmiar, więc łatwo jest je także czytać, zapisywać lub usuwać.

Nie wszystkie systemy operacyjne realizują w odniesieniu do plików zarówno dostęp sekwencyjny, jak i bezpośredni. Niektóre systemy pozwalają tylko na sekwencyjny dostęp do plików, w innych jest możliwy tylko dostęp bezpośredni. Pewne systemy wymagają, aby plik został zdefiniowany jako sekwencyjny lub bezpośrednio dostępny przy jego tworzeniu; do pliku takiego wolno się zwracać tylko w sposób zgodny z jego deklaracją. Zauważmy

Dostęp sekwencyjny	Implementacja przy dostępie bezpośredniem
przejdi na początek	$pb := 0;$
czytaj dalej	$czytaj pb;$ $pb := pb + 1;$
pisz dalej	$pisz pb;$ $pb := pb + 1;$

Rys. 10.4 Symulacja dostępu sekwencyjnego za pomocą pliku o dostępie bezpośredniem

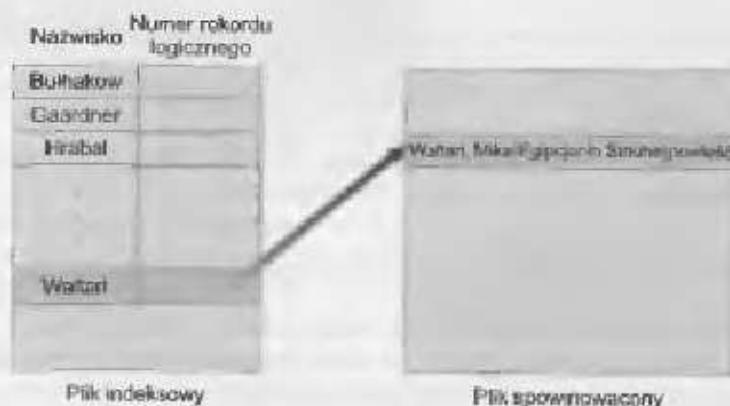
jednak, że dostęp sekwencyjny łatwo zasympułować w pliku o dostępie bezpośredniem. Mając po prostu zmienną pb określającą położenie bieżące w pliku, możemy symułować operacje dostępu sekwencyjnego, tak jak na rys. 10.4. Symulowanie dostępu bezpośredniego za pomocą pliku sekwencyjnego jest natomiast skrajnie nieefektywne.

10.2.3 Inne metody dostępu

Metoda dostępu bezpośredniego może służyć za podstawę do opracowywania innych metod dostępu. Te dodatkowe metody z zasady zawierają konstrukcję indeksu (ang. *index*) pliku. Indeks ten, podobnie jak skortowidz na końcu książki, zawiera wskaźniki do różnych bloków. Aby znaleźć jakąś pozycję w pliku, przeszukuje się najpierw indeks, a następnie używa wskaźnika w celu uzyskania bezpośredniego dostępu do pliku i odnalezienia potrzebnego wpisu.

Na przykład plik cen detalicznych może zawierać wykaz uniwersalnych kodów produktów wraz z cenami tych produktów. Każda pozycja w wykazie składa się z 10-cyfrowego kodu produktu i 6-cyfrowej ceny – zawartych w 16-bajtowym rekordzie. Jeśli bloki naszego dysku mają po 1024 B, to w jednym bloku możemy zapamiętać 64 pozycje. Plik złożony ze 120 000 pozycji zajmowałby około 2000 bloków (2 miliony bajtów). Dla pliku posortowanego według kodów produktów możemy zdefiniować indeks złożony z pierwszych kodów z każdego bloku. Indeks ten będzie miał 2000 pozycji (każda po 10 cyfr), czyli 20 000 B, a zatem będzie mógł się znajdować w pamięci operacyjnej. Aby znaleźć cenę danego artykułu, możemy przeszukać (binarnie) indeks. Na podstawie tego przeszukania będziemy wiedzieli dokładnie, który blok zawiera potrzebny wpis, będziemy więc mogli pobrać ten blok. Struktura taka umożliwia przeszukiwanie wielkiego pliku przy użyciu niewielu operacji wejścia-wyjścia.

Przy wielkich plikach sam plik indeksowy może okazać się za duży, aby można go było trzymać w pamięci operacyjnej. Jedno z rozwiązań polega na utworzeniu indeksu pliku z indeksem. Pierwotny plik indeksowy będzie zawierał wskaźniki do wtórnego pliku indeksowego, który będzie wskazywał na rzeczywiste dane.



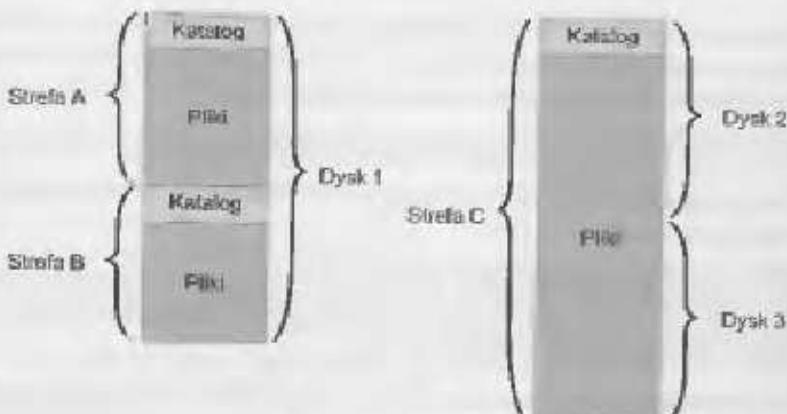
Rys. 10.5 Przykład pliku indeksowego i pliku spowinowaconego

Na przykład w stosowanej w IBM metodzie indeksowanego dostępu sekwencyjnego (ISAM) używa się małego indeksu głównego, który wskazuje na bloki dyskowe wtórnego indeksu. Bloki wtórnego indeksu wskazują na właściwe bloki pliku. Plik jest przechowywany w postaci posortowanej według zdefiniowanego klucza. Aby znaleźć konkretną jednostkę, najpierw przeszukuje się binarnie indeks główny i uzyskuje numer bloku indeksu wtórnego. Blok ten czyta się i znów za pomocą metody przeszukiwania binarnego odnajduje się blok zawierający wymagany rekord. Na koniec blok trzeba przeszukać liniowo. W ten sposób dowolny rekord można zlokalizować na podstawie jego klucza za pomocą co najwyżej dwu operacji czytania wykonywanych w trybie dostępu bezpośredniego. Na rysunku 10.5 jest pokazana podobna sytuacja, zaimplementowana w systemie VMS przez indeks i pliki spowinowacone.

10.3 ■ Struktura katalogowa

Komputerowe systemy plików mogą być rozległe. W niektórych systemach przechowuje się tysiące plików w setkach gigabajtów pamięci dyskowej. Aby zarządzać wszystkimi tymi danymi, musimy umieć je organizować. Organizowanie to przeprowadza się zazwyczaj w dwu częściach. Po pierwsze, system plików jest podzielony na *strefy* (ang. *partition*), w środowisku IBM nazywane też *minidyskami* (ang. *minidisks*), a w kregu komputerów PC i Macintosh określane jako *tomy** (ang. *volumes*). Na ogólny każdy dysk w systemie

* Także: *medium*, *volume*, *disk*, *logusz*; niejednoznacznie – „dyski” lub żargonowe „partycje”. – Przyp. tłum.



Rys. 10.6 Typowa organizacja systemu plików

zawiera co najmniej jedną strefę, która jest strukturą niskiego poziomu mieszczącą pliki i katalogi. Czasami używa się stref w celu uzyskiwania kilku różnych obszarów na jednym dysku, traktowanych jako oddzielne urządzenia pamięci. W innych systemach występują strefy w postaci logicznych struktur kilku dysków, co umożliwia zwiększenie rozmiarów stref ponad pojemność pojedynczego dysku. Dzięki temu użytkownik ma do czynienia tylko z katalogiem logicznym i strukturą plików, może więc całkiem pomijać zagadnienia fizycznego przydziela miejsca dla plików. Z tych powodów strefy mogą być pojmowane jako dyski wirtualne.

Po drugie, każda strefa zawiera informacje o zawartych w niej plikach. Informacje te są przechowywane w pozycjach *katalogu urządzenia* (ang. *device directory*), inaczej – w *tablicy zawartości toru*. Katalog urządzenia (częściej zwany po prostu „katalogiem”) zawiera dla każdego pliku danej strefy informacje takie, jak: nazwa, położenie, rozmiar oraz typ. Na rysunku 10.6 jest pokazana typowa organizacja systemu plików.

Katalog można uważać za tablicę symboli tłumaczącej nazwy plików na ich wpisy katalogowe. Jeśli przyjmiemy taki punkt widzenia, to stanie się widoczne, że sam katalog można zorganizować na wiele sposobów. Chcemy mieć możliwość dodawania pozycji w katalogu, ich usuwania, poszukiwania nazwanego wpisu oraz wyprowadzania zawartości wszystkich pozycji katalogu. W rozdziale 11 omawiamy struktury danych odpowiednie do implementowania struktury katalogowej. Zajmiemy się teraz schematami służącymi do definiowania logicznej struktury systemu katalogów. Rozważając daną strukturę katalogową, powinniśmy wziąć pod uwagę operacje, które będą wykonywane na katalogu:

- **Odnajdywanie pliku:** Musimy mieć możliwość przeszukiwania struktury katalogowej w celu odnalezienia w niej pozycji odpowiadającej kon-

kretnemu plikowi. Ponieważ pliki mają nazwy symboliczne, a podobne nazwy mogą uwidaczniać związki między plikami, chcielibyśmy móc odnajdować wszystkie pliki, które dają się dopasować do danego wzorca.

- **Tworzenie pliku:** Chcemy tworzyć nowe pliki i dodawać je do katalogu.
- **Usuwanie pliku:** Gdy plik przestaje być potrzebny, wtedy chcemy go usunąć z katalogu.
- **Wyprowadzanie katalogu:** Chcemy mieć możliwość sporządzania wykazu plików w danym katalogu i umieszczania w tym wykazie zawartości pozycji katalogowej dla każdego pliku.
- **Przemianowywanie pliku:** Ponieważ nazwa pliku kojarzy się użytkownikom z jego zawartością, powinna dać się zmienić, gdy zawartość lub przeznaczenie pliku ulegają zmianie. Przemianowywanie pliku powinno również umożliwiać zmianę jego miejsca w strukturze katalogowej.
- **Obchód systemu plików:** Pozytyczną możliwością jest uzyskiwanie dostępu do każdego katalogu i pliku w obrębie struktury katalogowej. Ze względu na niezawodność warto w regularnych odstępach czasu zapamiętywać zawartość i strukturę całego systemu plików. Często sprawdza się to do kopирования wszystkich plików na taśmę magnetyczną. W wyniku takiego postępowania można uzyskać zapasową kopię na wypadek awarii systemu lub wtedy, gdy plik po prostu wychodzi z użycia. W tym przypadku plik może zostać skopiowany na taśmę, a jego przestrzeń dyskowa zwolniona do ponownego użytku przez inny plik.

W punktach 10.3.1-10.3.5 omawiamy najpopularniejsze schematy definiowania logicznej struktury katalogu.

10.3.1 Katalog jednopoziomowy

Najprostszą strukturą katalogową jest katalog jednopoziomowy. Wszystkie pliki są ujęte w tym samym katalogu, który jest łatwo utworzyć i obsługiwać (rys. 10.7).



Rys. 10.7 Katalog jednopoziomowy

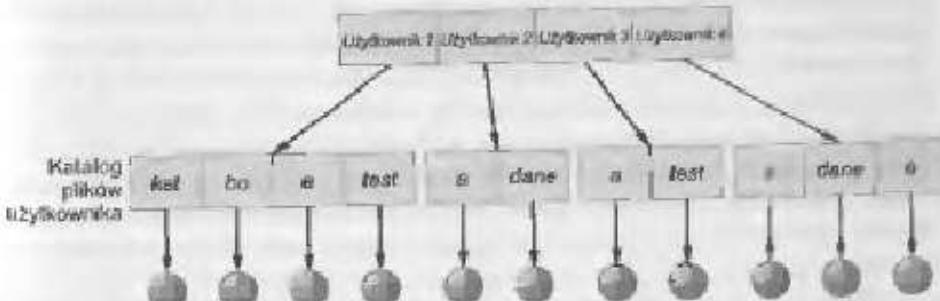
Katalog jednopoziomowy ma jednak spore ograniczenia wówczas, gdy wzrasta liczba plików lub gdy istnieje więcej użytkowników niż jeden. Ponieważ wszystkie pliki są w tym samym katalogu, muszą mieć jednoznaczne nazwy. Jeśli mamy dwoje użytkowników, którzy swoim plikom danych nadają nazwę *test*, to reguła jednoznacznych nazw będzie naruszona. (Na przykład 23 uczniów w pewnej klasie programowania nadali programowi będącemu ich drugim zadaniem nazwę *prog2*, a 11 innych uczniów nazwało go *zad2*). Choć nazwy plików są na ogół tak dobierane, aby odzwierciedlać zawartości plików, ich długość jest często ograniczona (system MS-DOS dopuszcza tylko 11-znakowe nazwy plików, UNIX zezwala na 255 znaków).

Nawet dla jednego użytkownika pamiętanie nazw wszystkich plików w celu tworzenia wyłącznie plików o jednoznacznych nazwach staje się trudne w miarę wzrostania liczby plików. Nie jest niczym niezwykłym posiadanie setek plików w jednym systemie komputerowym i takiej samej liczby dodatkowych plików w innym systemie. W takim środowisku utrzymywanie informacji o tak wielu plikach staje się poważnym problemem.

10.3.2 Katalog dwupoziomowy

Podstawową wadą katalogu jednopoziomowego są kolizje nazw plików należących do różnych użytkowników. Standardowym rozwiązaniem jest utworzenie *oddzielnego katalogu* dla każdego użytkownika.

W strukturze katalogu dwupoziomowego każdy użytkownik ma własny *katalog plików użytkownika* (ang. *user file directory* – UFD). Każdy katalog plików użytkownika ma podobną budowę, ale odnosi się tylko do plików jednego użytkownika. Gdy rozpoczyna się nowe zadanie użytkownika albo użytkownik rejestruje się w systemie, wówczas przegląda się główny *katalog plików* (ang. *master file directory* – MFD). Główny katalog plików jest indeksowany nazwami użytkowników lub numerami kont, a każdy jego wpis wskazuje na katalog odpowiadający danemu użytkownikowi (rys. 10.8).



Rys. 10.8 Struktura katalogu dwupoziomowego

Gdy użytkownik odnosi się do jakiegoś pliku, wówczas przeszukuje się tylko jego własny katalog. Dzięki temu różni użytkownicy mogą mieć pliki z tymi samymi nazwami, bacząc tylko, aby nazwy wszystkich plików w obrębie ich katalogów były jednoznaczne.

Tworząc plik dla użytkownika, system operacyjny przegląda tylko katalog danego użytkownika, aby upewnić się, że nie występuje tam taka sama nazwa pliku. Przy usuwaniu pliku system operacyjny ogranicza poszukiwanie do lokalnego katalogu plików użytkownika; nie może zatem przypadkowo usunąć innego pliku o takiej samej nazwie, lecz należącego do innego użytkownika.

Tworzeniu i usuwaniu podlegają zależnie od potrzeb również katalogi użytkowników. W tym celu jest wykonywany specjalny program systemowy, któremu przekazuje się odpowiednią nazwę użytkownika i informacje dotyczące rozliczania. Program ten tworzy katalog plików nowego użytkownika i dodaje nową pozycję do głównego katalogu plików. Wykonywanie takiego programu może być ograniczone tylko do administratorów systemu. Przydzielenia miejsca na dysku przeznaczonego na katalogi użytkowników można dokonywać za pomocą tych samych sposobów, które opisano w rozdz. 11 w odniesieniu do plików.

Struktura katalogu dwupoziomowego rozwiązuje problem kolizji nazw, lecz nadal ma pewne wady. Umożliwia ona skuteczne odizolowanie jednego użytkownika od drugiego. Jest to korzystne wówczas, gdy użytkownicy są całkowicie niezależni, lecz niepożądane wtedy, gdy chcą współpracować ze sobą w jakimś zadaniu i mieć wzajemny dostęp do swoich plików. Niektóre systemy po prostu nie pozwalały na to, aby do lokalnych plików użytkownika mieli dostęp inni użytkownicy.

Jeśli taki dostęp ma być dopuszczalny, to dany użytkownik musi mieć możliwość nazwania pliku w katalogu innego użytkownika. W celu jednoznacznego nazwania jakiegoś pliku w katalogu dwupoziomowym, należy podać zarówno nazwę użytkownika, jak i nazwę pliku. Katalog dwupoziomowy można uważać za odwrócone drzewo o wysokości 2. Korzeniem drzewa jest katalog główny. Jego bezpośrednimi potomkami są katalogi plików użytkowników. Potomkami katalogów plików użytkowników są same pliki. Pliki są liśćmi drzewa. Określając nazwę użytkownika i nazwę pliku, definiuję się ścieżkę w drzewie wiodącą od korzenia (główny katalog plików) do liścia (określony plik). Zatem nazwa użytkownika i nazwa pliku definiują nazwę ścieżki (ang. *path name*). Każdy plik w systemie ma nazwę ścieżki. W celu jednoznacznego nazwania pliku użytkownik musi znać nazwę ścieżki dla danego pliku.

Na przykład, jeśli użytkownik A chce sięgnąć po własny plik o nazwie *test*, to odwołuje się do niego po prostu za pomocą nazwy *test*. Jednak aby

dotrzeć do pliku testowego użytkownika *B* (który ma w katalogu nazwę *użytkownik-B*), musi on posłużyć się odwołaniem *użytkownik-B/test*. Każdy system ma swoją składnię nazw identyfikujących pliki w katalogach innych niż należący do danego użytkownika.

Do określenia strefy dyskowej pliku stosuje się dodatkową składnię. Na przykład w systemie MS-DOS strefę określa się za pomocą litery i występującego po niej dwukropka. Określenie pliku mogłoby zatem mieć postać: *c:\uzytk-b\test*. W niektórych systemach posunięto się jeszcze dalej, oddzielając od siebie nazwę strefy, katalogu i pliku. I tak w systemie VMS plik *login.com* może mieć określenie *u:ss1:jdeck\login.com:1*, gdzie *u* jest nazwą strefy, *ss1* jest nazwą katalogu, *jdeck* oznacza podkatalog, a *1* jest numerem wersji. W innych systemach nazwę strefy traktuje się po prostu jako część nazwy katalogu. Na samym początku występuje nazwa strefy, a po niej nazwa katalogu i pliku. Na przykład nazwa */u/pbg/test* może określać strefę *u*, katalog *pbg* i plik *test*.

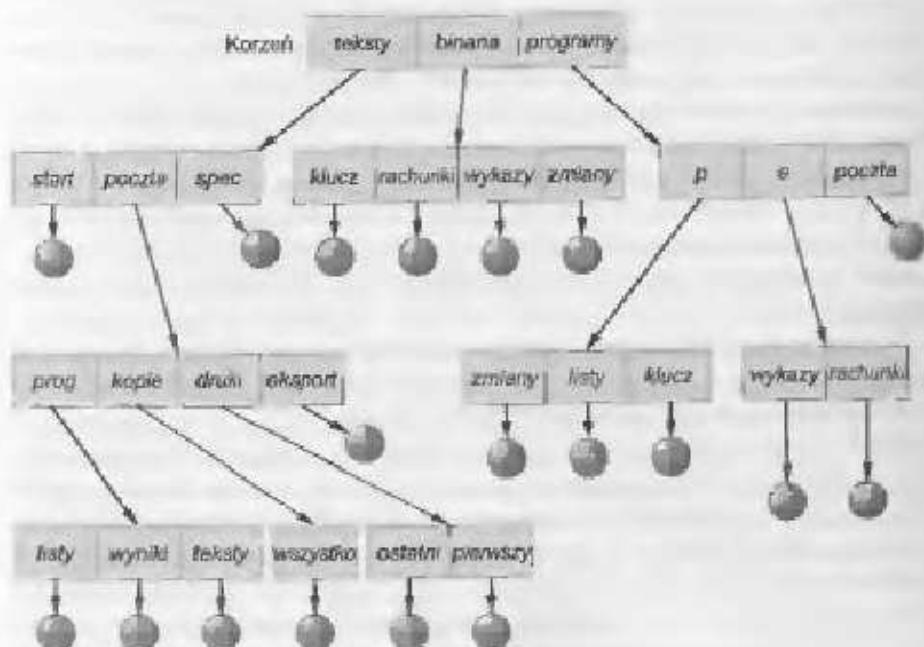
Specjalna sytuacja powstaje w przypadku plików systemowych. Programy dostarczane jako część systemu (programy ładowające, assemblery, kompilatory, procedury narzędziowe, biblioteki itp.) są z zasady przechowywane w postaci plików. Wskutek wydania systemowi operacyjnemu odpowiedniego polecenia pliki te są czytane przez program ładowający i wykonywane. Więcej interpreterów polecen traktuje samą nazwę polecenia jako nazwę pliku do załadowania i wykonania. Zgodnie z tym, co na razie powiedziano o systemie katalogowym, nazwy takiego pliku należałoby poszukiwać w bieżącym katalogu użytkownika. Można by zatem skopiować pliki systemowe do katalogu każdego użytkownika. Jednak kopiowanie wszystkich plików systemowych spowodowałoby olbrzymie zamieszanie w miejscu. (Jeżeli pliki systemowe zajmują 5 MB, to dostarczenie ich 12 użytkownikom wymagałoby 60 ($5 \times 12 = 60$) MB samych tylk kopii plików systemowych).

Standardowe rozwiązanie polega na niewielkim skomplikowaniu procedury przeszukiwania. Określa się specjalny katalog na przechowywanie plików systemowych (np. *użytkownik-0*). Za każdym razem, gdy wystąpi nazwa pliku do załadowania, system operacyjny przeszukuje najpierw katalog lokalny użytkownika. Jeśli znajdzie w nim dany plik, to plik ten zostanie użyty. W przeciwnym razie system automatycznie przeszukuje katalog specjalnego użytkownika, w którym są gromadzone pliki systemowe. Ciąg katalogów przeszukiwanych w celu odnalezienia nazwy pliku jest nazywany ścieżką wyszukiwania (ang. *search path*). Pomyśl ten można rozwijać tak, aby ścieżka wyszukiwania zawierała dowolnej długości wykaz katalogów do przeszukania w związku z podaną nazwą polecenia. Tę metodę bardzo często stosuje się w systemach UNIX i MS-DOS.

10.3.3 Katalogi o strukturach drzewiastych

Skoro wiemy już, jak można katalog dwupoziomowy przedstawić jako dwupoziomowe drzewo, to naturalnym uogólnieniem jest rozszerzenie struktury katalogowej do postaci drzewa o dowolnej wysokości (rys. 10.9). Uogólnienie takie pozwala użytkownikom na tworzenie własnych podkatalogów i stosowną do tego organizację ich plików. Na przykład system plików MS-DOS ma strukturę drzewa. Drzewo jest w istocie najpowszechniejszą strukturą katalogową. Drzewo ma katalog główny (korzeń). Każdy plik w systemie ma jednoznaczną nazwę ścieżki. Nazwą ścieżki jest ścieżka wiodąca od korzenia, przez wszystkie pośredniczące podkatalogi aż do określonego pliku.

Katalog (lub podkatalog) zawiera zbiór plików lub podkatalogów. Katalog jest po prostu jeszcze jednym plikiem, lecz traktowanym w specjalny sposób. Wszystkie katalogi mają taką samą wewnętrzną budowę. Jeden bit w każdym wpisie katalogowym określa, czy dany wpis dotyczy pliku (0) czy podkatalogu (1). Do tworzenia i usuwania katalogów używa się specjalnych funkcji systemowych.



Rys. 10.9 Katalog o strukturze drzewiastej

Podczas normalnej pracy każdy użytkownik ma *katalog bieżący* (ang. *current directory*)*. Katalog bieżący powinien zawierać większość plików, którymi aktualnie użytkownik jest zainteresowany. Przy każdym odniesieniu do pliku przegląda się katalog bieżący. Jeśli jest potrzebny plik, którego nie ma w katalogu bieżącym, to użytkownik musi określić nazwę jego ścieżki albo zmienić katalog bieżący, aby stał się nim katalog z danym plikiem. Do zmiany katalogu bieżącego na inny stosuje się funkcję systemową, która przyjmuje nazwę katalogu jako parametr i używa jej do określenia nowego katalogu bieżącego. Tak więc użytkownik może stosownie do potrzeb zmieniać katalog bieżący. Po wykonaniu funkcji systemowej *zmień katalog*, do chwili następnego jej wywołania wszystkie wywołania systemowe *otwórz* będą poszukiwać plików w katalogu bieżącym.

Początkowy katalog bieżący użytkownika jest wyznaczany w chwili rozpoczęcia zadania użytkownika lub wtedy, kiedy użytkownik rejestruje się w systemie. System operacyjny przegląda plik kont (lub jakieś inne, z góry ustalone miejsce), aby znaleźć dane o użytkowniku (w celach rozliczeniowych). W pliku kont znajduje się wskaźnik (lub nazwa) początkowego katalogu użytkownika. Wskaźnik ten kopiuje się do zmiennej lokalnej związanego z danym użytkownikiem, która określa początkowy katalog bieżący użytkownika.

Nazwy ścieżek mogą być dwu rodzajów: bezwzględne lub względne. Bezwzględne nazwa ścieżki zaczyna się od korzenia i biegnie w głąb do określonego pliku, zawierając nazwy napotykanych po drodze katalogów. Względna nazwa ścieżki określa drogę od bieżącego katalogu. Jeśli na przykład w drzewistej strukturze systemu plików z rys. 10.10 bieżącym katalogiem będzie *korzen/teksty/poeta*, to względna nazwa ścieżki *druki/pierwszy* odnosi się do tego samego pliku co absolutna nazwa ścieżki *korzen/teksty/poeta/druki/pierwszy*.

Pozwalając użytkownikowi na definiowanie własnych podkatalogów, daje się mu możliwość kształtowania struktury jego plików. Struktura ta może przybrać postać oddzielnych katalogów dla plików związanych z różnymi tematami (np. do przechowywania tekstu tej książki został założony podkatalog) lub dla różnych form informacji (np. katalog *programy* może zawierać programy źródłowe; katalog *binaria* – wszystkie pliki binarne).

Interesującą decyzją polityczną związaną z drzewistą strukturą katalogową jest sposób postępowania przy usuwaniu katalogu. Jeśli katalog jest pusty, to można po prostu usunąć jego wpis z zawierającego go katalogu. Założymy jednak, że zakwalifikowany do usunięcia katalog nie jest pusty, ale zawiera pewną liczbę plików lub – być może – podkatalogów. Można wybrać jeden z dwóch sposobów. Niektóre systemy, na przykład MS-DOS, nie

* Niekiedy nazywany też katalogiem aktywnym lub roboczym. – Przyp. tłum.

usuń katalogu, jeśli nie jest on pusty. Toteż aby skasować katalog, użytkownik musi najpierw usunąć wszystkie znajdujące się w nim pliki. Jeśli występują tam jakieś podkatalogi, to procedurę tę należy zastosować w odniesieniu do nich rekurencyjnie, tak aby można było je również usunąć. Podejście takie może wymagać sporo pracy.

Drugi sposób, używany na przykład w poleceniu *rmdir* systemu UNIX, polega na dopuszczeniu możliwości, że jeśli nadeszło zamówienie na usunięcie katalogu, to wszystkie jego pliki i podkatalogi również kwalifikują się do usunięcia. Zauważmy, że każdy z tych sposobów jest dość łatwy do zimplementowania, wybór jest kwestią przyjęcia jakieś polityki. Drugie postępowanie jest wygodniejsze, lecz bardziej niebezpieczne, gdyż jednym poleceniem można usuwać całą strukturę katalogową. Gdyby polecenie to wynikało z błędnej decyzji, wówczas z taśm archiwalnych należałoby odtworzyć dużą liczbę plików i katalogów.

W systemie katalogów o drzewistej budowie użytkownicy mogą mieć dostęp nie tylko do własnych plików, lecz również do plików innych użytkowników. Na przykład użytkownik *B* może sięgać do plików użytkownika *A*, jeśli określi nazwy ich ścieżek. Użytkownik *B* może określić bezwzględną lub względową nazwę ścieżki. Może on również spowodować, aby jego katalogiem bieżącym był katalog użytkownika *A*, po czym może sięgać po pliki za pomocą ich nazw. Niektóre systemy pozwalają również użytkownikom definiować własne ścieżki przeszukiwania. W takim przypadku użytkownik *B* mógłby zdefiniować swoją ścieżkę przeszukiwania jako zawierającą kolejno: (1) jego lokalny katalog, (2) katalog plików systemowych i (3) katalog użytkownika *A*. Wówczas dopóki nazwa pliku użytkownika *A* nie pozostałaaby w konflikcie z jakąś nazwą pliku lokalnego lub systemowego, do płyty do pliku użytkownika *A* można by się odnosić po prostu za pomocą nazwy tego pliku.

Zauważmy, że ścieżka do pliku w katalogu drzewiastym może być dłuższa niż w katalogu dwupoziomowym. Aby umożliwić użytkownikom dostęp do programów bez konieczności pamiętania takich długich ścieżek, w systemie operacyjnym komputera Macintosh zautomatyzowano wyszukiwanie programów wykonywalnych. System utrzymuje plik *Desktop File*, zawierający nazwy i lokalizacje wszystkich znanych mu programów zdolnych do wykonywania. Kiedy w systemie przybywa nowy dysk twardy bądź elastyczny lub następuje połączenie z siecią, wtedy system operacyjny przebiega strukturę katalogową, poszukując na danym urządzeniu programów wykonywalnych i zapisuje niezbędne informacje. Taki mechanizm umożliwia rozpoczęwanie wykonywania programów za pomocą wcześniej opisanego, dwukrotnego naciśnięcia przycisku myszki. Dwukrotne „kliknięcie na pliku” powoduje odczytanie atrybutu jego twórcy i przeszukanie pliku *Desktop File* w celu

dopasowania programu. Jeśli dopasowanie się powiedzie, to odpowiedni program wykonywalny rozpoczyna działanie, a jego źródłem danych staje się plik wskazany za pomocą myszki.

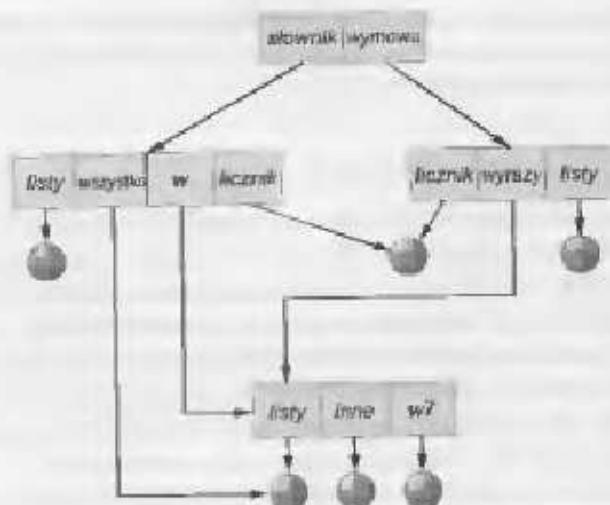
10.3.4 Acykliczne grafy katalogów

Przypuśćmy, że dwu programistów pracuje wspólnie nad pewnym projektem. Pliki związane z tym projektem mogą oni umieszczać w podkatalogu w celu wyodrębnienia ich spośród innych swoich projektów i plików. Ponieważ jednak obaj programiści są w równym stopniu odpowiedzialni za projekt, obaj życzyliby sobie, aby podkatalog z plikami należącymi do projektu znajdował się w ich katalogach. Wspólny podkatalog powinien być dzielony. Dzielony katalog lub plik będzie istniał w systemie w kilku miejscach naraz. Zauważmy, że plik dzielony (lub katalog) nie jest tym samym co dwie kopie pliku. Jeśli istnieją dwie kopie pliku, to każdy z programistów ma dostęp do kopii, a nie do oryginału; gdy jeden programista zmieni coś w pliku, wówczas zmiany te nie wystąpią w drugiej kopii. Przy dzieleniu pliku faktycznie istnieje tylko jeden plik, więc wszelkie zmiany dokonane przez jedną osobę będą natychmiast widoczne dla drugiej. Taki sposób dzielenia jest szczególnie istotny przy dzieleniu katalogów: nowy plik, utworzony przez jedną osobę, pojawi się automatycznie w wszystkich podkatalogach dzielonych.

Struktura drzewiasta nie pozwala na dzielenie plików lub katalogów. Graf acykliczny (ang. *acyclic graph*) umożliwia umieszczenie podkatalogów i plików (rys. 10.10) w katalogach dzielonych. Ten sam plik lub podkatalog może występować w dwóch różnych katalogach. Graf acykliczny (tzn. graf bez cyklu) jest naturalnym uogólnieniem koncepcji katalogu o strukturze drzewa.

Gdy pewna grupa ludzi pracuje zespołowo, wówczas wszystkie pliki, które mają być dzielone, można umieścić razem w jednym podkatalogu. Wszystkie katalogi plików poszczególnych członków zespołu będą zawierały katalog plików dzielonych jako podkatalog. Nawet w przypadku jednego użytkownika organizacja jego plików może wymagać, aby pewne pliki zostały umieszczone w kilku różnych podkatalogach. Na przykład program napisany w ramach jakiegoś projektu powinien znajdować się zarówno w katalogu wszystkich programów, jak i w katalogu danego projektu.

Pliki i podkatalogi dzielone mogą być implementowane na kilku sposobów. Powszechna metoda, którą zastosowano w wielu systemach uniksoowych, polega na utworzeniu nowej pozycji w katalogu, zwanej *dowiązaniem* (ang. *link*). Dowiązanie jest w istocie wskaźnikiem do innego pliku lub podkatalogu. Na przykład jako dowiązania można użyć bezwzględnej lub względnej nazwy ścieżki (dowiązanie symboliczne). Gdy wystąpi odniesienie do pliku, wówczas przeszukuje się katalog. Pozycja w katalogu jest oznaczona



Rys. 10.10 Katalog o strukturze grafu bez cyklu

na jako dowlązanie i zawiera nazwę rzeczywistego pliku (lub katalogu). Następuje przetłumaczenie dowlązania, czyli użycie pamiętanej w nim nazwy ścieżki do zlokalizowania rzeczywistego pliku. Dowlązania można łatwo identyfikować w katalogu na podstawie ich budowy (lub specjalnego typu w systemach operujących typami) – są one faktycznie pośrednimi wskaźnikami nazwowymi. System operacyjny pomija takie dowlązania podczas przeglądania drzew katalogowych, aby zachować strukturę bez cykli.

Inne podejście do realizacji plików dzielonych polega po prostu na podwojeniu wszystkich dotyczących ich informacji w obu dzielących je katalogach. Tak więc oba wpisy katalogowe są identyczne. Dowlązanie wyraźnie różni się od pierwotnego wpisu katalogowego, zatem te dwa elementy nie są jednakowe. Podwojone wpisy katalogowe powodują jednak, że oryginal i kopia są nieroróżnicalne. Głównym problemem przy stosowaniu podwojonych wpisów katalogowych jest utrzymanie spójności w przypadku wykonywania zmian w pliku.

Struktura katalogowa w postaci grafu acyklicznego jest elastyczniejsza niż prosta struktura drzewiasta, lecz równocześnie bardziej złożona. Trzeba starannie rozważyć kilka problemów. Zauważmy, że jednemu plikowi może obecnie odpowiadać wiele bezwzględnych nazw ścieżek. W rezultacie do tego samego pliku mogą odnosić się różne nazwy plików. Jest to sytuacja podobna do zagadnienia synonimów występujących w językach programowania. Jeśli próbujemy przejść cały system plików (w poszukiwaniu pliku, w celu zebrań statystyki ze wszystkich plików lub skopiowania wszystkich plików

w rezerwowej pamięci), to natykamy poważny problem, ponieważ nie chceliśmy obchodzić struktur dzielonych więcej niż jeden raz.

Inny problem powstaje przy usuwaniu plików: kiedy przestrzeń przydzieloną plikowi można mu odebrać i przekazać do wtórnego użytku? Jednym z rozwiązań byłoby usunięcie pliku wtedy, gdy którykolwiek go usuwa, lecz w wyniku takiego działania mogłyby pozostać oswobodzone wskaźniki do już nie istniejącego pliku. Co gorsza, jeśli pozostałe wskaźniki plikowe zawierająby rzeczywiste adresy przestrzeni dyskowej, którą następnie przydzielono by innym plikom, to te bezroku pozostawione wskaźniki mogłyby prowadzić gdzieś do środka innych plików.

Sytuacja ta jest trochę łatwiejsza do opanowania w systemie, w którym dzielenie plików i katalogów zaimplementowano za pomocą dowiezienia symbolicznych. Usunięcie dowiezienia nie narusza samego pliku – usuwa się tylko dowiezanie. Jeśli usuwa się wpis katalogowy pliku, to zwolni się obszar danego pliku, co spowoduje oswoboczenie dowiezienia. Można odnaleźć takie dowiezania i również je usuwać, lecz przeszukiwanie to byłoby bardzo kosztowne, chyba że wykaz dowiezienia byłby przechowywany wraz z każdym plikiem. Można więc pozostawić te dowiezania do czasu, aż pojawi się próba ich użycia. Wówczas można ustalić, że plik o razie określonej przez dowiezanie nie istnieje, wskutek czego użycie dowiezania zakończy się niepowodzeniem i zostanie potraktowane jak każda niedozwolona nazwa pliku. (W tym przypadku projektant systemu powinien starannie rozważyć, co zrobić, gdy po usunięciu pliku zostanie utworzony inny plik z tą samą nazwą, zanim jeszcze dojdzie do odniesienia dowiezania symbolicznego do poprzedniego pliku). W przypadku systemu UNIX dowiezania symboliczne pozostają po usunięciu pliku i uświadomienie sobie, że pierwotny plik już nie istnieje lub został zastąpiony, należy do użytkownika.

Inne podejście do zagadnienia usuwania polega na zachowaniu pliku do czasu, aż zostaną usunięte wszystkie odniesienia do niego. Aby ten pomysł urzeczywiścić, należy mieć mechanizm rozstrzygania o tym, czy zostało usunięte ostatnie odniesienie do pliku. Można prowadzić wykaz odniesień do pliku (wpisów katalogowych lub dowiezzeń symbolicznych). Kiedy powstaje dowiezanie lub kopię wpisu katalogowego, wtedy dodaje się nową pozycję do wykazu odniesień do pliku. Kiedy usuwa się dowiezanie lub wpis katalogowy, wtedy usuwa się też jego pozycję z wykazu. Plik zostaje usunięty wówczas, gdy wykaz odniesień do niego będzie pusty.

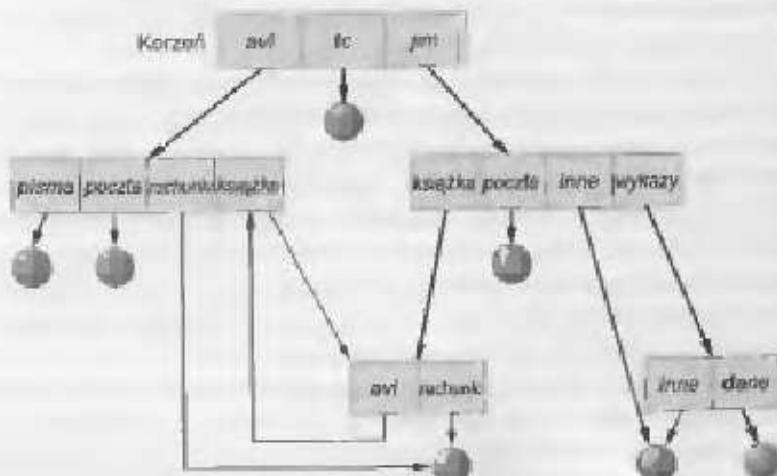
Wadą tego pomysłu jest zmienny i potencjalnie duży rozmiar wykazu odniesień do pliku. Jednakże nie ma istotnego powodu, aby przechowywać taki wykaz w całości – naprawdę jest potrzebna tylko liczba odniesień do pliku. Nowe dowiezanie lub nowa pozycja w katalogu powoduje zwiększenie licznika odniesień do pliku; usunięcie dowiezania lub pozycji z katalogu powo-

duje zmniejszenie tego licznika. Z chwilą wyzerowania licznika plik może zostać usunięty, bo nie ma już do niego żadnych odniesień. Metoda ta znalazła zastosowanie w systemie UNIX w przypadku *dowiązań niesymbolicznych*, czyli *twardych* (ang. *hard links*), w którym licznik odniesień jest prowadzony w bloku informacyjnym pliku (inaczej w i-węźle; zob. p. 21.7.2). Dzięki skutecznemu zakazowi wielokrotnych odniesień do katalogów zachowuje się struktura grafu acyklicznego.

W celu ominięcia tych trudności niektóre systemy nie zezwalają na dzierżenie katalogów lub stosowanie dowiązań. Na przykład w systemie MS-DOS struktura katalogowa ma postać drzewa, a nie grafu acyklicznego, dzięki czemu unika się problemów związanych z usuwaniem plików w strukturze katalogowej o postaci grafu acyklicznego.

10.3.5 Graf ogólny katalogów

Poważnym problemem przy używaniu struktury o postaci grafu bez cykli jest gwarantowanie, że nie powstaną w niej cykle. Jeśli rozpoczniemy od dwupoziomowego katalogu i pozwolimy użytkownikom tworzyć podkatalogi, to powstanie drzewiasta struktura katalogów. Dosyć łatwo zauważyc, że zwykłe dodawanie nowych plików i podkatalogów do istniejącej drzewiastej struktury katalogowej zachowuje jej drzewistą naturę. Jednak dodanie dowiązań do istniejącej drzewiastej struktury katalogowej miszczą ją jako drzewo, przekształcając w ogólną strukturę grafu (rys. 10.11).



Rys. 10.11 Graf ogólny katalogów

Podstawową zaletą grafu acyklicznego jest względna prostota algorytmów jego przechodzenia i sposobu określenia, kiedy do pliku nie ma już żadnych odniesień. Dwukrotnego przechodzenia przez dzielone sekcje grafu acyklicznego unika się głównie w celu optymalizowania działania. Jeśli przeglądnięto właśnie większy podkatalog dzielony i nie znaleziono w nim poszukiwanego pliku, to jest zrozumiałe, że dąży się do uniknięcia jego powtórnego przeglądania – drugie przeszukiwanie byłoby marnowaniem czasu.

Jeśli dopuszcza się występowanie cykli w istniejącym katalogu, to po dobrze jak uprzednio zależy nam na unikaniu przeszukiwania jakiekolwiek składowej dwukrotnie – zarówno ze względu na poprawność, jak i wydajność. Zły określony algorytm może spowodować powstanie niekończącej się pętli nieustannego przeszukiwania. Jedno rozwiązywanie polega na ogólnym ograniczeniu liczby katalogów, z którymi następuje kontakt podczas przeszukiwania.

Podebny problem pojawia się przy próbie określenia, kiedy plik można usunąć. W strukturach katalogowych o postaci grafów bez cykli zerowa wartość licznika odniesień oznacza, że nie ma żadnych odniesień do danego pliku lub katalogu i plik ten można usunąć. Gdy jednak istnieją cykle, to jest również możliwe, że licznik odniesień będzie różny od zera nawet wtedy, gdy nie może już wystąpić odniesienie do katalogu lub pliku. Ta anomalia wynika z możliwości istnienia pętli w strukturze katalogowej. W tym przypadku na ogół niecodzienne staje się zastosowanie schematu *łączenia nieużytków* (ang. *garbage collection*) w celu określenia, kiedy usunięto ostatnie odniesienie i umożliwienia ponownego przydziału przestrzeni dyskowej. Łączenie nieużytków wymaga obходu systemu plików i oznaczenia wszystkiego, do czego można dostrzec. Następnie w drugim przebiegu wszystkie nie zaznaczone obszary zbiera się na wykazie wolnych przestrzeni. (Podobną procedurą zaznaczania można się posłużyć w celu zagwarantowania, że obchód lub przeszukanie obejmie wszystkie pliki w systemie dokładnie jeden raz). Jednak łączenie nieużytków w dyskowym systemie plików jest skrajnie czasochłonne i z tego powodu rzadko podejmowane.

Łączenie nieużytków jest konieczne tylko z powodu możliwości wystąpienia cykli w grafie. Znacznie łatwiej jest zatem pracować z grafem o strukturze bez cykli. Trudnością jest unikanie cykli wówczas, gdy do struktury dodaje się nowe dowiązania. Jak poznac, czy nowe dowiązanie nie dopełni cyklu? Istnieją algorytmy wykrywania cykli w grafach, jednak wymagają one dużych nakładów obliczeniowych, zwłaszcza gdy graf znajduje się w pamięci dyskowej. Ogólnie rzecz biorąc, można powiedzieć, że drzewiaste struktury katalogowe są powszechniej stosowane niż struktury grafów bez cykli.

10.4 ■ Ochrona

Do podstawowych zagadnień związanych z przechowywaniem informacji w systemie komputerowym należy zarówno chronienie jej reprezentacji przed fizycznym uszkodzeniem (niezawodność), jak i przed niewłaściwym dostępem (ochrona).

Niezawodność (ang. *reliability*) z zasady osiąga się dzięki wykonywaniu zapasowych kopii plików. Wciąż komputerów ma programy systemowe, które automatycznie (lub wskutek interwencji operatora) kopią pliki dyskowe na taśmę w regularnych odstępach czasu (raz na dzień, na tydzień lub na miesiąc), aby na wypadek uszkodzenia systemu istniały kopie plików. Systemy plików mogą ulegać uszkodzeniom z przyczyn sprzętowych (takich jak błędy przy czytaniu lub pisaniu), z powodu wahań napięcia w sieci zasilającej lub jej awarii, uszkodzenia głowic, brudu, zbytnich wahań temperatury oraz vandalizmu. Pliki mogą zostać usunięte przypadkowo. Błędy oprogramowania systemu plików mogą również doprowadzać do utraty zawartości plików.

Ochronę (ang. *protection*) można zapewnić różnymi sposobami. W małym, jednostanowiskowym systemie można w tym celu wyjąć dyskietki z komputera i zamknąć je w szufladzie biurka lub szafce na dokumenty. Jednakże w systemie dla wielu użytkowników trzeba zastosować inne środki ochrony.

10.4.1 Rodzaje dostępu

Potrzeba ochrony plików jest bezpośrednią konsekwencją możliwości dostępu do plików. W systemach, które nie pozwalały na dostęp do plików innych użytkowników, ochrona nie jest potrzebna. Tak więc jedną skrajnością byłoby wprowadzenie pełnej ochrony przez zakazanie dostępu do plików. Drugą skrajnością jest pozostawienie swobody dostępu, bez żadnej ochrony. Oba podejścia są zbyt skrajne, aby można je było stosować powszechnie. Potrzebny jest *dostęp kontrolowany*.

Mechanizmy ochrony urzeczywistniają dostęp kontrolowany przez ograniczenie możliwych rodzajów dostępu do pliku. Dostęp jest dozwolony lub zabroniony zależnie od grupy czynników, z których jednym jest rodzaj zgłoszonego dostępu. Kontrola może podlegać kilku różnych typów operacji:

- **Czytanie:** czytanie z pliku.
- **Pisanie:** pisanie do pliku lub zapisywanie go na nowo.
- **Wykonywanie:** załadowanie pliku do pamięci i wykonanie go.

- **Dopisywanie:** zapisywanie danych na końcu pliku.
- **Usuwanie:** usuwanie pliku i zwalnianie jego obszaru do ewentualnego wtórnego użycia.
- **Opisywanie:** wyprowadzenie nazwy i atrybutów pliku.

Inne operacje, takie jak przemianowywanie, kopowanie lub redagowanie pliku, również mogą podlegać kontroli. Jednak w wielu systemach te funkcje wyższego poziomu (jak kopowanie) mogą być wykonywane za pomocą programów systemowych, które korzystają z wywołań systemowych niższego poziomu. Ochrona występuje tylko na tym niższym poziomie. Na przykład kopowanie pliku można zaimplementować po prostu jako ciąg zamówień na operację czytania. W tym przypadku użytkownik, który ma prawo do czytania pliku, będzie mógł także spowodować jego kopowanie, drukowanie itd.

Zaproponowano wiele różnych mechanizmów ochrony. Każda metoda ma swoje zalety i wady – należy więc dobierać ją odpowiednio do zamierzzonego zastosowania. Mały system komputerowy, z którego korzysta tylko kilku członków grupy badawczej, może nie potrzebować takiej samej ochrony, jakiej wymaga wielki komputer korporacji używany do wspomagania badań, rozliczeń finansowych i operacji personalnych. Pełne omówienie problemów ochrony odkładamy do rozdz. 19.

10.4.2 Wykazy i grupy dostępów

Najpopularniejszym podejściem do problemu ochrony jest uzależnienie dostępu od identyfikacji użytkownika. Różni użytkownicy mogą potrzebować różnych rodzajów dostępu do plików lub katalogów. Najogólniejsza metoda implementacji dostępu zależnego od tożsamości polega na skojarzeniu z każdym plikiem i katalogiem *wykazu dostępów* (ang. *access list*)^{*} zawierającego nazwy użytkowników i dozwolenie dla poszczególnych użytkowników rodzaje dostępu. Kiedy użytkownik żąda dostępu do jakiegoś pliku, wtedy system operacyjny przeszuka wykaz dostępów przyporządkowany danemu plikowi. Jeśli użytkownik widnieje na nim jako uprawniony do dostępu danego rodzaju, to uzyska ten dostęp. W przeciwnym razie zostanie wykryta próba naruszenia reguł ochrony i zadanie użytkownika otrzyma odmowę dostępu do danego pliku.

Główna wadą wykazów dostępów jest ich długość. Gdybyśmy chcieli wszystkim pozwolić czytać plik, musielibyśmy sporządzić spis wszystkich użytkowników uprawnionych do jego czytania. Ma to dwie niepożądane konsekwencje:

^{*} W użyciu jest również termin *listy kontroli dostępu* (ang. *access control list*). — Przyp. tłum.

- Sporządzanie takiego wykazu może być żmudnym i nie przynoszącym korzyści zajęciem, zwłaszcza jeśli nie znamy z góry spisu użytkowników systemu.
- Wpis katalogowy, który dotąd był stałego rozmiaru, musiałby obecnie mieć zmienną długość, a to komplikuje zarządzanie przestrzenią dyskową.

Można unikać tych kłopotów dzięki zagościowej wersji wykazu dostępów.

Aby skrócić wykaz dostępów, w wielu systemach do każdego z plików odnosi się trzy klasy użytkowników:

- **Właściciel:** Właścicielem jest użytkownik, który utworzył dany plik.
- **Grupa albo zespół roboczy:** Tak określa się zbiór użytkowników, którzy wspólnie korzystają z pliku i potrzebują do niego podobnego dostępu.
- **Wszechświat:** Wszyscy inni użytkownicy stanowią wszechświat⁷.

Przypuszcmy na przykład, że pewna osoba, Aja, pisze nową książkę. Do pomocy w tym przedsięwzięciu wynajęła ona troje starszych studentów – Janka, Jerzego i Basię. Treść książki znajduje się w pliku o nazwie *książka*. Ochrona tego pliku wygląda następująco:

- Aja powinna móc wykonać każdą operację na pliku.
- Janek, Jerzy i Basia powinni móc tylko czytać, pisać i wykonywać ten plik; nie powinno się im pozwolić na usuwanie pliku.
- Innym użytkownikom powinno się umocliwić czytanie pliku. (Aja jest zainteresowana tym, aby jak najwięcej osób zapoznawało się z jej tekstem, ponieważ jest ciekawa ich opinii).

W celu uzyskania takiej ochrony należy utworzyć nową grupę, powiedzmy *tekst*, do której wejdą Janek, Jerzy i Basia. Nazwę grupy *tekst* należy następnie skojarzyć z plikiem *książka*, a prawa dostępu określić zgodnie z powyższymi zasadami.

Zauważmy, że do poprawnego działania tego schematu członkostwo grupy musi podlegać ścisłej kontroli. Kontrolę tę można zapewnić różnymi sposobami. Na przykład w systemie UNIX tworzeniem grup i ich aktualnianiem zajmuje się tylko zarządcą tej usługi systemowej (lub dowolny użytkownik o zwiększonej przywilejach). Zatem kontrola ta jest sprawowana na zasadzie współpracy systemu z człowiekiem. W systemie VMS z każdym plikiem

⁷ A raczej jego „reszta” – Przyp. tłum.

można skojarzyć wykaz dostępów (znany również pod nazwą listy kontroli dostępów) zawierający użytkowników mających prawa dostępu do pliku. Właściciel pliku może taką listę utworzyć i zmieniać. Dalsze omówienie wykazów dostępów znajduje się w p. 19.4.2.

Przy tak ograniczonej klasyfikacji do zdefiniowania ochrony wystarczą zaledwie trzy pola. Każde pole jest najczęściej zbiorem bitów, z których każdy albo pozwala, albo zabrania określonego rodzaju dostępu. Na przykład w systemie UNIX zdefiniowano trzy pola 3-bitowe: *rwx*, przy czym bit *r* kontroluje prawo czytania, bit *w* – prawo pisania, a bit *x* – prawo dostępu umożliwiającego wykonanie. Przechowuje się osobne pola dla właściciela pliku, dla podlegającej właściwości grupy i dla reszty użytkowników. W takim schemacie na zapisanie informacji o ochronie pliku książka są następujące: właścicielka, Aja, ma wszystkie trzy bity określone na 1, dla grupy *tekst* ustawione na 1 są bity *r* i *w*, a wszechnisław ma ustawiony na 1 tylko bit *r*.

Zauważmy jednak, że schemat ten nie jest tak ogólny jak schemat z wykazem dostępów. Aby to zilustrować, powróćmy do naszego przykładu z książką. Założymy, że Aja chce wykluczyć Jędrka ze spisu osób uprawnionych do czytania tekstu. Za pomocą naszkicowanego, podstawowego schematu postępowania nie uda się jej tego zrobić.

10.4.3 Inne metody ochrony

Inny sposób potraktowania ochrony polega na przypisaniu każdemu plikowi hasła. Na podobieństwo dostępu do systemu komputerowego, który jest często kontrolowany za pomocą hasła, dostęp do każdego pliku można nadzorować przy użyciu hasła. Jeżeli hasła są wybierane losowo i często zmieniane, to postępowanie takie może skutecznie ograniczać dostęp do pliku do grona tylko tych użytkowników, którzy znają hasło. Schemat taki ma jednak kilka wad. Po pierwsze, jeśli z każdym plikiem będziemy kojarzyć osobne hasło, to liczba hasł, które przyjdzie pamiętać użytkownikowi, może stać się ogromna, czyniąc tę metodę niepraktyczną. Jeżeli do wszystkich plików zastosuje się to samo hasło, to po jego odkryciu, wszystkie pliki staną się dostępne. Aby usunąć tę niedogodność, w niektórych systemach (np. w systemie TOPS-20) pozwala się użytkownikowi przypisywać hasła do katalogów, a nie do poszczególnych plików. W systemie operacyjnym IBM VM/CMS do jednego minidysku można odnosić trzy hasła: do czytania, do pisania i do zapisywania wielokrotnego. Po drugie, na ogół każdy plik ma tylko jedno hasło. Ochrona odbywa się więc w myśl zasad "wszystko albo nic". W celu uzyskania ochrony bardziej szczegółowej należy zastosować wiele hasł.

Ograniczona ochrona plików jest także możliwa obecnie w systemach operacyjnych dla jednego użytkownika, takich jak MS-DOS i Macintosh. Pierwotnie w tych systemach nie zakładano żadnej konieczności występowania ochrony. Jednak z uwagi na to, że systemów tych zaczęto używać w sieciach, w których występuje konieczność wspólnego korzystania z plików oraz potrzeba komunikacji, więc również i te systemy operacyjne są wtórnie zapatrzywane w mechanizmy ochrony. Zauważmy, że prawie zawsze łatwiej jest zaprojektować pewną własność w nowym systemie operacyjnym, niż dodawać ją do systemu istniejącego. Uaktualnienia takie są zazwyczaj mniej skuteczne i nie daje się ich wprowadzić w sposób niewidoczny.

Zwróciliśmy już uwagę na to, że w wielopoziomowej strukturze katalogów należy chronić nie tylko poszczególne pliki, lecz także ich zbiory zawarte w podkatalogach. Potrzebujemy zatem mechanizmów ochrony katalogów. Operacje katalogowe, które wymagają ochrony, różnią się nieco od operacji plikowych. Należy nadzorować tworzenie i usuwanie plików w katalogach. Być może będą potrzebne również środki kontrolowania, czy użytkownikowi jest wolno dowiedzieć się o występowaniu pliku w katalogu. Samą znajomość istnienia pliku i jego nazwy może być niekiedy istotna, dlatego wypowiadanie zawartości katalogu powinno być operacją chronioną. Jeśli zatem ścieżka odnosi się do pliku w katalogu, to użytkownik powinien mieć dostęp zarówno do katalogu, jak i do pliku. W systemach, w których pliki mogą mieć po kilka nazw ścieżek (np. w grafach acyklicznych lub ogólnych), prawa dostępu do pliku mogą się różnić między sobą w zależności od użytcej przez użytkownika nazwy ścieżki.

10.4.4 Przykład – system UNIX

W systemie UNIX ochrona katalogu jest podobna do ochrony pliku. Z każdym podkatalogiem są skojarzone trzy pola: właściciela, grupy i świata, przy czym każde z nich zawiera 3 bity: rwx. Użytkownik może wypowadać za-

-rw-rw-r-	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx—	5 pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwxr-	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r-r-	1 pbg	staff	9423	Feb 24 1993	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 1993	program
drwx-ox-ox	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx—	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrws	3 pbg	staff	512	Jul 8 09:35	test/

Rys. 10.12 Przykładowa zawartość katalogu

wartość katalogu tylko wtedy, gdy w odpowiednim polu bit *r* ma wartość 1. Na podobnej zasadzie zmiana bieżącego katalogu na inny (powiedzmy – *takim*) jest dozwolona tylko wtedy, kiedy w odpowiednim polu związanym z katalogiem *takim* jest ustawiony bit *x*.

Przykład wykazu pozycji katalogowych, zaczerpnięty z systemu UNIX, widać na rys. 10.12. Pierwsze pole opisuje sposób ochrony pliku lub katalogu. Litera *d* występująca w pierwszej kolumnie symbolizuje katalog. W każdym wierszu wykazu ujawnia się również liczbę dowiązań do pliku, nazwę jego właściciela, nazwę grupy, rozmiar pliku w bajtach, datę utworzenia pliku oraz nazwę pliku (z ewentualnym rozszerzeniem).

10.5 ■ Semantyka spójności

Semantyka spójności (ang. *consistency semantics*) jest ważnym kryterium oczekiwany dowolnego systemu plików realizującego dzielenie plików. Jest to właściwość systemu określająca semantykę jednoczesnego dostępu do pliku dla wielu użytkowników. Semantyka ta powinna zwłaszcza określać warunki, przy których zmiany danych wykonywane przez jednego użytkownika są obserwowlane przez innych użytkowników.

W naszych rozważaniach będziemy zakałać, że ciąg dostępów do pliku (tj. operacji czytania i pisania), zgłaszanych przez użytkownika do tego samego pliku, jest zawsze zowany między operacjami otwierania i zamknięcia pliku. Taki ciąg dostępów nazywamy *sesją plikową* (ang. *file session*). Aby zilustrować to pojęcie, naszkicujemy kilka przejrzystych przykładów semantyki spójności.

10.5.1 Semantyka spójności systemu UNIX

W systemie plików UNIX (zob. rozdz. 17) zastosowano następującą semantykę spójności:

- Wynik operacji pisania wykonanej przez użytkownika na otwartym pliku jest natychmiast widoczny dla innych użytkowników, którzy mają równocześnie otwarty ten sam plik.
- Istnieje tryb dzielenia, w którym użytkownicy wspólnie korzystają ze wskaźnika bieżącego położenia w pliku. Zatem przesunięcie wskaźnika przez jednego użytkownika oddziałuje na wszystkich użytkowników dzielących plik. Plik ma jeden obraz, do którego odnoszą się wszystkie dostępy, niezależnie od ich pochodzenia.

Tego rodzaju семантику można zastosować w implementacji приватної відносини, яка пов'язує кожному файлу один фізичний зображення, до якого доступ відбувається на основі виключності застосування. Відповідність до цього об'єкту об'єкт поводить до опізнання процесів користувачів.

10.5.2 Семантика сесії

У системі пілків Andrew (зуб. розд. 17) зastosовано наступуючу семантику спільноти:

- Результат операції запису, виконаної користувачем на відкритому пілку, не є видимий для інших користувачів, які мають одночасно відкрити цей сам пілк.
- Зміни, впроваджені до пілку, будуть видимі після його закриття доділо в наступних сесіях. Відкрите, більше екземпляри даного пілку не будуть відображати цих змін.

Згідно з цією семантикою пілк може бути в тому ж самому часі окресово скожжений з кількома (або більшими) зображеннями. У наслідку цього користувачам вільно без опізнання виконувати спільноти на їх зображеннях пілку одночасно операції читання та запису. Зauważмо, що на планування доступу до пілку не наложено практично будь-яких обмежень.

10.5.3 Семантика статичних пілків, поділених між

З грунту однієї позиції є застосування статичних пілків, поділених (англ. *immutable shared files*). У момент, коли створюється пілк, він відмінюється від змін. Пілк статичний має дві важливі властивості: його назва не може бути змінена, і його зміст не може бути змінений. Затем назва пілку сталою означає його незмінну зміст, а не поjemnik на різні інформації. Реалізація цієї семантики в системі розподіленому (розд. 17) є проста, оскільки поділ пілку на зображення вимагає додаткової контролі (допускається тільки читання).

10.6 ■ Подсумування

Пілк є абстрактним типом даних, визначеного та реалізованого системою операційною. Він є послідовністю рекордів. Рекордом логічним може бути рядок, рядок (з постійною або змінною довжиною) або більші згрупованими одиницями даних. В операційній системі можуть існувати специфічні методи

tworzenia rekordów różnych typów lub może to należeć do zadań programów użytkowych.

Głównym obowiązkiem systemu operacyjnego jest odwzorowywanie logicznej koncepcji pliku na fizyczne urządzenia pamięci, takie jak taśma magnetyczna lub dysk. Ponieważ fizyczny rozmiar rekordu nie musi być równy rozmiarowi rekordu logicznego, może się okazać konieczne zestawianie rekordów logicznych w rekordy fizyczne. I w tym przypadku zadanie to może być wykonywane przez system operacyjny lub pozostawione dla programu użytkowego.

Systemy plików dla taśm magnetycznych mają ograniczone możliwości; w większości systemów plików stosuje się dyski. Taśmy są powszechnie używane do przenoszenia danych między maszynami, do tworzenia kopii zapasowych lub do celów archiwalnych.

Każde urządzenie w systemie plików przechowuje spis treści reprezentowanego przez nie tomu informacji lub katalog ze spisem lokalizacji przechowywanych w nim plików. Tworzenie katalogów umożliwia ponadto organizowanie plików. Katalog jednopoziomowy w systemie z wieloma użytkownikami jest klepotliwy, ponieważ każdy plik musi mieć niepowtarzalną nazwę. Trudności tych umika się, stosując katalogi dwupoziomowe, w których można utworzyć oddzielnny katalog dla każdego użytkownika. Każdy użytkownik otrzymuje własny katalog zawierający jego pliki.

Pliki w katalogach są wykazywane za pomocą nazw, ponadto katalogi zawierają takie informacje, jak położenie pliku na dysku, jego długość, typ, przynależność do właściciela, czas utworzenia, czas ostatniego użycia itp.

Naturalnym uogólnieniem katalogu dwupoziomowego jest drzewista struktura katalogowa. Drzewista struktura katalogowa pozwala użytkownikowi na tworzenie podkatalogów w celu organizowania jego plików. Struktury katalogowe o postaci grafów acyklicznych pozwalały dzielić katalogi i pliki, lecz komplikują ich przeszukiwanie i usuwanie. Graf ogólny umożliwia całkowitą elastyczność w dzieleniu plików i katalogów, lecz wymaga niekiedy odświeżania w celu odzyskiwania nieużywanych obszarów dyskowych.

Ponieważ w większości systemów komputerowych pliki są głównym środkiem przechowywania informacji, jest więc potrzebna ich ochrona. Kontroli może podlegać oddzielnie każdy rodzaj dostępu do pliku: czytanie, pisanie, wykonywanie, dopisywanie, wyprowadzanie katalogu itd. Do ochrony plików można posługiwać się hasłami, wykazami dostępów lub specjalnymi technikami ohmyślanymi *ad hoc*.

Systemy plików są często implementowane za pomocą struktur warstwowych lub modułarnych. Niższe poziomy zajmują się obsługą fizycznych urządzeń pamięci. Wyższe poziomy działają na symbolicznych nazwach plików i dotyczą logicznych właściwości plików. Poziomy pośrednie odwzorowują logiczne koncepcje pliku na właściwości urządzeń fizycznych.

■ Ćwiczenia

- 10.1** Rozważmy system plików, w którym plik można usunąć i zwolnić zajmowany przez niego obszar dyskowy nawet wtedy, gdy istnieją jeszcze jakieś dowiązania do tego pliku. Jakie mogą pojawić się kłopoty, jeśli przy tworzeniu nowego pliku zostanie wybrane to samo miejsce pamięci lub taka sama bezwzględna nazwa ścieżki? Jak można ich uniknąć?
- 10.2** Niektóre systemy automatycznie usuwają wszystkie pliki użytkownika, gdy wyrejestruje się on z systemu lub skończy się jego zadanie, chyba że użytkownik jawnie poprosi o zachowanie plików. Inne systemy dopóty przechowują pliki, dopóki użytkownik jawnie ich nie usunie. Przedyskutuj względne zalety każdego z tych podejść.
- 10.3** Dlaczego w niektórych systemach przechowuje się informacje o typach plików, podczas gdy w innych pozostawia się to użytkownikowi lub po prostu nie implementuje się wielu typów plików? Które z systemów są „lepsze”?
- 10.4** Analogicznie, w niektórych systemach realizuje się wiele typów struktur danych plikowych, a w innych operuje się po prostu strumieniami bajtów. Jakie są tego zalety i wady?
- 10.5** Jakie są zalety i wady zapamiętywania nazwy programu tworzącego plik jako jednego z atrybutów pliku (dziedzicząc tak np. w systemie operacyjnym Macintosh)?
- 10.6** Czy potrafisz zasymulować wielopoziomową strukturę katalogową za pomocą jednopoziomowej struktury katalogowej, w której można używać dowolnie długich nazw? Jeśli odpowiesz twierdząco, wyjaśnij, w jaki sposób można to osiągnąć i porównaj ten schemat ze schematem katalogu wielopoziomowego. Jeśli odpowiesz przecząco, wyjaśnij, co uniemożliwia powodzenie takiej symulacji. Jak zmieniłaby się Twoja odpowiedź, gdyby nazwy plików były ograniczone do 7 znaków?
- 10.7** Wyjaśnij przeznaczenie operacji otwierania i zamknięcia pliku.
- 10.8** Niektóre systemy automatycznie otwierają plik przy pierwszym do niego odwołaniu i zamkniętym go po zakończeniu zadania. Przeanalizuj zalety i wady takiej metody w porównaniu z bardziej tradycyjną, w której użytkownik musi jawnie otwierać i zamkniąć pliki.
- 10.9** Podaj przykład zastosowania, w którym dostęp do danych w pliku powinien być:
- (a) sekwencyjny,
 - (b) losowy.

- 10.10** Niektóre systemy umożliwiają dzielenie pliku za pośrednictwem pojedynczej kopii danego pliku; inne w tym celu tworzą kilka kopii – po jednej dla każdego użytkownika pliku dzielonego. Oceń względne zalety obu metod.
- 10.11** W pewnych systemach użytkownik uprawniony może czytać i zapisywać podkatalogi tak jak zwykłe pliki.
- (a) Opisz, jakie to może rodzić problemy związane z ochroną.
 - (b) Zaproponuj schemat postępowania w przypadku każdego z wymienionych przez Ciebie problemów.
- 10.12** Rozważ system obsługujący 5000 użytkowników. Założmy, że chcesz zezwolić na to, aby 4990 spośród tych użytkowników miało dostęp do jakiegoś pliku.
- (a) Jak można określić ten rodzaj ochrony, przyjmując metodę stosowaną w systemie UNIX?
 - (b) Czy potrafisz zaproponować inny schemat ochrony, który można byłoby w tym celu zastosować skuteczniej niż to ma miejsce w systemie UNIX?
- 10.13** Specjalisci zaproponowali, aby – zamiast przyporządkowywać każdemu plikowi wykaz dostępów (którzy użytkownicy mają prawo dostępu do pliku i w jaki sposób), tworzyć dla każdego użytkownika listę jego dozgórów (ang. *user control list*) określającą, do których plików użytkownik ma prawo dostępu i w jaki sposób). Omów względne zalety obu tych schematów.

Uwagi bibliograficzne

Ogólne omówienie systemu plików zawiera praca Grosshansa [157]. Golden i Pechura [151] opisali struktury systemów plików dla mikrokomputerów. Pełne omówienie systemów baz danych oraz ich budowy zawiera książka Silberschatza i in. [389].

Strukturę katalogu wielopoziomowego po raz pierwszy zaimplementowano w systemie MULTICS [317]. W większości obecnych systemów operacyjnych są realizowane wielopoziomowe struktury katalogowe. Należy do nich system UNIX [354], system operacyjny Apple Macintosh [14] oraz system MS-DOS [290].

System plików MS-DOS opisali Norton i Wilton w książce [312]. System plików wbudowany w system operacyjny VAX VMS omówili Kenah i in.

[205]; jego opis zawiera również podręcznik [109]. Sieciowy system plików NFS, opracowany w firmie Sun Microsystems, pozwala obejmować strukturami katalogów sieciowe systemy komputerowe. System NFS omówiono w pracach Sandberga i in. [369], [368] oraz w podręczniku [410]. System NFS opisano też szczegółowo w rozdz. 17 niniejszej książki. Semantykę stałych plików dzielonych omówili Schroeder i in. [381].

Trwają interesujące badania w dziedzinie interfejsów systemów plików. W kilku artykułach z konferencji USENIX [433] omówiono zagadnienia nazewnictwa plików i ich atrybutów. Na przykład wszystkie obiekty systemu operacyjnego Plan 9, rodem z Bell Laboratories (Lucent Technology), wyglądają jak systemy plików. Aby więc wyprowadzić spis procesów w systemie, użytkownik po prostu ogląda zawartość katalogu */proc*. Podobnie, żeby wyświetlić porę dnia, wystarczy wydrukować plik */dev/time*.

IMPLEMENTACJA SYSTEMU PLIKÓW

Zgodnie z tym, co powiedzieliśmy w rozdz. 10, system plików dostarcza mechanizmów bezpośredniego przechowywania i dostępu zarówno do danych, jak i programów. System plików rezyduje na stałe w *pamięci pomocniczej*, której podstawowym zadaniem jest trwałe przechowywanie dużej ilości danych. W tym rozdziale zajmujemy się głównie zagadnieniami dotyczącymi magazynowania plików i dostępu do nich na najpopularniejszym nośniku pamięci pomocniczej – dysku. Analizujemy sposoby przydzielania miejsca na dysku, odzyskiwania zwolnionych obszarów pamięci dyskowej, doglądania położenia danych oraz środki komunikacji innych części systemu operacyjnego z pamięcią pomocniczą. Rozważamy też problemy dotyczące wydajności.

11.1 ■ Budowa systemu plików

Dyski stanowią podstawową część pamięci pomocniczej, w której jest utrzymywany system plików. W celu poprawienia wydajności wejścia-wyjścia przesyłanie informacji między pamięcią operacyjną a dyskiem odbywa się w jednostkach nazywanych *blokami* (ang. *blocks*). Każdy blok zawiera jeden lub więcej sektorów. W zależności od rodzaju dysku długość sektorów waha się w granicach od 32 do 4096 B; zwykle jest to 512 B. Dyski charakteryzuje się dwiema ważnymi cechami, które czynią z nich wygodny środek przechowywania wielu plików:

1. Informacje mogą być aktualniane bez zmiany miejsca; można przeczytać blok z dysku, wprowadzić do niego zmiany i zapisać go z powrotem na tym samym miejscu na dysku.

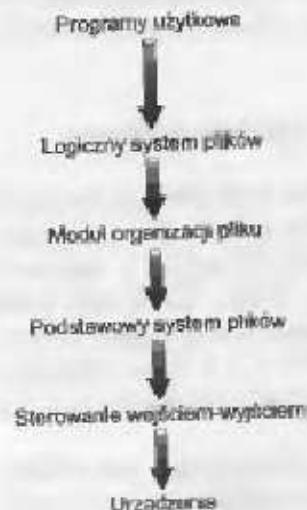
2. Dowolny blok informacji na dysku można zaadresować bezpośrednio. Zatem zarówno sekwencyjny, jak i swobodny dostęp do dowolnego pliku jest łatwy do osiągnięcia, a przełączanie od jednego pliku do drugiego wymaga tylko przesunięcia głowic czytająco-zapisujących i zaczekania na obrót dysku.

Budowę dysku omawiamy szczegółowo w rozdz. 13.

11.1.1 Organizacja systemu plików

W celu umożliwienia wydajnego i wygodnego dostępu do dysku system operacyjny wprowadza pewien *system plików* (ang. *file system*), umożliwiający łatwe zapamiętywanie, odnajdywanie i odzyskiwanie danych. Z systemem plików kojarzą się dwa zupełnie odmienne zagadnienia projektowe. Pierwsze z nich dotyczy sposobu, w jaki system plików jawi się przed użytkownikiem. Obejmuje ono definicję pliku oraz jego atrybutów, operacje dozwolone na pliku oraz strukturę katalogową do organizowania plików. Następnie należy utworzyć algorytmy i struktury danych służące do odwzorowywania logicznego systemu plików na fizyczne urządzenia pamięci pomocniczej.

Sam system plików składa się na ogół z wielu różnych poziomów. Struktura widoczna na rys. 11.1 jest przykładem projektu warstwowego. Każdy poziom korzysta z właściwości niższych poziomów do tworzenia nowych właściwości, używanych przez poziomy wyższe.



Rys. 11.1 Warstwowy system plików

Poziom najwyższy – sterowanie wejściem-wyjściem (ang. *I/O control*) – składa się z modułów obsługi urządzeń i procedur obsługi przerwań związanych z przesyaniem informacji między pamięcią operacyjną a systemem dyskowym. *Moduł obsługi urządzenia** (ang. *device driver*) można uważać za translator. Jego wejście zawiera polecenia wysokiego poziomu, w rodzaju: „pobierz blok 123”. Na jego wyjściu występują zalecne od sprzętu, niskopoziomowe rozkazy używane przez sterownik sprzętowy, który łączy urządzenie wejścia-wyjścia z resztą systemu. Moduł obsługi urządzenia zazwyczaj wpisuje określony układ bitów do specjalnych komórek pamięci sterownika wejścia-wyjścia, aby go poinformować, w którym miejscu urządzenia należy podjąć działania i jakie. Szczegóły dotyczące modułów obsługi urządzeń oraz infrastruktury wejścia-wyjścia są podane w rozdz. 12.

Podstawowy system plików (ang. *basic file system*) wydaje tylko ogólne instrukcje odpowiedniemu modułowi obsługi urządzenia w celu czytania i pisania poszczególnych bloków na dysku. Każdy fizyczny blok dysku jest identyfikowany za pomocą jego adresu liczbowego (np. napęd 1, cylinder 73, powierzchnia 2, sektor 10).

Moduł organizacji pliku (ang. *file-organization module*) interpretuje zarówno pliki i ich bloki logiczne, jak i bloki fizyczne. Znając zastosowany rodzaj przydziału miejsca dla pliku i położenie pliku, moduł organizacji pliku może tłumaczyć adresy logiczne bloków na adresy bloków fizycznych do przesłania przez podstawowy system plików. Bloki logiczne każdego pliku są ponumerowane od 0 (lub 1) do N . Numery bloków fizycznych z danymi na ogół nie odpowiadają numerom logicznym, dlatego podczas tłumaczenia należy lokalizować każdy blok. Moduł organizacji pliku zawiera również zarządcę wolnych obszarów, który odnotowuje nie przydzielone bloki i udogodnia je modułowi organizacji pliku na życzenie.

Na koniec *logiczny system plików* (ang. *logical file system*) używa struktury katalogowej, aby na podstawie symbolicznej nazwy pliku dostarczać informacji potrzebnych modułowi organizacji pliku. Logiczny system plików odpowiada także za ochronę i bezpieczeństwo (zob. rozdz. 10 i rozdz. 19).

W celu utworzenia nowego pliku program użytkowy wywołuje logiczny system plików. Logiczny system plików zna format struktur katalogowych. Aby utworzyć nowy plik, system czyta odpowiedni katalog do pamięci, uaktualnia go, dodając nowy wpis, i z powrotem zapisuje na dysku. Niektóre systemy operacyjne, w tym system UNIX, traktują katalog dokładnie tak samo jak plik, z tym że w polu typu zaznacza się, że jest to katalog. Inne systemy operacyjne, jak Windows NT, realizują osobne wywołania systemowe dla plików i katalogów i traktują katalogi jako jednostki odrębne od plików. Jeśli katalog traktuje

* Inaczej: moduł sterujący urządzeniem – Przyp. tłum.

się jako plik specjalny, to logiczny system plików może wywoływać moduł organizacji pliku w celu odwzorowywania katalogowych operacji wejścia-wyjścia na numery bloków dyskowych, które są przekazywane do podstawowego systemu plików i systemu sterowania wejściem-wyjściem.

Po utworzeniu pliku można go używać do operacji wejścia-wyjścia. Przed każdą operacją wejścia-wyjścia można by odnajdywać plik w strukturze katalogowej, sprawdzać jego parametry, lokalizować jego bloki danych, aby na koniec wykonać na tych blokach operację. Każda operacja wymagałaby dużego zatracenia. Zamiast tego plik przed udostępnieniem go procedurom wejścia-wyjścia należy otworzyć. Podczas otwierania pliku następuje odnalezienie odpowiedniego wpisu pliku w strukturze katalogowej. Fragmenty struktury katalogowej są zazwyczaj przechowywane podręcznie w pamięci operacyjnej, aby przyspieszyć działania na katalogach. Po znalezieniu pliku związane z nim informacje, takie jak rozmiar, właściciel, prawa dostępu i umiejscowienie bloków danych są z reguły kopowane do tablicy nazywanej *tablicą otwartych plików*, znajdującej się w pamięci operacyjnej, i zawierającej informacje o wszystkich bieżąco otwartych plikach (rys. 11.2).

Pierwsze odwołanie do pliku (zazwyczaj operacja jego otwarcia – *open*) powoduje przeszukanie katalogu i przekopiowanie wpisu katalogowego dotyczącego danego pliku do tablicy otwartych plików. Do programu użytkownika zostaje przekazany indeks do tej tablicy, wszystkie dalsze odwołania są więc wykonywane za pomocą tego indeksu, a nie nowej symbolicznej. Indeks ten bywa rozmaitie określany. W systemach uniksowych nazywa się go *deskryptorem pliku* (ang. *file descriptor*), w systemie Windows NT nosi on nazwę *uchwytu plikowego* (ang. *file handle*), a w innych systemach jest nazywany *blokiem kontrolnym pliku* (ang. *file control block*). Tak więc dopóki nie nastąpi zamknięcie pliku, dopóty wszystkie operacje plikowe są wykonywane z użyciem tablicy otwartych plików. Po zamknięciu pliku przez wszystkich korzystających z niego użytkowników aktualne informacje o nim są z powrotem kopowane do dyskowej struktury katalogowej.

indeks	nazwa pliku	prawa	daty dostępu	wskaznik do bloku dyskowego
0	TEST.C	rw rw rw	...	→
1	MAIL.TXT	rw		→
2				
...				
n				

Rys. 11.2 Typowa tabela plików otwartych

W niektórych systemach schemat ten komplikuje się jeszcze bardziej wskutek zastosowania w pamięci operacyjnej tablic wielopoziomowych. Na przykład w systemie plików UNIX BSD każdy proces ma tablicę otwartych plików, która zawiera wykaz wskaźników indeksowanych za pomocą de-skryptorów. Wskaźniki prowadzą do ogólnosystemowej tablicy otwartych plików. W tej tablicy są zawarte informacje o otwartych obiektach. W przypadku plików jest wskazywana *tablica aktywnych i-węzłów* (ang. *inodes*). W przypadku innych obiektów, takich jak przyłącza sieciowe lub urządzenia, są wskazywane podobne informacje służące do ich udostępniania. Tablica aktywnych i-węzłów jest organizowanym w pamięci operacyjnej podrzędnym zbiorem bieżąco używanych i-węzłów. Zawiera ona pola indeksowe i-węzłów wskazujące na dyskowe bloki danych. W ten sposób po otwarciu pliku wszystko, z wyjątkiem samych bloków danych, znajduje się w pamięci operacyjnej do natychmiastowego użycia przez dowolny proces sięgający do pliku. W rzeczywistości operacja *open* przeszukuje najpierw tablicę otwartych plików, aby sprawdzić, czy plik nie jest już używany przez inny proces. Jeśli tak się wyjaśni, to następuje utworzenie procesowego wpisu w tablicy otwartych plików, prowadzącego do ogólnosystemowej tablicy otwartych plików. W przeciwnym razie następuje przekopowanie i-węzła do tablicy aktywnych i-węzłów, utworzenie nowego wpisu w tablicy ogólnosystemowej oraz nowego wpisu związanego z konkretnym procesem.

System UNIX BSD ze stosowanym w nim podrzędnym przechowywaniem danych w celu oszczędzania operacji dyskowych należy uznać za typowy. Średni współczynnik traflów dla pamięci podrzędnej wynosi tu 85%, co dowodzi, że metody takie są zdecydowanie wartościowe implementowania. Pełny opis systemu UNIX BSD zawiera rozdz. 21. Tablica otwartych plików jest szczegółowo omówiona w p. 10.1.2.

11.1.2 Montowanie systemu plików

Na podobieństwo tego, że plik przed użyciem wymaga otwarcia, system plików musi być zamontowany, zanim stanie się dostępny dla procesów w systemie operacyjnym. Procedura montażu jest prosta. Systemowi operacyjnemu podaje się nazwę urządzenia oraz określa miejsce w strukturze plików, do którego należy przyłączyć system plików (nazywane *punktem montażu* – ang. *mount point*). Na przykład w systemie UNIX system plików zawierający prywatny katalog użytkownika można by zamontować pod nazwą */home*. Wówczas w celu dostępu do struktury katalogowej w obrębie tego systemu plików nazwy katalogów można by poprzedzać przedrostkiem */home*, jak w nazwie */home/jane*. Zamontowanie tego systemu plików w punkcie */users* spowodowałoby, że ścieżka prowadząca do tego samego katalogu przybrałaby postać */users/jane*.

System operacyjny sprawdza następnie, czy urządzenie zawiera właściwy system plików. W tym celu prosi moduł obsługi urządzenia o przeczytanie katalogu urządzenia i sprawdza, czy katalog ma oczekiwany format. Na koniec system operacyjny zaznacza w swojej strukturze katalogowej zamontowanie pliku w określonym punkcie montażu. Metoda ta umożliwia systemowi operacyjnemu przeglądanie jego struktury katalogowej z dokonywaniem odpowiednich przełączeń między systemami plików.

Rozważmy działanie systemu operacyjnego komputera Macintosh. Niekroć system napotka dysk po raz pierwszy (dyski twardc są odnajdywanie podczas rozruchu, dyskietki zauważa się z chwilą włożenia ich do stacji dyskietek), tylekroć poszukuje na danym urządzeniu systemu plików. W przypadku znalezienia systemu plików następuje jego automatyczne zamontowanie na poziomie głównym (korzenia) i umieszczenie na ekranie ikony katalogu zaopatrzonej w nazwę systemu plików (zgodną z zapamiętaną w katalogu urządzenia). Użytkownik może wtedy wyświetlić zawartość nowo zamontowanego systemu plików po prostu przez odniesienie się do ikony.

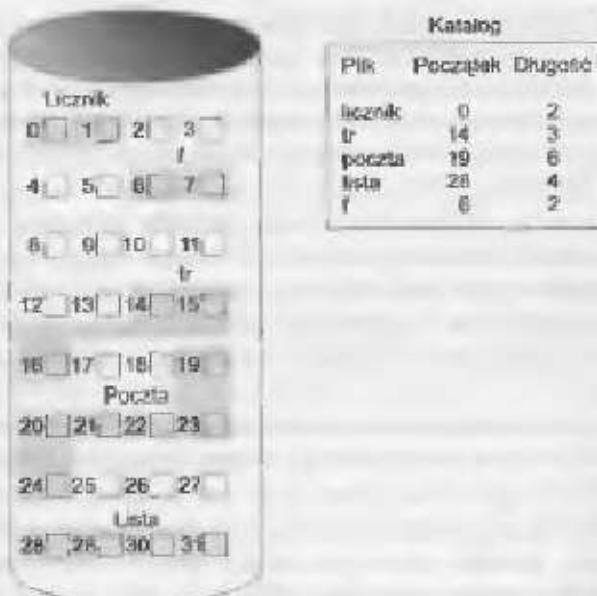
Montowanie systemów plików omawiamy dalej w p. 17.6.2 i 21.7.5.

11.2 ■ Metody przydziału miejsca na dysku

Dzięki charakterystycznej dla dysków właściwości bezpośredniego dostępu możliwe jest elastyczne podejście do zagadnienia implementacji plików. Nieomal zawsze na jednym dysku przechowuje się wiele plików. Podstawową kwestią jest wybór takiego sposobu przydziału miejsca dla plików, aby obszar dysku był zagospodarowany efektywnie, a dostęp do plików szybki. Powszechnie używa się trzech głównych metod przydziału miejsca na dysku: ciągłej, listowej i indeksowej. Każda metoda ma swoje zalety i wady. W związku z tym w niektórych systemach (np. w będącym produktem Data General systemie RDOS przeznaczonym dla komputerów serii Nova) stosuje się wszystkie trzy metody równocześnie. Częściej jednak używa się jednej metody do wszystkich plików w systemie.

11.2.1 Przydział ciągły

W metodzie *przydziału ciągłego* (ang. *continuous allocation*) każdy plik musi zajmować ciąg kolejnych bloków na dysku. Adresy dyskowe definiują na dysku uporządkowanie liniowe. Zauważmy, że w tym uporządkowaniu, przy założeniu, że z dyskiem kontaktuje się tylko jedno zadanie, dostęp do bloku $b + 1$ po bloku b nie wymaga na ogół ruchu głowicy. Gdy ruch głowicy będzie konieczny (między ostatnim sektorem na jednym cylindrze i pierwszym



Rys. 11.3 Przydział ciągły miejsca na dysku

sektorem na następnym cylindrze), wówczas będzie to przesunięcie tylko o jedną ścieżkę. Tak więc liczba operacji przeszukiwania dysku, wymaganych przy rozmieszczaniu plików na dysku metodą ciągłą jest minimalna. Dotyczy to także czasu przeszukiwania, jeśli staje się ono nieuniknione.

Przydział ciągły pliku jest określony za pomocą adresu dyskowego i długości pierwszej porcji (mierzonej w blokach dyskowych). Jeśli plik składa się z n bloków i zaczyna od adresu bloku b , to zajmuje bloki $b, b + 1, b + 2, \dots, b + n - 1$. Wpis katalogowy każdego pliku zawiera adres bloku początkowego i długość obszaru przydzielonego danemu plikowi (rys. 11.3).

Dostęp do pliku, któremu przydzielono miejsce w sposób ciągły, jest łatwy. Przy dostępie sekwencyjnym system plików pamięta adres dyskowy ostatniego bloku, do którego było odniesienie, i w razie potrzeby czyta następny blok. W celu bezpośredniego dostępu do bloku i w pliku zaczynającym się od bloku b można natychmiast odnieść się do bloku $b + i$. Tak więc za pomocą przydziału ciągłego można implementować zarówno dostęp sekwencyjny, jak i swobodny.

Pewną trudnością w przydziale ciągłym jest znalezienie miejsca na nowy plik. O tym, jak zadanie to jest wykonywane, przesąduje implementacja systemu zarządzania wolnymi obszarami, omówiona w p. 11.3. Można w tym celu zastosować dowolny system zarządzania, lecz niektóre systemy są wolniejsze niż inne.

Problem ciągłego przydziału przestrzeni dyskowej można rozpatrywać jako szczególny przypadek ogólnego problemu *dynamicznego przydziału pamięci* (ang. *dynamic storage-allocation*) omówionego w p. 8.4, a polegającego na spełnieniu zamówienia o rozmiarze n na podstawie listy wolnych dziur. Strategie pierwszego dopasowania i najlepszego dopasowania są najczęściej stosowane do wybrania wolnego obszaru z listy dostępnych obszarów. Za pomocą symulacji wykazano, że zarówno pierwsze dopasowanie, jak i najlepsze dopasowanie są lepsze od strategii najgorszego dopasowania pod względem czasu oraz wykorzystania pamięci. Ani dopasowanie pierwsze, ani najlepsze nie są jednak rzeczywiście najlepsze ze względu na wykorzystanie pamięci, lecz działania metodą pierwszego dopasowania są z reguły szybsze.

Algorytmy te są obarczone problemem fragmentacji zewnętrznej. Wskutek tworzenia i usuwania plików wolna przestrzeń dyskowa ulega podziałowi na małe kawałki. *Fragmentacja zewnętrzna* występuje zawsze wtedy, kiedy wolna przestrzeń jest pokawalkowana. Staje się ona kłopotliwa wówczas, gdy największy ciągły kawałek nie wystarcza, aby sprostać zamówieniu; pamięć jest podzielona na pewną liczbę dziur. Zależnie od ogólnej ilości pamięci dyskowej i średniego rozmiaru pliku fragmentacja zewnętrzna może stanowić problem poważny lub błahy.

W niektórych starszych systemach mikrokomputerowych używano przydziału ciągłego w odniesieniu do dyskietek. W celu zapobiegania utratie znaczej ilości miejsca na dysku spowodowanej fragmentacją zewnętrzna użytkownik musiał wykonywać procedurę przepakowywania, która kopowała cały system plików na inną dyskietkę lub na taśmę. Oryginalną dyskietkę opróżniało przy tym zupełnie, tworząc na niej jeden wielki i ciągły obszar. Procedura kopowała następnie pliki z powrotem na dyskietkę, przydzielając im spójne obszary z tej jednej, wielkiej dziury. W ten sposób wszystkie wolne obszary były efektywnie upakowywane w jeden obszar ciągły, rozwiązuje problem fragmentacji. Ceną takiego upakowywania jest czas. Daje się to szczególnie odczuć w przypadku wielkich dysków twardych, na których zastosowano przydział ciągły; upakowywanie całej ich przestrzeni może trwać godzinami i może być wymagane w odstępach tygodniowych. W tym „marowym okresie” system z zasadą nie może wykonywać swoich normalnych działań, dlatego na maszynach stosowanych w produkcji za wszelką cenę dąży się do unikania takiego upakowywania.

Z przydziałem ciągłym wiążą się jeszcze inne problemy. Podstawowym zagadnieniem jest określenie wielkości obszaru potrzebnego dla pliku. Przy tworzeniu pliku musi być znaleziony i przydzielony cały wymagany przez niego obszar. Skąd twórcza (program lub osoba) ma znać rozmiar tworzonego pliku? W pewnych przypadkach określenie to może nie nastąpić trudności

(np. przy kopiowaniu istniejącego pliku), lecz w ogólnym przypadku rozmiar pliku wyjściowego może być trudny do oszacowania.

Jesli przydzieli się plikowi za mało miejsca, to może się okazać, że nie da się go rozszerzyć. Szczególnie przy strategii najlepszego dopasowania obszary po obu stronach pliku mogą okazać się zajęte. Pliku nie da się więc powiększyć w miejscu jego występowania. Istnieją wtedy dwie możliwości. Pierwą jest zakończenie programu użytkownika z odpowiednim komunikatem błędu. Użytkownik musi wówczas przydzielić więcej miejsca i wykonać program od nowa. Takie powtarzanie programu może być kosztowne. Aby temu zapobiec, użytkownik będzie uciekać się do nadmiernego oszacowywania zapotrzebowania na obszar na dysku, co spowoduje poważne marnotrawstwo miejsca.

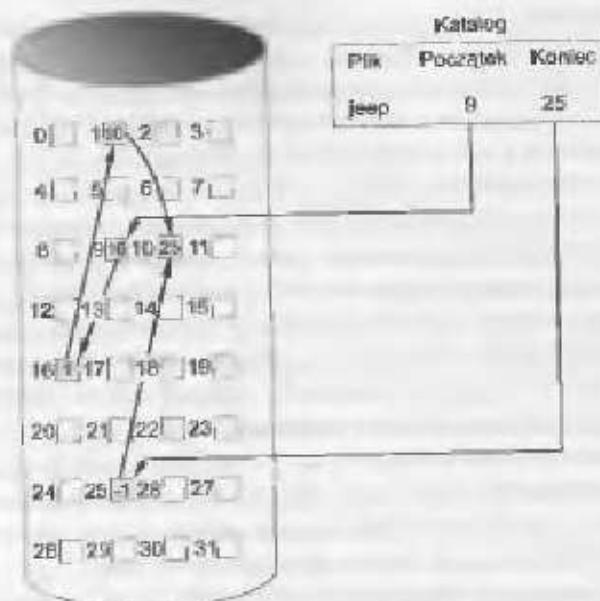
Inna możliwość polega na znalezieniu większej dziury, skopiowaniu zawartości pliku do nowego obszaru i zwolnieniu poprzedniego. Ten ciąg działań – wprawdzie też czasochłonny – można powtarzać tak dugo, jak długo istnieje miejsce na dysku. Zauważmy jednak, że w tym przypadku użytkownik nigdy nie musi być jawnie informowany o tym, co się zdarzyło; system pracuje mimo trudności – inna sprawa, że coraz to wolniej.

Nawet wówczas, gdy łącznie zapotrzebowanie na miejsce dla pliku jest znane z góry, wstępny przydział może nie być wydajny. Plikowi, który rośnie powoli w długim okresie czasu (miesiące lub lata), musimy przydzielić obszar dostosowany do jego koncowego rozmiaru, chociaż znaczna część tej przestrzeni może bardzo dugo pozostawać niewykorzystana. Pociąga to za sobą znaczną *fragmentację wewnętrznej*.

Aby uniknąć kilku z tych wad w niektórych systemach operacyjnych stosuje się zmodyfikowaną metodę przydziału ciągłego, w której na początku przydziela się kawałek ciągłego obszaru, a potem, gdy zaczyna go brakować, do początkowego przydziału dodaje się następny, nadmiarowy kawałek ciągłego obszaru. Na informację o położeniu bloków pliku składa się wówczas adres pierwszego bloku, liczba bloków oraz połączenie z pierwszym blokiem w obszarze nadmiarowym. W pewnych systemach właściciel pliku może określić rozmiar części nadmierowej, lecz jeśli będzie czynić to nieodpowiednio, to regulacje te mogą spowodować pogorszenie wydajności systemu. W przypadku gdy nadmiarowe porcje pliku będą za duże, pojawią się kłopoty z fragmentacją wewnętrzną, a w przypadku przydzielania i zwalniania obszarów nadmiarowych o różnych wielkościach, da znać o sobie fragmentacja zewnętrzna.

11.2.2 Przydział listowy

Przydział listowy (ang. *linked allocation*) usuwa wszystkie kłopoty związane z przydziałem ciągłym. W przydziale listowym każdy plik jest listą powiązanych ze sobą bloków dyskowych; bloki te mogą znajdować się gdziekolwiek



Rys. 11.4. Przydział listowy miejsc na dysku

na dysku. Katalog zawiera wskaźnik do pierwszego i ostatniego bloku pliku. Na przykład plik złożony z pięciu bloków może zaczynać się w bloku 9, jego dalszy ciąg może znajdować się w bloku 16, potem w bloku 1, 10 i kończyć się w bloku 25 (rys. 11.4). Każdy blok zawiera wskaźnik do następnego bloku. Wskaźniki te nie są dostępne dla użytkownika. Jeśli więc każdy blok ma 512 B, a adres dyskowy (wskaźnik) zajmuje 4 B, to użytkownik widzi bloki o długości 508 B.

W celu utworzenia pliku tworzymy po prostu nowy wpis w katalogu. W przydziale listowym każda pozycja w katalogu ma wskaźnik do pierwszego dyskowego bloku pliku. Początkową wartością tego wskaźnika jest *nil* (wskaźnik symbolizujący koniec listy) w celu zaznaczenia, że plik jest pusty. Pole rozmiaru również przyjmuje wartość zero. Operacja pisania powoduje odnalezienie przez system zarządzania wolnymi obszarami nie zajętego bloku i jego zapisanie oraz dowiezanie do końca pliku. Czytanie pliku odbywa się po prostu według wskaźników zapamiętywanych w kolejnych blokach.

W przydziale listowym nie ma zewnętrznej fragmentacji, a do spełnienia zamówienia może być użyty każdy wolny blok z listy wolnych obszarów. Zauważmy też, że nie ma żadnego powodu, by deklarować rozmiar pliku w chwili jego tworzenia. Plik może rosnąć dopóty, dopóki są wolne bloki. Dzięki temu nie ma nigdy potrzeby upakowywania dysku.

Przydział listowy ma również wady. Podstawowym problemem jest ograniczenie jego efektywnego zastosowania jedynie do plików o dostępie sekwencyjnym. Aby znaleźć blok i pliku, należy zacząć od początku pliku i postępować za wskaźnikami aż do dotarcia do bloku i . Każdy dostęp do wskaźnika wymaga czytania dysku. Dlatego też realizacja dostępu bezpośredniego w plikach o przydziale listowym jest niawydajna.

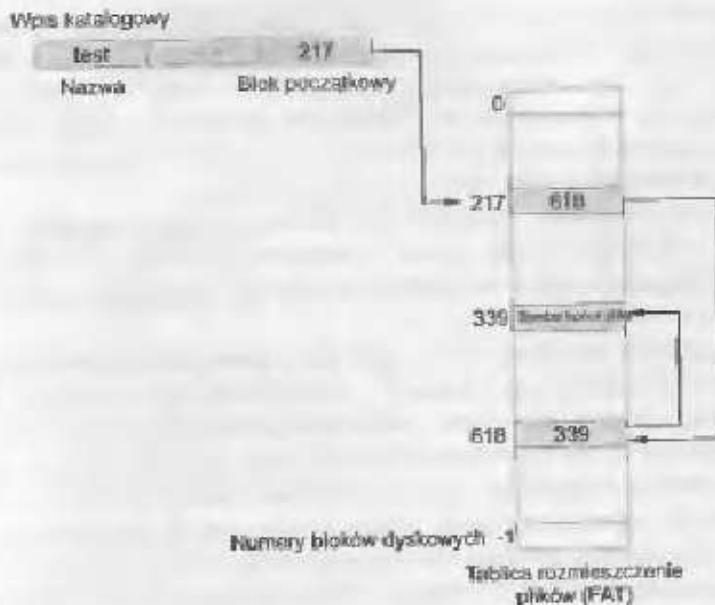
Inną wadą przydziału listowego jest przestrzeń zajmowana przez wskaźniki. Jeśli wskaźnik wymaga 4 B z 512-bytowego bloku, to 0,78% dysku zajmują wskaźniki zamiast informacji. Każdy plik potrzebuje nieco więcej miejsca niż w innym przypadku.

Niedogodność tę usuwa się na ogólny przez grupowanie bloków po kilka w tzw. grona (klastry) (ang. *clusters*)⁷ i przydzielanie gron zamiast bloków. Na przykład system plików może definiować grono wielkości czterech bloków i działać na dysku tylko za pomocą jednostek równych gronom. Wskaźniki będą wówczas zajmować znacznie mniejszy ułamek miejsca na dysku. W tej metodzie odwzorowywanie bloków logicznych na fizyczne pozostaje proste, a jednocześnie polepsza się przepustowość (mniej przeszukań angażujących głowice dyskowe) i zmniejsza obszar wymagany na przydział bloku i zarządzanie listą wolnych przestrzeni. Kosztem tego rozwiązania jest wzrost wewnętrznej fragmentacji, ponieważ w przypadku częściowego zapelnienia grona marnuje się więcej miejsca niż w niepełnym bloku. Grona używane w celu polepszania czasu dostępu do dysku przydają się w wielu innych algorytmach, dlatego występują one w większości systemów operacyjnych.

Kolejnym problemem jest niezawodność. Skoro pliki są powiązane wskaźnikami rozrzuconymi po całym dysku, zauważmy, co by się stało, gdyby jakiś wskaźnik uległ zagubieniu lub uszkodzeniu. Błąd w oprogramowaniu systemu operacyjnego lub w urządzeniu dyskowym mógłby spowodować pobranie złego wskaźnika. Błąd taki mógłby spowodować dowiezanie (pliku) do listy wolnych przestrzeni lub do innego pliku. Częściowym rozwiązaniem jest używanie list z podwojonymi dowiezaniami lub zapamiętywanie w każdym bloku nazwy pliku i względnego numeru bloku; takie schematy zwiększa jednak jeszcze bardziej nakłady ponoszone na każdy plik.

Ważną odmianą metody przydziału listowego jest użycie *tablicy przydziału* (rozmieszczenia) plików (ang. *file allocation table* – FAT). Tę prostą, lecz wydajną metodę dokonywania przydziałów miejsca na dysku zastosowano w systemach operacyjnych MS-DOS i OS/2. Początkowa część każdej strefy na dysku jest zarezerwowana na przechowywanie tablicy. Tablica ta ma po jednej pozycji na każdy biegły dyskowy i jest indeksowana za pomocą nu-

⁷ Nazywane też jednostkami przydziału (ang. *allocation units*) lub – żargonowo – „klasterem”. – Przyp. tłum.



Rys. 11.5 Tablica rozmieszczenia plików

merów bloków. Tablica FAT jest używana w sposób bardzo przypominający listę powiązaną. Wpis katalogowy zawiera numer pierwszego bloku pliku. Pozyция w tablicy, indeksowana przez numer tego bloku, zawiera numer następnego bloku w pliku. Łancuch taki ciągnie się aż do ostatniego bloku, który ma na odpowiadającej mu pozycji w tablicy specjalny symbol końca pliku. Bloki nie używane są wskazywane za pomocą liczby 0 umieszczonej na ich pozycji w tablicy. Przydzielenie nowego bloku do pliku sprowadza się do znalezienia pierwszej zerowej pozycji w tablicy i zastąpienia poprzedniej wartości symbolizującej koniec pliku adresem nowego bloku. Zero zostaje wtedy zastąpione symbolem końca pliku. Na rysunku 11.5 widać przykład struktury FAT, w którym plik składa się z bloków dyskowych o numerach 217, 618 i 339.

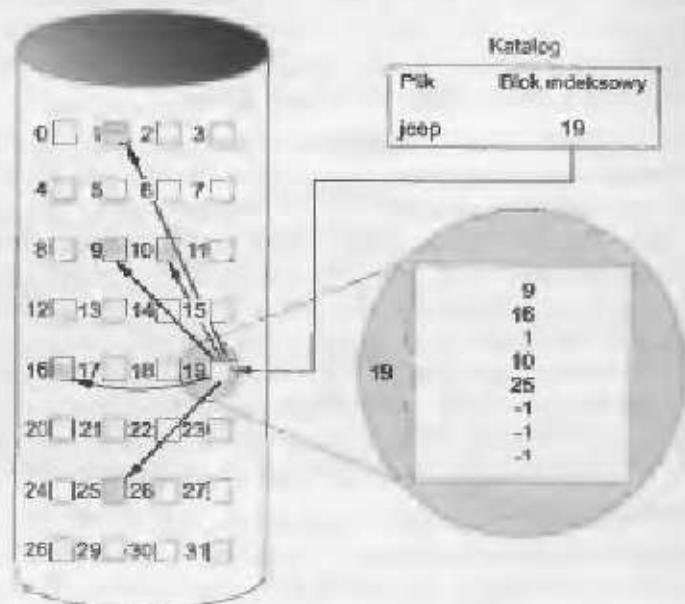
Zwróćmy uwagę, że jeśli tablicy FAT nie przechowuje się w pamięci podręcznej, to przydział według schematu FAT może powodować znaczny ruch głowic dyskowych. Głowica dysku musi przemieszczać się na początek strefy, aby przeczytać dane w tablicy FAT i odnaleźć położenie potrzebnego bloku, a następnie przesuwać się do miejsca występowania tego bloku. W najgorszym przypadku oba przemieszczenia występują dla każdego bloku. Korzyść polega na polepszeniu czasu dostępu swobodnego, ponieważ głowica dyskowa może znaleźć położenie dowolnego bloku na podstawie informacji czytanych w tablicy FAT.

11.2.3 Przydział indeksowy

Przydział listowy rozwiązuje problemy zewnętrznej fragmentacji i deklarowania rozmiaru, występujące w przydziale ciągłym. Niemniej jednak przy braku tablicy FAT przydział listowy nie może służyć do organizacji wydajnego dostępu bezpośredniego, gdyż wskaźniki do bloków są rozrzucone wraz z blokami po całym dysku i muszą być odzyskiwane po kolei. Przydział indeksowy rozwiązuje ten problem przez akumpliowanie wskaźników w jednym miejscu – w *bloku indeksowym* (ang. *index block*).

Każdy plik ma własny blok indeksowy, będący tablicą adresów bloków dyskowych. Pozycja o numerze i w bloku indeksowym wskazuje na blok i danego pliku. Katalog zawiera adres bloku indeksowego (rys. 11.6). Aby przeczytać blok i , używa się wskaźnika z pozycji o numerze i w bloku indeksowym i odnajduje się za jego pomocą odpowiedni blok do czytania. Schemat ten przypomina schemat stroncowania opisany w rozdz. 8.

Półczas tworzenia pliku wszystkie wskaźniki w bloku indeksowym otrzymują wartość *nil*. Gdy blok i jest po raz pierwszy zapisywany, pobiera się go od zarządcy obszarów wolnych, a jego adres zostaje umieszczony w i -tej pozycji bloku indeksowego.



Rys. 11.6 Przydział indeksowy miejsca na dysku

Przydział indeksowy umożliwia dostęp bezpośredni bez powodowania zewnętrznej fragmentacji, ponieważ zamówienie na dodatkowy obszar może spełnić wolny blok znajdujący się gdziekolwiek na dysku.

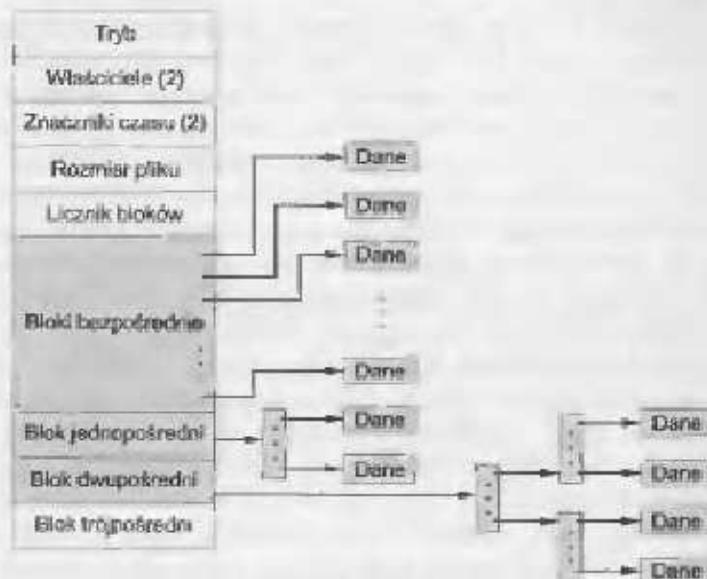
Wadą przydziału indeksowego jest natomiast marnowanie przestrzeni. Wskaźniki bloku indeksowego zajmują na ogół więcej miejsca niż wskaźniki przy przydziale listowym. Rozważmy typowy przypadek, w którym plik ma tylko jeden lub dwa bloki. W przydziale listowym tracimy miejsce dla tylko jednego wskaźnika na blok (jeden lub dwa wskaźniki). W przydziale indeksowym trzeba przypdzielić cały blok nawet wówczas, gdy zaledwie jeden lub dwa wskaźniki będą w nim różne od *nil*.

Ta kwestia nasuwa pytanie o to, jaka powinna być wielkość bloku indeksowego. Każdy plik musi mieć blok indeksowy, toteż chciałoby się, aby blok indeksowy był możliwie jak najmniejszy. Jeśli jednak blok indeksowy będzie za mały, to nie pomoże wystarczającej liczby wskaźników do wielkiego pliku, należałoby więc mieć mechanizm postępowania w tej sytuacji.

- **Schemat listowy:** Blok indeksowy zawiera się zwykle w jednym bloku dyskowym. Dzięki temu on sam może być czytany lub zapisywany bezpośrednio. Aby umożliwić organizowanie wielkich plików, można połączyć kilka bloków indeksowych. Blok indeksowy może na przykład zawierać mały nagłówek z nazwą pliku oraz zbiór stu pierwszych adresów bloków dyskowych. Następny adres (ostanie słowo w bloku indeksowym) będzie miał wartość *nil* (dla małego pliku) lub będzie wskaźnikiem do innego bloku indeksowego (dla dużego pliku).
- **Indeks wielopoziomowy:** Wariantem reprezentacji listowej jest użycie bloku indeksowego pierwszego poziomu do wskazywania zbioru bloków indeksowych drugiego poziomu, których wskaźniki wskazują już na bloki pliku. Aby dojść do bloku, system operacyjny posługuje się indeksem pierwszego poziomu w celu znalezienia bloku indeksowego drugiego poziomu, za pomocą którego odkonwertuje potrzebny blok danych. Podejście takie można kontynuować do trzeciego lub czwartego poziomu w zależności od pożądanej, maksymalnej wielkości pliku. Mając bloki o rozmiarze 4096 B, możemy w bloku indeksowym zapamiętać 1024 czterobajtowe wskaźniki. Dwa poziomy indeksów umożliwiają użycie 1 048 576 bloków danych, co pozwala na operowanie plikami o rozmiarach do 4 GB.
- **Schemat kombinowany:** Innym podejściem, zastosowanym w systemie UNIX BSD, jest przechowywanie – powiedzmy – pierwszych 15 wskaźników bloku indeksowego w i-węźle pliku (ang. *inode*)*. (Wpisy katalogu

* Czyli rekordzie opisującym plik w systemie UNIX. Rekord ten jest również nazywany „blokiem indeksowym pliku”, co w darymukartekscie prowadzi do niejednoznaczności. – Przyp. tłum.

gowe prowadzące do i-węzłów są omówione w p. 21.7). Pierwszych 12 z tych wskaźników wskazuje na *bloki bezpośrednie* (ang. *direct blocks*), tzn. zawierające adresy bloków z danymi pliku. Dzięki temu dane w małych plikach (nie dłuższych niż 12 bloków) nie muszą mieć oddzielnego bloku indeksowego. Jeśli rozmiar bloku wynosi 4 KB, to można osiągać bezpośrednio dane o wielkości dochodzącej do 48 KB. Następne trzy wskaźniki wskazują na *bloki pośrednie* (ang. *indirect blocks*). Pierwszy wskaźnik bloku pośredniego jest adresem *bloku jednoposredniego* (ang. *single indirect block*). Blok jednoposredni jest blokiem indeksowym, który zamiast docelowych danych zawiera adresy bloków z danymi. Dalej następuje wskaźnik *bloku dwuposredniego* (ang. *double indirect block*), który zawiera adres bloku z adresami bloków zawierającymi wskaźniki do faktycznych bloków danych. Ostatni wskaźnik zawierałby adres *bloku trójposredniego* (ang. *triple indirect block*). Dzięki tej metodzie liczba bloków, które można przydzielić do pliku przekracza liczbę obszarów adresowanych za pomocą 4-bajtowych wskaźników stosowanych w wielu systemach operacyjnych. Zakres 32-bitowego wskaźnika plikowego osiąga tylko wartość 2^{32} B, czyli 4 GB. I-węzeł jest pokazany na rys. 11.7.



Rys. 11.7 I-węzeł w systemie UNIX (Zaczerpnięto z: S. Leffler, M. Karels, M. McKusick, J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System* (fig. 7.6, p. 194). © 1989 Addison-Wesley Publishing Company Inc.
Reprinted by permission of Addison Wesley Longman)

Zauważmy, że schemat przydziału indeksowego jest obarczony podobnymi problemami wydajności jak przydział listowy. W szczególności bloki indeksowe mogą być przechowywane w pamięci podręcznej, lecz bloki danych mogą być rozrzucone po całej strefie dyskowej.

11.2.4 Wydajność

Metody przydziału, które omówiliśmy, różnią się pod względem ich zapotrzebowania na pamięć i czasów dostępu do bloków danych. Oba te parametry są ważnymi kryteriami wyboru właściwej metody (lub metod) do zimplementowania w systemie operacyjnym.

Jedną z trudności napotykanych przy porównywaniu działania różnych systemów jest określenie sposobu, w jaki systemy te będą eksploatowane. W systemie, w którym dostęp jest najczęściej sekwencyjny, powinno się zastosować inną metodę niż w systemie, w którym przeważa dostęp swobodny. Dla dowolnego typu dostępu w przydziale ciągłym do pobrania bloku dyskowego wystarcza tylko jeden kontakt z dyskiem. Ponieważ adres początku pliku może być z łatwością przechowywany w pamięci, można natychmiast obliczyć adres dyskowy bloku o numerze i (lub bloku następnego) i przeczytać blok bezpośrednio.

Przy przydziale listowym adres następnego bloku również można przechowywać w pamięci i czytać blok bezpośrednio. Metoda ta jest dobra dla dostępu sekwencyjnego, jednak siegniecie po blok o numerze i może wymagać i operacji czytania z dysku. Ta niedogodność przesądza o tym, że przydział listowy nie powinien być stosowany tam, gdzie jest potrzebny dostęp bezpośredni.

W efekcie, w pewnych systemach dostęp bezpośredni jest uzyskiwany za pomocą przydziału ciągłego, sekwencyjny zaś – za pomocą przydziału listowego. W takich systemach typ dostępu musi być deklarowany przy tworzeniu pliku. Plik utworzony w celu dostępu sekwencyjnego będzie miał strukturę listową i dostęp bezpośredni do niego nie będzie możliwy. Plik utworzony w celu dostępu bezpośredniego będzie ciągły i umożliwi zarówno dostęp bezpośredni, jak i sekwencyjny, ale jego maksymalna długość będzie musiała być zadeklarowana w chwili tworzenia. Zauważmy, że w tym przypadku system operacyjny musi mieć odpowiednie struktury danych i algorytmy umożliwiające obie metody przydziału. Konwersji typów plików można dokonywać przed utworzeniem nowego pliku o żadanym typie i przekopiowanie do niego zawartości starego pliku. Stary plik można wtedy usunąć, a nowy przemianować.

Przydział indeksowy jest bardziej złożony. Jeśli blok indeksowy jest już w pamięci operacyjnej, to dostęp może się odbywać bezpośrednio. Jednak blok indeksowy wymaga sporo miejsca w pamięci. Jeżeli brak jest miejsca w pamięci, to można najpierwczytać blok indeksowy, a potem potrzebny blok danych.

Przy dwupoziomowym indeksie może okazać się niezbędne dwukrotne czytanie bloków indeksowych. Dla wyjątkowo wielkich plików dostęp do bloków położonych blisko końca pliku może wymagać przeczytania wszystkich bloków indeksowych, aby w ślad za wskaźnikami można było dotrzeć w końcu do bloku danych. Zatem działanie przydziału indeksowego zależy od struktury indeksu, od rozmiaru pliku i od położenia potrzebnego bloku.

W niektórych systemach przydział ciągły łączy się z przydziałem indeksowym w ten sposób, że do małych plików (złożonych z trzech lub czterech bloków) stosuje się przydział ciągły i automatycznie przechodzi na przydział indeksowy, jeśli plik się rozrasta. Ponieważ pliki na ogół są niewielkie, a przydział ciągły jest wydajny dla małych plików, średnia uzyskiwana wydajność może być całkiem zadowalająca.

Na przykład wersja systemu operacyjnego UNIX pochodząca z Sun Microsystems została zmieniona w 1991 r. w celu poprawienia wydajności algorytmu przydziału w systemie plików. Pomiarły wydajności wykazały, że praca dysku typowej stacji roboczej (komputer Sparstation 1 o szybkości 12 MIPS) przy maksymalnej przepustowości zajmowała 50% czasu jednostki centralnej, co powodowało, że szerokość pasma w przypadku dysku wynosiła zaledwie 1,5 MB/s. Aby polepszyć wydajność, firma Sun dokonała zmian polegających na przydzielaniu obszarów gronami o rozmiarze 56 KB, gdy tylko było to możliwe. Ten rodzaj przydziału umożliwił zmniejszenie fragmentacji zewnętrznej, a co za tym idzie – również czasu przeszukiwania dysku i wykrywania sektorów. Ponadto procedury czytania dysku poddano optymalizacji pod kątem pobierania informacji z tych wielkich gron. Strukturę i-węzła pozostawiono bez zmian. Te zmiany, wraz z zastosowaniem czytania z wyprzedzeniem i wcześniego zwalniania (metody te są omówione w p. 11.5.2), spowodowały istotne polepszenie przepustowości, zmniejszając zapotrzebowanie na jednostkę centralną o 25%.

Można wymyślić wiele innych optymalizacji i stosuje się wiele różnych ich odmian. Uwzględniając dysproporcję między szybkościami procesora i dysku, dodanie do systemu operacyjnego tysięcy dodatkowych instrukcji w celu zaoszczędzenia kilku ruchów głowicy może okazać się nie pozbawione sensu. Co więcej, ta dysproporcja powiększa się z upływem czasu, osiągając stopień, przy którym daje się uzasadnić użycie setek tysięcy rozkazów w celu optymalizacji ruchu głowic.

11.3 ■ Zarządzanie wolną przestrzenią

Ponieważ obszar dysku jest ograniczony, więc w misę możliwości należy koniecznie dbać o właściwe zagospodarowywanie dla nowych plików przestrzeni po plikach usuniętych. (Dyski optyczne zapisywanej jednorazowo pozwalają tylko

jednokrotnie zapisać dowolny sektor, w związku z czym tego rodzaju wtórne ich czycie jest fizycznie niemożliwe). Aby mieć bieżące informacje o wolnych obszarach dyskowych, system utrzymuje listę wolnych obszarów (ang. *free-space list*). Na liście wolnych obszarów są odnotowane wszystkie bloki, które są wolne, czyli nie przydzielone do żadnego pliku ani katalogu. W celu utworzenia pliku sprawdza się, czy lista wolnych obszarów wykazuje wymaganą ilość wolnego miejsca, po czym przydziela się to miejsce nowemu plikowi. Przydzielone miejsce zostaje wtedy usunięte z listy wolnych obszarów. Podczas usuwania pliku zajmowany przez niego obszar na dysku jest dodawany do listy wolnych obszarów. Wbrew naźwie lista wolnych obszarów nie musi być zaimplementowana w postaci listy, co zaraz omówimy.

11.3.1 Wektor bitowy

Lista wolnych obszarów bywa często implementowana w postaci *mapy bitowej* (ang. *bit map*) lub *wektora bitowego* (ang. *bit vector*). Każdy blok jest reprezentowany przez 1 bit. Jeśli blok jest wolny, to bit ma wartość 1. Jeśli blok jest przydzielony, to dany bit wynosi 0*.

Rozważmy na przykład dysk, którego bloki 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 i 27 są wolne, a pozostałe są przydzielone. Mapa wolnych obszarów przyjmie postać

00111100111110001100000011100000 ...

Podstawową zaletą takiego podejścia jest to, że umożliwia ono względnie proste i wydajne odnajdywanie pierwszego wolnego bloku lub n kolejnych takich bloków na dysku. W istocie, wiele komputerów ma rozkazy przeznaczone do wykonywania operacji na bitach; można w tym celu te rozkazy skutecznie zastosować. Na przykład rodzina procesorów Intel, poczynając od procesora 80386, oraz rodzina procesorów Motorola, poczynając od procesora 68020 (procesory te działają w komputerach PC i Macintosh), rozporządza rozkazami przekazującymi pozycję pierwszego bitu w słowie, który ma wartość 1. W rzeczywistości system operacyjny Apple Macintosh korzysta podczas przydzielania miejsca na dysku z wektora bitowego. W celu znalezienia pierwszego wolnego bloku system operacyjny Macintosh sprawdza, czy kolejne słowa w mapie bitowej są równe 0, ponieważ wyzerowane słowo ma wszystkie bity równe 0 i oznacza zbiór przydzielonych bloków. W pierwszym niezerowym słowie szuka się pierwszego bitu 1, którego położenie** wyznacza lokalizację pierwszego wolnego bloku. Numer bloku oblicza się w sposób następujący:

* Przypisanie znaczeń bitom w mapie bitowej może zależeć od systemu. – Przyp. tłum.

** Liczone od początku mapy. – Przyp. tłum.

$$(liczba bitów w słowie) \times (\text{liczba wyzerowanych słów}) + \\ + \text{pozyja pierwszego bitu } 1$$

I znów widzimy, jak właściwości sprzętu wpływają na funkcjonalność oprogramowania. Niestety, zastosowanie wektorów bitowych jest mało wydajne, jeśli nie można przechowywać całego wektora w pamięci operacyjnej (z której zapisuje się go od czasu do czasu na dysku na wypadek odzyskiwania). Przechowywanie wektora bitowego w pamięci operacyjnej jest możliwe dla mniejszych dysków, takich jak stosowane w mikrokomputerach*, lecz nie dla dysków większych. Aby rejestrować wolne bloki dysku o pojemności 1,3 GB w blokach 512-bajtowych, należałoby użyć mapy bitowej o wielkości ponad 310 KB. Połączenie bloków w czteroblokowe grupy zmniejsza tę liczbę do 78 KB na dysk.

11.3.2 Lista powiązana

Odmiennym podejściem jest powiązanie ze sobą wszystkich wolnych bloków dyskowych i przechowywanie wskaźnika do pierwszego wolnego bloku w specjalnym miejscu na dysku oraz podręcznie – w pamięci. Ten pierwszy blok zawiera wskaźnik do następnego wolnego bloku itd. W naszym przykładzie (zob. p. 11.3.1) należałoby przechować wskaźnik do bloku 2 jako do pierwszego wolnego bloku. Blok 2 powinien zawierać wskaźnik do bloku 3, który powinien zawierać wskaźnik do bloku 4, ten z kolei powinien wskazywać na blok 5, który wskazywałby na blok 8 itd. (rys. 11.8). Jednakże metoda ta nie jest wydajna – aby przejrzeć listę, należy odczytać każdy blok, co wymaga znacznej ilości czasu zużytego na operacje wejścia-wyjścia. Na szczęście przeglądanie listy wolnych bloków nie jest częstym działaniem. Zazwyczaj system operacyjny potrzebuje po prostu wolnego bloku w celu przydzielenia go plikowi, więc korzysta z pierwszego bloku z listy bloków wolnych. Zauważmy, że w metodzie FAT rachowanie wolnych bloków jest włączone do struktury danych dotyczącej przydziału. Nie potrzeba więc uciekać się do osobnej metody.

11.3.3 Grupowanie

Modyfikacją podejścia polegającego na zakładaniu listy wolnych obszarów jest przechowanie adresów n wolnych bloków w pierwszym wolnym bloku. Pierwszych $n - 1$ z tych bloków to bloki rzeczywiście wolne. Ostatni blok zawiera adresy kolejnych n wolnych bloków itd. Znaczenie tej implementacji polega na tym, że – w odróżnieniu od standardowego podejścia listowego – umożliwia ona szybkie odnajdywanie adresów wielkiej liczby wolnych bloków.

* Należy zachować pewien dyskretny w ocenie wielkości dysków/mikrokomputerów, ponieważ – łatwo się zauważać – ich pojemność wzrasta w latach dziesięcioleciowych sukrotnie – Przyp. tłum.



Rys. 11.8 Powiązanie listy wolnych obszarów na dysku

11.3.4 Zliczanie

Jeszcze inny sposób polega na skorzystaniu z faktu, że kilka przylegających do siebie bloków można na ogół przydzielać i zwalniać jednocześnie, zwłaszcza wtedy, gdy stosuje się algorytm przydziału ciągłego lub używa się gron. Toteż zamiast wykazu n wolnych adresów dyskowych można przechowywać adres pierwszego wolnego bloku i liczbę n wolnych bloków następujących jeden za drugim bezpośrednio po nim. Każda pozycja na wykazie wolnych obszarów składa się z adresu dyskowego i licznika. Choć każdy wpis zajmuje więcej miejsca niż zwykły adres dyskowy, cały wykaz będzie krótszy, jeśli liczniki będą zwykle większe niż 1.

11.4 ■ Implementacja katalogu

Wybór algorytmów przydzielania miejsca dla katalogu i zarządzania katalogiem na istotny wpływ na wydajność, działanie i niezawodność systemu plików. Dlatego jest ważne, aby rozumieć występujące w tych algorytmach kompromisy.

11.4.1 Lista liniowa

Najprostsza metoda implementacji katalogu polega na zastosowaniu liniowej listy nazw plików ze wskaźnikami do bloków danych. W celu znalezienia konkretnej pozycji na liniowej liście wpisów katalogowych należy zastosować przeszukiwanie liniowe. Metoda ta jest łatwa do zaprogramowania, lecz kosztowna pod względem czasu zużywanego podczas działania. W celu utworzenia nowego pliku musimy najpierw przeszukać katalog, aby upewnić się, że żaden z istniejących plików nie ma takiej samej nazwy. Następnie dodajemy nowy wpis na końcu katalogu. Aby usunąć plik, odnajduje się w katalogu jego nazwę, po czym zwalnia przydzielony mu obszar. W celu powtórnego wykorzystania pozycji w katalogu możemy postąpić na kilka sposobów. Możemy oznaczyć daną pozycję jako nieużywaną (nadając jej specjalną nazwę, w rodzaju nazwy pustej, lub za pomocą bitu zajęcia-zwolnienia związanego z każdą pozycją) albo możemy dodać ją do wykazu wolnych pozycji katalogowych. Trzecia możliwość polega na skopiowaniu ostatniej pozycji w katalogu na zwolnione miejsce i zmniejszeniu długości katalogu. Aby zmniejszyć czas usuwania pliku, można też posłużyć się listą prwiązaną.

Prawdziwą wadą liniowej listy wpisów katalogowych jest liniowe poszukiwanie pliku. Informacje katalogowe są często używane i powolna implementacja dostępu do nich została zauważona przez użytkowników. W rzeczywistości wiele systemów operacyjnych realizuje programową pamięć podręczną, aby przechowywać ostatnio używane informacje katalogowe. Odnajdywanie informacji w pamięci podręcznej pozwala unikać ustawicznego odczytywania ich z dysku. Lista uporządkowana umożliwia przeszukiwanie binarne i zmniejszenie średniego czasu przeszukiwania. Jednak wymaganie, aby lista była uporządkowana może skomplikować tworzenie i usuwanie plików, gdyż utrzymywanie porządku może wymagać przemieszczania sporych ilości informacji katalogowych. Pomocna mogłaby się tu okazać bardziej wymyślna, drzewiasta struktura danych, taka jak B-drzewo. Zaletą listy uporządkowanej jest możliwość wyprowadzania jej formie uporządkowanej bez wykonywania dodatkowego kroku sortowania.

11.4.2 Tablica haszowania

Odmienią strukturą danych znajdująca zastosowanie w katalogach plików jest *tablica haszowania* (ang. *hash table*). W tej metodzie wpisy katalogowe są przechowywane na liście liniowej, lecz stosuje się również haszowaną strukturę danych. Wartość obliczona na podstawie nazwy pliku^{*} jest odnoszona do

^{*} Za pomocą funkcji haszowania, czyli równoczesnie odwzorowującej argumenty na większe przedziały wartości. — Przyp. tłum.

tablicy haszowania, z której pobiera się wskaźnik do nazwy pliku na liście liniowej. Można dzięki temu znacznie zmniejszyć czas przeszukiwania katalogu. Wstawianie i usuwanie jest także zupełnie proste, chociaż trzeba uwzględnić sytuacje kolizyjne, tj. takie, w których wyniki haszowania nazw dwóch plików odnoszą się do tego samego miejsca. Poważną trudnością w użyciu tablicy haszowania jest na ogół jej stały rozmiar oraz zależność funkcji haszowania od tego rozmiaru.

Załóżmy na przykład, że przygotowujemy liniowo sondowaną tablicę haszowania o 64 pozycjach. Funkcja haszowania zmienia nazwy plików na liczby całkowite z przedziału 0-63 – na przykład za pomocą reszty z dzielenia przez 64. Jeżeli w późniejszym czasie spróbujemy utworzyć sześćdziesiąty piąty plik, to będziemy musieli powiększyć katalogową tablicę haszowania, powiedzmy do 128 wpisów. Wskutek tego będziemy potrzebować nowej funkcji haszowania, która powinna odwzorowywać nazwy plików na przedział 0-127 oraz będziemy zmuszeni zreorganizować istniejące wpisy katalogowe, aby odpowiadały wartościom nowej funkcji haszowania. Można by też zastosować dodzoną, nadrzędową tablicę haszowania. Każda pozycja haszowania mogłaby być listą powiązaną, a nie zwyczajną wartością, moglibyśmy więc rozwiązywać sytuacje kolizyjne za pomocą dodawania nowego wpisu na liście. Przeszukiwanie byłoby cokolwiek wolniejsze, ponieważ odnajdywanie nazwy mogłoby wymagać przechodzenia przez listę kolidujących wpisów tablicy, lecz zapewne byłoby ono znacznie szybsze niż liniowe przeszukiwanie całego katalogu.

11.5 ■ Efektywność i wydajność

Teraz, kiedy już omówiliśmy przydział bloków i możliwości zarządzania katalogami, możemy zastanowić się nad ich wpływem na skuteczność i wydajność posługiwania się dyskiem. Dyski bywają główną przeszkodą sprawnego działania systemu, ponieważ są najwolniejszymi spośród podstawowych elementów komputera. W tym punkcie omawiamy różnorodne metody stosowane w celu poprawienia efektywności i wydajności pamięci pomocniczej.

11.5.1 Efektywność

Efektywne wykorzystanie miejsca na dysku w dużym stopniu zależy od algorytmów przydziału tego miejsca i algorytmów obsługiwanego katalogów. Na przykład i-węzły systemu UNIX są wstępnie rozmieszczane w strefie dyskowej. Nawet „pusty” dysk traci pewną część swojej przestrzeni na i-węzły. Jednak dzięki wstępnomu przydziałowi i-węzłów i rozrzuceniu ich w obrębie

strefy polepszamy wydajność systemu. Poprawa wydajności jest wynikiem unikowych metod przydziału i algorytmów zarządzania wolnymi obszarami, w których dołożono starań, aby bloki danych pliku występowały blisko bloku z i-węzłem danego pliku, co oszczędza czasu przeszukiwania dysku.

Jako inny przykład rozważmy schemat łączenia bloków w grom, omówiony w p. 11.2, który za cenę wewnętrznej fragmentacji pomaga poprawiać wydajność odnajdywania plików i ich przekazywania. Aby zmniejszyć tę fragmentację, w systemie UNIX BSD zastosowano zmienne rozmiary grom, zależne od wzrostu pliku. Jeśli istnieje szansa na zapelnienie dużych grom, to się ich używa, a w przypadku małych plików oraz ostatnich grom w plikach używa się grom małego rozmiaru. System ten opisano w rozdz. 21.

Przemyślenia wymagają również typy danych zazwyczaj przechowywanych we wpisie katalogowym pliku (lub i-węzle). Powszechną praktyką jest zapisywanie „daty ostatniego pisania” w celu zapatrywania użytkownika w informację i określanie, czy plik powinien być składowany. Niektóre systemy utrzymują też „datę ostatniego dostępu”, dzięki czemu użytkownik może określić, kiedy plik był ostatnio czytany. Przechowywanie tej informacji wymaga zapisywania pola w strukturze katalogowej przy każdym czytaniu pliku. Wykonanie takiej zmiany powoduje konieczność przeczytania bloku do pamięci, uaktualnienia jego fragmentu i zapisania bloku z powrotem na dysku, ponieważ operacje dyskowe dotyczą wyłącznie bloków lub ich grom. Tak więc przy każdym otwarciu pliku do czytania należy także przeczytać i uaktualnić jego wpis katalogowy. Ten wymóg może być trudny do zrealizowania w przypadku często używanych plików. Należy więc porównać korzyści wynikające z jego spełnienia z kosztami powodowanymi jego istnieniem w projektowanym systemie plików. Ogólnie biorąc, należy rozważyć wpływ na efektywność i wydajność każdego elementu danych kojarzonego z plikiem.

Jako przykład przemyślmy wpływ rozmiaru wskaźników stosowanych przy dostępie do danych na efektywność systemu. W większości systemów korzysta się ze wskaźników o rozmiarze 16 lub 32 B w każdym miejscu systemu. Takie rozmiary wskaźników ograniczają długość pliku do 2^{16} (64 KB) lub 2^{32} B (4 GB). W pewnych systemach zaimplementowano wskaźniki 64-bitowe, aby przesunąć to ograniczenie do 2^{64} B, co w istocie jest już bardzo wielką liczbą. Jednakże 64-bitowe wskaźniki zajmują więcej pamięci i w konsekwencji powodują większe zużycie przestrzeni dyskowej przy wykonywaniu przydziałów i zarządzaniu wolnymi obszarami (na listy powiązane, indeksy itp.).

Jedną z trudności przy dobieraniu rozmiaru wskaźnika, a gruncie rzeczy przy określaniu w systemie operacyjnym rozmiaru każdego stałego przydziału, jest branie pod uwagę skutków zmian technologicznych. Zauważmy, że komputer IBM PC XT był wyposażony w dysk twardy o pojemności 10 MB i system

plików MS-DOS, który zarządzał tylko obszarem 32 MB. (Każdy wpis w tablicy FAT miał 12 bitów i wskazywał na grono o rozmiarze 8 KB). Ze wzrostem pojemności dysków większe dyski trzeba było dzielić na 32-megabajtowe strefy, ponieważ system plików nie mógł adresować bloków powyżej 32 MB. Z chwilą spopularyzowania się dysków o pojemności przekraczającej 100 MB dyskowe struktury danych i algorytmy w systemie MS-DOS musiały zostać zmienione, aby umożliwić korzystanie z większych systemów plików. (Wpisy w tablicy FAT rozszerzono do 16, a potem do 32 bitów). Pierwotne decyzje dotyczące systemu plików były podektowane względami efektywności. Jednak z pojawieniem się czwartej wersji systemu MS-DOS miliony użytkowników komputerów odczuły niewygodę, gdy przyszło przerzucić się na nowy, większy system plików.

Jako inny przykład rozważmy ewolucję systemu operacyjnego Solaris. Początkowo wiele jego struktur danych miało stałą długość ustaloną podczas rozruchu systemu. Do struktur tych należała tablica procesów oraz tablica otwartych plików. Gdy dochodziło do zapelnienia tablicy procesów, nie można było utworzyć żadnego nowego procesu. Zapelnienie tablicy plików powodowało niemożność otwarcia następnego pliku. Taki system był narażony na załamanie w trakcie świadczenia usług dla użytkowników. Rozmiary tablic mogły być zwiększane tylko przez powtórną komplikację jądra i wznowienie działania systemu. Od wydania 2 system Solaris ma prawie wszystkie struktury jądra przydzielane dynamicznie, co usuwa owe sztuczne ograniczenia w jego zachowaniu. Oczywiście, algorytmy wykonywane na tych tablicach są bardziej złożone, a system operacyjny jest nieco wolniejszy, gdyż musi przydziełać i zwalniać elementy tablic w sposób dynamiczny. Jest to jednak zwyczajna cena, jaką przychodzi płacić za ogólniejsze działanie.

11.5.2 Wydajność

Po wybraniu podstawowych metod dyskowych nadal istnieje kilka dróg poprawiania wydajności. Jak zauważyliśmy w rozdz. 2, większość sprzętowych sterowników dyskowych zawiera lokalną pamięć podręczną „na płycie”, wystarczającą dużą, aby pomieścić jednorazowo całe ścieżki dysku. Po odrakietowaniu ścieżki następuje przeczytanie jej do pamięci podręcznej dysku, poczynając od sektora znajdującego się pod głowicą (przez co redukuje się czas związany z wykrywaniem sektora). Sterownik dysku przesyła wówczas żądane sektory do systemu operacyjnego. Gdy bloki ze sterownika dysku przedostaną się do pamięci głównej, wówczas system operacyjny może je w niej przechować podręcznie. W niektórych systemach są wydzielone osobne miejsca w pamięci operacyjnej na utrzymywanie pamięci podręcznej bloków dyskowych (ang. *disk cache*), w której bloki przechytymuje się z założeniem, że

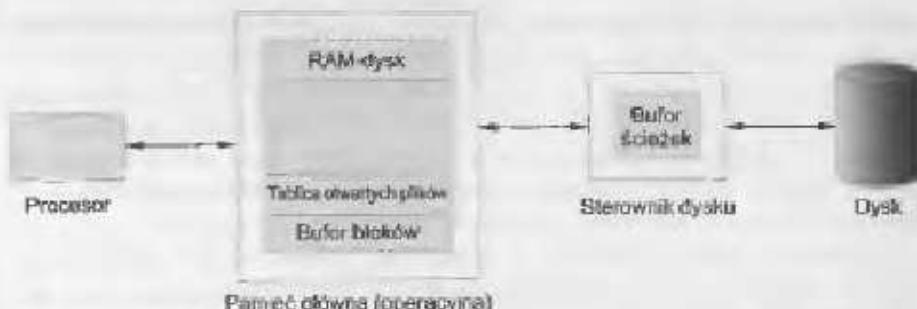
wkrótce staną się znów potrzebne. W innych systemach (jak np. w wersjach systemu UNIX firmy Sun) całą niewykorzystaną pamięć operacyjną traktuje się jako pulę buforów, dzieloną przez system stronicowania i system podręcznego przechowywania bloków dyskowych. System wykonujący dużo operacji wejścia-wyjścia będzie wykorzystywał większość pamięci w charakterze podręcznej pamięci bloków, natomiast system wykonujący dużo programów będzie wykorzystywał pamięć w większości jako obszar stronicowania.

Niektóre systemy optymalizują podręczne pamięci bloków dyskowych, stosując różne algorytmy zastępowania bloków w zależności od rodzaju dostępu do pliku. Bloki pliku czytanego lub zapisywanego sekwencyjnie nie powinny być zastępowane według algorytmu LRU, ponieważ (właściwie) najnowsze bloki nie będą dugo używane, a być może nie zostaną użyte w ogóle. Zamiast tego dostęp sekwencyjny można optymalizować za pomocą metod znanych jako *wczesne zwalnianie* (ang. *free-behind*)¹ i *czytanie z wyprzedzeniem* (ang. *read-ahead*). Wczesne zwalnianie oznacza usuwanie bloku z bufora, gdy tylko pojawi się zamówienie na następny blok. Poprzednie bloki prawdopodobnie nie będą potocznie używane i marnują miejsce w buforze. Czytanie z wyprzedzeniem polega na przeczytaniu wraz z zamawianym blokiem kilku kolejnych bloków i przechowaniu ich w pamięci podręcznej. Przewiduje się, że być może te bloki będą zamawiane po przetworzeniu bloku bieżącego. Sprowadzenie bloków z dysku za pomocą jednej transmisji i podręczne ich przechowanie oszczędza sporą ilość czasu. W systemie wieloprogramowym pamięć podręczna ścieżek w sterowniku dysku nie wyklucza zaspotrzebowania na czytanie z wyprzedzeniem, co wynika z dużych opóźnień i pracochłonności wielu małych przesłanek z podręcznej pamięci ścieżek do pamięci głównej.

W komputerach osobistych do polepszania wydajności z użyciem pamięci głównej stosuje się powszechnie jeszcze inną metodę. Wydziela się część pamięci i traktuje ją jako *dysk virtualny*, inaczej – *RAM-dysk*². W takim rozwiąaniu moduł obsługi urządzenia dysku wirtualnego przyjmuje wszystkie standardowe operacje dyskowe, lecz zamiast na dysku – wykonuje je w wydzielonej części pamięci operacyjnej. Na RAM-dysku można w ten sposób wykonywać wszystkie operacje dyskowe, a użytkownicy, poza błyskawicznym działaniem takich operacji, nie dostrzegają żadnych innych różnic. Niestety, dyski wirtualne są pożyteczne tylko jako pamięć tymczasowa, ponieważ awaria zasilania lub ponowne uruchomienie systemu powoduje z reguły utratę ich zawartości. Przechowuje się na nich na ogół pliki tymczasowe, takie jak pośrednie produkty pracy kompilatorów.

¹ Inaczej: upróżnianie tylów – Przyp. tłum.

² Nawet w najszliskich dyskach czas tca wynosi kilka milisekund. – Przyp. tłum.



Rys. 11.9 Różne miejsca występowania podręcznej pamięci informacji dyskowej

Różnica między RAM-dyskiem a pamięcią podręczną bloków dyskowych polega na tym, że zawartość RAM-dysku pozostaje w całości pod nadzorem użytkownika, natomiast podręczna pamięć bloków dyskowych jest kontrolowana przez system operacyjny. Dysk wirtualny pozostaje na przykład pusty do chwili, w której użytkownik (lub nadzorowany przez niego program) utworzy na nim pliki. Na rysunku 11.9 są pokazane możliwe umiejscowienia pamięci podręcznej w systemie.

11.6 ■ Rekonstrukcja

Ponieważ pliki i katalogi są przechowywane zarówno na dysku, jak i w pamięci operacyjnej, należy dołożyć starań, aby zapewnić, że awaria systemu nie spowoduje utraty danych ani nie doprowadzi do ich niespójności.

11.6.1 Sprawdzanie spójności

Jak powiedzieliśmy w p. 11.4, część informacji katalogowych jest przechowywana w pamięci operacyjnej (pamięć podręczna) w celu przyspieszania dostępu. Informacje katalogowe w pamięci operacyjnej są na ogół nowsze niż odpowiadające im informacje na dysku, gdyż zapisanie na dysku informacji katalogowych zgromadzonych w pamięci podręcznej nie zawsze odbywa się wraz z ich aktualnianiem.

Weźmy pod uwagę możliwe skutki awarii komputera. Tablica otwartych plików z reguły zostaje utracona, a wraz z nią giną wszystkie zmiany wprowadzone w katalogach otwartych plików. Zdarzenie takie może pozostawić system plików w stanie niespójnym – bieżący stan niektórych plików będzie niezgodny z widniejącym w strukturze katalogowej. Podeczas rozruchu systemu często wykonuje się specjalny program wykrywający i korygujący niespójności występujące na dysku.

Program sprawdzania spójności (ang. *consistency checker*) porównuje dane w strukturze katalogowej z blokami danych na dysku i próbuje usunąć wszelkie napotykane niezgodności. Kłopoty, jakie może napotkać program sprawdzania spójności, oraz możliwość ich naprawy wynikają z algorytmów przydziału oraz zarządzania wolnymi obszarami. Jeśli na przykład stosuje się przydział listowy i istnieje połączenie między każdym dwoma blokami, to na postawie bloków danych można zrekonstruować cały plik i odtworzyć strukturę katalogową. Utrata wpisu katalogowego w systemie z przydziałem indeksowym mogłaby być bardziej bolesna, gdy w blokach danych nie ma żadnych informacji dotyczących ich wzajemnego powiązania. Z tego powodu w systemie UNIX wpisy katalogowe używane do czytania są przechowywane podręcznicie, lecz wszelkie operacje pisania prowadzące do przydzielenia obszaru z zasady powodują, że przed zapisaniem bloków danych na dysku zostanie zapisany blok i-węzła.

11.6.2 Składowanie i odtwarzanie

Ponieważ dyski magnetyczne ulegają od czasu do czasu awariom, należy troszczyć się o to, aby zawarte na nich dane nie zostały utracone na zawsze. W tym celu można stosować programy systemowe do *składowania* (ang. *back up*) danych z dysku na innym urządzeniu magazynowania informacji, takim jak dyski elastyczne, taśmy magnetyczne lub dyski optyczne. Zrekonstruowanie utraconego pliku lub całego dysku może wówczas polegać na *odtworzaniu danych* (ang. *restoring*) na postawie danych zawartych w kopii zapasowej.

Aby minimalizować niezbędnego kopianie, można korzystać z informacji o plikach zawartych w każdym wpisie katalogowym. Jeżeli na przykład program wykonujący kopię zapasową zna czas ostatniego składowania pliku, a z zapamiętaną w katalogu datą ostatniego pisania do pliku wynika, że plik nie zmienił się od tamtego czasu, to pliku takiego nie trzeba ponownie kopować. Typowy plan składowania może więc wyglądać następująco:

- **Dzień 1:** Kopiowanie na nośnik zapasowy wszystkich plików z dysku. Nazywa się to *składowaniem pełnym* (ang. *full backup*).
- **Dzień 2:** Kopiowanie na inny nośnik wszystkich plików zmienionych od dnia 1. Jest to *składowanie przyrostowe* (ang. *incremental backup*).
- **Dzień 3:** Kopiowanie na inny nośnik wszystkich plików zmienionych od dnia 2.

- **Dzień N:** Kopiowanie na inny nośnik wszystkich plików zmienionych od dnia $N - 1$. Przejście do dnia 1.

Składowanie w nowym cyklu może być wykonane na miejscu poprzedniego zbioru lub na nowym zbiorze nośników zapasowych. W ten sposób możemy odtworzyć cały dysk, rozpoczęjąc rekonstrukcję od składowania pełnego i kontynuując ją z wykorzystaniem każdego ze składowań przyrostowych. Rzecz jasna, im większe jest N , tym więcej taśm lub dysków trzeba będzie przeczytać w celu wykonania całkowitej rekonstrukcji. Dodatkową zaletą takiego cyklu składowania jest możliwość odtworzenia dowolnego pliku usuniętego nieopatrznie podczas cyklu na podstawie składowania z dnia poprzedniego. Długość cyklu stanowi kompromis między ilością nośników składowania i liczbą dni, z których można będzie wykonać rekonstrukcję.

Może się zdarzyć, że użytkownik zauważycie zaginięcie lub uszkodzenie któregoś z plików dłużej po wystąpieniu tej usterki. Aby się chronić na wypadek takiej sytuacji, praktykuje się wykonanie od czasu do czasu pełnego składowania, które zachowuje się „na zawsze”, tzn. zawierający je nośnik nie wraca do obiegu. Za rozsądne należy także uznać przechowywanie trwałych kopii z datą od zwykłych składowań, aby uchronić je przed przypadkami losu, jak pożar niszczący komputer wraz ze wszystkimi składowaniami. Ponadto, jeśli nośników do składowań używa się cyklicznie, to należy zwrócić uwagę na to, aby nie korzystać z tych samych nośników zbyt wiele razy – w przypadku starcia nośnika odtworzenie danych ze składowanych kopii mogłoby się okazać nietykalne.

11.7 ■ Podsumowanie

System plików pozostaje stale w pamięci pomocniczej, której głównym zadaniem jest przechowywanie wielkiej ilości danych w sposób trwały. Najpopularniejszym nośnikiem pamięci pomocniczej jest dysk.

Różnym plikom można przydzielać obszary na dysku trzema sposobami: za pomocą przydziału ciągłego, listowego oraz indeksowego. Przydział ciągły może być obarczony zewnętrzna fragmentacją. W przydziale listowym nie można uzyskać wydajnego dostępu swobodnego. Przydział indeksowy może wymagać sporego nakładu pamięci na blok indeksów. Istnieje wiele sposobów optymalizowania tych algorytmów. Obszar ciągły można powiększać za pomocą obszarów nadmiarowych, zwiększając elastyczność i zmniejszając fragmentację zewnętrzną. Przydziału indeksowego można dokonywać gronami złożonymi z wielu bloków, aby zwiększyć przepustowość i zmniejszyć wymaganą liczbę wpisów indeksowych. Indeksowanie z użyciem wielkich gron jest podobne do przydziału ciągłego z obszarami nadmiarowymi.

Metody przydziału obszarów wolnych również wpływają na efektywność wykorzystania miejsca na dysku, wydajność systemu plików i niezawodność.

pamięci pomocniczej. Stosuje się tu metody wektorów bitowych i list powiązanych. Do optymalizacji należy grupowanie, zliczanie oraz zastosowanie tablicy FAT, w której lista powiązana występuje w obszarze ciągłym.

W procedurach zarządzania katalogami należy uwzględnić efektywność, wydajność i niezawodność. Najczęściej stosowana metoda polega na użyciu tablicy haszowania – jest ona szybka i efektywna. Niestety, uszkodzenie takiej tablicy lub awaria systemu może doprowadzić do niezgodności informacji katalogowych z zawartością dysku. Do naprawy uszkodzeń stosuje się wówczas program sprawdzania spójności, czyli taki program systemowy jak **fsck** w systemie UNIX lub **chkdsk** w systemie MS-DOS.

■ Ćwiczenia

11.1 Przypuśćmy, że mamy plik, który w danej chwili składa się ze 100 bloków. Założmy, że blok kontrolny pliku (j. blok indeksowy w przypadku przydziału indeksowego) znajduje się już w pamięci. Oblicz, ile dyskowych operacji wejścia-wyjścia wymaga każda ze strategii przydziału: ciągła, listowa i indeksowa (jednopozycyjowa), jeśli dla danego bloku ma być wykonana jedna z poniższych operacji. W przypadku przydziału ciągłego zakładamy, że nie ma więcej miejsca do rozrastania się pliku na początku, ale jest wolne miejsce na jego końcu. Zakładamy też, że blok informacji, która ma być dodana do pliku jest przechowywany w pamięci operacyjnej.

- (a) Blok jest dodawany na początku.
- (b) Blok jest dodawany w środku.
- (c) Blok jest dodawany na końcu.
- (d) Blok jest usuwany z początku.
- (e) Blok jest usuwany ze środka.
- (f) Blok jest usuwany z końca.

11.2 Rozważmy system, w którym informacja o wolnej przestrzeni na dysku jest trzymana na liście wolnych obszarów.

- (a) Założmy, że został utracony wskaźnik do tej listy. Czy system może zrekonstruować listę wolnych obszarów?
- (b) Zaproponuj postępowanie gwarantujące, że wskaźnik taki nigdy nie zginie wskutek błędu pamięci.

- 11.3 Jakie problemy mogłyby powstać, gdyby w systemie dopuszczone jednocześnie zamontowanie systemu plików w kilku miejscach?
- 11.4 Dlaczego mapa bitowa obszarów przydzielonych plikowi powinna być przechowywana w pamięci masowej, a nie w pamięci operacyjnej?
- 11.5 Rozważ system realizujący strategię przydziału ciągłego, listowego i indeksowego. Jakie kryteria trzeba brać pod uwagę przy decyduowaniu, która ze strategii powinna zostać użyta do konkretnego pliku?
- 11.6 Rozważmy dyskowy system plików, którego bloki logiczne i fizyczne mają rozmiar 512 B. Założmy, że informacja o każdym z plików jest już w pamięci. Dla każdej z trzech strategii przydziału (ciąglej, listowej i indeksowej) udziel odpowiedzi na pytania:
- (a) Jak wygląda w tym systemie odwzorowanie adresów logicznych na fizyczne? (Dla przydziału indeksowego założmy, że długość pliku nie przekracza nigdy 512 bloków).
 - (b) Jeśli znajdujemy się przy bloku logicznym 10 (ostatni dostęp dotyczył bloku 10) i chcemy dotrzeć do logicznego bloku 4, to ile fizycznych bloków trzeba będzie przeczytać z dysku?
- 11.7 Jednym z problemów przydziału ciągłego jest konieczność przewidywania przez użytkownika wystarczającej ilości miejsca dla każdego pliku. Jeśli plik przekracza zaplanowany rozmiar, to trzeba podejmować specjalne działania. Jednym z rozwiązań jest przydzielanie plikowi pewnego obszaru początkowego (o określonym rozmiarze), który – po zapelnieniu – jest automatycznie łączony przez system operacyjny z dodatkowym obszarem nadmiarowym. To samo może dotyczyć obszaru nadmiarowego po jego zapelnieniu. Porównaj taką implementację pliku z standardową implementacją ciągłą i listową.
- 11.8 Fragmentacja pamięci w urządzeniu może być usunięta przez przepakowanie informacji. Typowe urządzenie dyskowe nie ma rejestrów przemieszczenia uniwersalnego (które są używane przy upakowywaniu pamięci operacyjnej). Jak zatem można przemieszczać pliki? Podaj trzy przyczyny unikania częstego przepakowywania i przemieszczania plików.
- 11.9 W jaki sposób pamięci podręczne polepszają wydajność? Dlaczego w systemach nie stosuje się więcej lub większych pamięci podręcznych, skoro są one takie pomocne?
- 11.10 W jakich sytuacjach użycie pamięci operacyjnej w roli RAM-dysku byłoby wygodniejsze niż posłużenie się nią jako pamięcią podręczną?

11.13 Dlaczego jest korzystne dla użytkownika, aby system operacyjny dynamicznie przydzielał swoje wewnętrzne tablice?

11.12 Rozważ następujący schemat składowania:

- **Dzień 1.** Kopiuje na nośnik zapasowy wszystkie pliki z dysku.
- **Dzień 2.** Kopiuje na inny nośnik wszystkie pliki zmienione od dnia 1.
- **Dzień 3.** Kopiuje na inny nośnik wszystkie pliki zmienione od dnia 1.

W stosunku do planu podanego w p. 11.6.2 różnica polega tu na wykonywaniu wszystkich następnych kopii zapasowych z odniesieniem do pierwszego składowania pełnego. Jakie są zalety tego systemu w porównaniu z systemem z p. 11.6.2? Jakie są jego wady? Czy wykonanie rekonstrukcji będzie łatwiejsze, czy trudniejsze? Odpowiedź uzasadnij.

Uwagi bibliograficzne

Schemat zarządzania przestrzenią dyskową systemu Apple Macintosh jest omówiony w Apple [13], [14]. System MS-DOS/FAT omówili Norton i Wilton w książce [312]. Opis systemu OS/2 można znaleźć w książce Jacobuego [185]. Wymienione systemy operacyjne działają, odpowiednio, na procesorach rodzin Motorola MC68000 [300] i Intel 8086 [189], [190], [191] i [193]. Metody przydziału stosowane przez firmę IBM opisuje Deitel w książce [96]. Wewnętrzna budowa systemu UNIX/BSD przedstawili szczegółowo Leffler i in. [245]. McVoy i Kleiman zaprezentowali ulepszenia metod użytych w systemie SunOS [280].

Przydział miejsca dla plików dyskowych oparty na systemie kumpla omówili Koch [219]. Schemat organizacji plików zapewniający odzyskiwanie za pomocą jednego dostępu przedstawili Larson i Kajla w artykule [240].

Pamięć podrzędną bloków dyskowych omówili: McKeon [276] i Smith [396]. Przechowywanie podrzędne w eksperymentalnym systemie operacyjnym Sprite opisali Nelson i in. w artykule [308]. Ogólne omówienie technologii pamięci masowych przedstawili Chi [72] i Hoagland [171]. Folk i Zoellick w książce [137] zawarli przegląd struktur plikowych.



Część 4

SYSTEMY WEJŚCIA-WYJŚCIA

Urządzenia przyłączone do komputera różnią się pod wieloma względami. Jednorazowo urządzenie przesyła jeden znak lub blocz znaków. Dostęp do urządzeń może odbywać się tylko sekwencyjnie lub swobodnie. Urządzenia przesyłają dane synchronicznie lub asynchronicznie, są użytkowane w sposób wyłączny (dedykowane) lub są dzielone (użytkowane wspólnie). Urządzenia mogą umożliwiać tylko czytanie lub zarówno czytanie, jak i pisanie; różnią się również znacznie szybkością działania. Z wielu powodów urządzenia są również najwolniejszymi częściami komputera.

Ze względu na całą tę różnorodność system operacyjny powinien oddawać do dyspozycji aplikacji szeroki zakres funkcji umożliwiających im panowanie nad wszystkimi właściwościami urządzeń. Jednym z podstawowych celów podsystemu wejścia-wyjścia systemu operacyjnego jest tworzenie możliwie prostszego interfejsu z resztą systemu. Ponieważ wydajność urządzeń jest wąskim gardłem w systemie, innym kluczowym celem jest optymalizacja wejścia-wyjścia pod kątem uzyskiwania jak największej współbieżności. W rozdziale 12 opisujemy różnorodne odmiany urządzeń wejścia-wyjścia oraz sposoby, za pomocą których system operacyjny sprawuje nad nimi kontrolę. Pozostałe rozdziały w tej części książki są poświęcone omówieniu bardziej skomplikowanych urządzeń wejścia-wyjścia, stosowanych jako pamięci pomocnicze pierwszego i drugiego stopnia; wyjaśniamy w nich również potrzebę specjalnej uwagi, z jaką należy podechodzić w systemach operacyjnych do obsługi tych urządzeń.

Rozdział 12

SYSTEMY WEJŚCIA-WYJŚCIA

Komputer wykonuje dwa główne zadania: przetwarza informacje i obsługuje działania na wejściu i wyjściu. W wielu sytuacjach podstawowym działaniem są operacje wejścia-wyjścia, a przetwarzanie występuje zaledwie od czasu do czasu. Gdy na przykład przeglądamy strony WWW lub edytujemy plik, interesujemy się głównie czytaniem lub wpisywaniem pewnych informacji – nie polega to na obliczaniu wyników.

Zadaniem systemu operacyjnego w odniesieniu do komputerowych urządzeń wejścia-wyjścia jest zarządzanie operacjami wejścia-wyjścia, sprawowanie nad nimi nadzoru oraz dopilnowywanie działania samych urządzeń zewnętrznych. Pomimo że pokrewne zagadnienia występują w innych rozdziałach, w tym miejscu zebraliśmy je razem, aby odmalować pełny ich obraz. Po pierwsze, opisujemy podstawy sprzętowe wejścia-wyjścia, ponieważ właściwości interfejsu sprzętowego określają wymagania na wewnętrzne cechy systemu operacyjnego. Następnie omawiamy usługi wejścia-wyjścia świadczone przez system operacyjny oraz sposób ich wkomponowania w interfejs wejścia-wyjścia programu użytkowego. Z kolei wyjaśniamy, jak system operacyjny zapewnia lukę między interfejsem sprzętu a interfejsem aplikacji. Na koniec zajmujemy się aspektami wydajności wejścia-wyjścia i podstawami projektowania systemów operacyjnych, które służą jej polepszaniu.

12.1 ■ Przegląd

Sterowanie urządzeniami podłączonymi do komputera leży w sferze podstawowych zainteresowań konstruktorów systemów operacyjnych. Ponieważ

urządzenia wejścia-wyjścia (urządzenia zewnętrzne) różnią się tak bardzo pod względem funkcji i szybkości (zestawmy ze sobą myszkę, dysk twardy i „grającą szafę” z płytami CD-ROM), więc do sterowania nimi potrzeba różnych metod. Metody te tworzą w jądrze *podsystem wejścia-wyjścia* (ang. *I/O subsystem*), który oddziela resztę jądra od złożoności zarządzania wejściem-wyjściem.

W technologii urządzeń wejścia-wyjścia uwidaczniają się dwie przeciwnostawne tendencje. Z jednej strony jesteśmy świadkami rosnącej standaryzacji interfejsów programowych i sprzętowych. Te dążenie pomaga nam wdrażać ulepszone generacje urządzeń w istniejących komputerach i systemach operacyjnych. Z drugiej strony widzimy coraz większą różnorodność urządzeń zewnętrznych. Niektóre nowe urządzenia są tak niepodobne do starych, że włączenie ich do naszych komputerów i systemów operacyjnych stanowi prawdziwe wyzwanie. Odpowiadamy na nie za pomocą łączenia metod sprzętowych i programowych. Podstawowe elementy sprzętowe wejścia-wyjścia, takie jak porty, szyny i sterowniki urządzeń (ang. *device controllers*) występują w wielu urządzeniach zewnętrznych. Aby ogarnąć szczegóły i osobliwości różnych urządzeń, w budowie jądra systemu operacyjnego przewiduje się stosowanie modułów sterujących (ang. *device drivers*). Moduły sterujące tworzą jednolity interfejs dostępu do podsystemu wejścia-wyjścia na podobieństwo odwołań do systemu tworzących interfejs między programem użytkowym a systemem operacyjnym.

W tym rozdziale omawiamy podstawowe mechanizmy sprzętowe służące do wykonywania operacji wejścia-wyjścia oraz sposoby, za pomocą których system operacyjny organizuje urządzenia wejścia-wyjścia według ich kategorii w celu uformowania ogólnego interfejsu wejścia-wyjścia dla aplikacji. Rozpatrujemy mechanizmy jądra tworzące pośrost między sprzętowym wejściem-wyjściem a oprogramowaniem użytkowym, jak również opisujemy struktury, usługi i zagadnienia wydajności podsystemu wejścia-wyjścia.

12.2 ■ Sprzęt wejścia-wyjścia

Komputery korzystają z wielu rodzajów urządzeń. Z grubsza można je podzielić na urządzenia pamięci (dyski, taśmy), urządzenia przesyłania danych (karty sieciowe, modemy) i urządzenia interfejsu z człowiekiem (ekran monitów, klawiatury, myszki). Są jeszcze inne urządzenia, bardziej specjalizowane. Rozważmy sterowanie odrzutowym myśliwcem wojskowym lub statkiem kosmicznym. W tych pojazdach ludzie przekazują dane wejściowe do komputera pokładowego za pomocą manetek, a komputer przesyła polecenia wyjściowe powodujące manewrowanie za pomocą silników sterami, klapami lub odrzutowymi korektorami lotu.

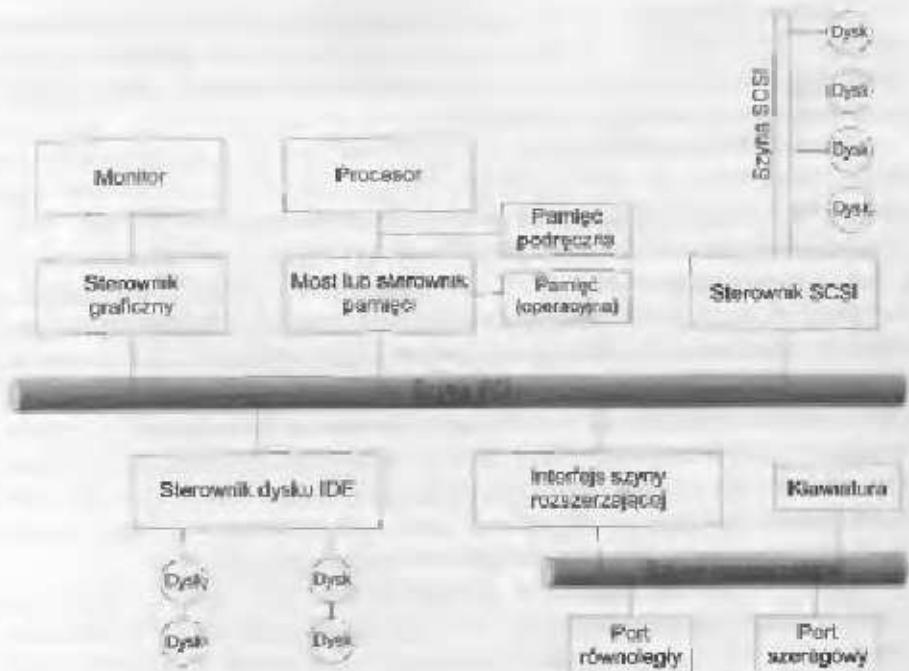
Pomimo niezwykłej różnorodności urządzeń zewnętrznych, które można stosować w komputerze, do zrozumienia sposobu, w jaki urządzenia są do niego podłączane oraz jak oprogramowanie może sterować pracą sprzętu, wystarczy nam kilka zaledwie koncepcji.

Urządzenia komunikują się z systemem komputerowym, przesyłając sygnały przez kable lub nawet metodami bezprzewodowymi. Urządzenie kontaktuje się z maszyną przez punkt łączący nazywany *portem*. Jeśli jedno urządzenie, lub większa ich liczba, korzysta ze wspólnej wiązki przewodów, to połączenie takie nazywa się *szyną*. Określając nieco formalniej, możemy powiedzieć, że szyna jest wiązką przewodów wraz ze ścisłe zdefiniowanym protokołem, przyjmującym zbiór komunikatów, które możliwe tymi przewodami przesyłać. Fizycznie rzecz biorąc, komunikaty są przenoszone za pomocą ciągów impulsów elektrycznych o zmiennym napięciu, pojawiających się w przewodach w określonych chwilach. Jeśli urządzenie A jest połączone kablem z urządzeniem B, a urządzenie B jest podłączone kablem do urządzenia C i na koniec urządzenie C jest podłączone do portu w komputerze, to układ taki zwie się *szeregowym* (ang. *serial*). Zazwyczaj działa on jako szyna.

Szyny są szeroko stosowane w architekturze komputerów. Na rysunku 12.1 jest przedstawiona typowa struktura szyny komputera PC. Widac na nim szynę PCI (typowa szyna komputerów osobistych), która łączy podsystem procesora i pamięci z szybkimi urządzeniami, oraz szynę rozszerzającą, łączącą względnie powolne urządzenia, takie jak klawiatura i porty szeregowe lub równoległe. W prawej górnej części rysunku znajdują się cztery dyski podłączone razem do szyny SCSI, która prowadzi do sterownika SCSI⁴.

Sterownik (ang. *controller*) jest zespołem układów elektronicznych, które mogą kierować pracą portu, szyny lub urządzenia. Sterownik portu szeregowego jest przykładem prostego sterownika urządzenia. Jest to pojedynczy układ scalony, zamontowany w komputerze i nadzorujący sygnały w okablowaniu portu szeregowego. Dla porównania, sterownik szyny SCSI nie jest prosty. Z powodu złożoności protokołu SCSI sterownik szyny SCSI jest często wykonywany w postaci oddzielnej płyty z układami elektronicznymi (adapter główny – ang. *host adapter*), podłączonej do komputera. Na ogół jest on wyposażony w procesor, mikrokod i trochę prywatnej pamięci, co umożliwia mu przetwarzanie komunikatów protokołu SCSI. Niektóre urządzenia mają własne, wbudowane sterowniki. Jeśli spojrzymy na napęd dysku, to zauważymy płytę układów elektronicznych zamocowaną po jednej jego stronie. Ta płyta jest sterownikiem dysku. Realizuje ona dyskową część protokołu

⁴ SCSI, to skrót od angielskich słów *small computer system interface*, czyli interfejs małych systemów komputerowych. – Przyp. tłum.



Rys. 12.1 Budowa szyny typowego komputera osobistego

dla pewnego rodzaju połączenia (np. SCSI lub IDE*). Ma ona mikrokod i procesor do wykonywania wielu zadań, takich jak zaznaczenie złych sektorów, ładowanie wstępne, buforowanie i przechowywanie podręczne.

W jaki sposób procesor przekazuje sterownikowi polecenia i dane potrzebne do wykonania przesłania? Najkrócej mówiąc, sterownik ma w tym celu rejestr (jeden lub więcej) do pamiętania danych i sygnałów sterujących. Procesor komunikuje się ze sterownikiem, czytając i pisząc w tych rejestrach układy bitów. Jedną z możliwości realizacji tej komunikacji są specjalne rozkazy wejścia-wyjścia określające przesłanie bajta lub słowa na adres portu wejścia-wyjścia. Rozkazy wejścia-wyjścia pobudzają linie szyny, umożliwiając wybór odpowiedniego urządzenia oraz przemieszczanie bitów do lub z rejestrów urządzenia. Sterownik urządzenia może też umożliwiać operacje wejścia-wyjścia odwzorowywane w pamięci operacyjnej. W tym przypadku rejesty sterujące urządzenia są odwzorowywane w przestrzeni adresowej procesora.

W niektórych systemach są stosowane obie metody. Na przykład komputery zgodne ze standardem PC używają rozkazów wejścia-wyjścia do ste-

*IDE skrót od integrated drive electronics. – Przyp. tłum.

Przedział adresów wejścia-wyjścia (szesnastkowe)	Urządzenie
000-00F	Sterownik DMA
020-021	Sterownik przerwań
040-043	Czasomierz
200-20F	Sterownik gier
2FB-2FF	Port szeregowy (zapasowy)
320-32F	Sterownik dysku twardego
37B-37F	Port równoległy
3D0-3DF	Sterownik graficzny
3F0-3F7	Sterownik napędu dyskietek
3FB-3FF	Port szeregowy (podstawowy)

Rys. 12.2 Położenie portów wejścia-wyjścia w komputerach zgodnych z PC (fragment)

rowania pewnymi urządzeniami, a do sterowania innymi urządzeniami korzystają z wejścia-wyjścia odwzorowywanego w pamięci. Na rysunku 12.2 są podane adresy typowych portów wejścia-wyjścia komputera PC. Oprócz portów wejścia-wyjścia podstawowych sterownik graficzny ma również duży obszar odwzorowany w pamięci, służący do przechowywania zawartości ekranu. Sterownik ten wysyła dane na ekran przycz zapisywanie ich w odwzorowanym w pamięci obszarze. Obraz na ekranie jest generowany przez sterownika na podstawie zawartości tej części pamięci operacyjnej. Jest to metoda prosta w użyciu. Ponadto zapisywanie milionów bajtów do pamięci obrazu jest szybsze niż wykonywanie milionów rozkazów wejścia-wyjścia. Jednak łatwość zapisywania danych w sterowniku wejścia-wyjścia odwzorowywanym w pamięci jest dystansowana przez wadę. Ze względu na typowy rodzaj błędu programowego, polegającego na zapisaniu wskutek użycia niewłaściwego wskaźnika niepożądanego obszaru pamięci, odwzorowany w pamięci rejestr sterownika jest podatny na przypadkowe zmiany. Oczywiście, stosując pamięć chronioną można zmniejszyć to ryzyko.

Port wejścia-wyjścia składa się na ogół z czterech rejestrów o nazwach: *stan* (ang. *status*), *sterowanie* (ang. *control*), *dane wejściowe* (ang. *data-in*) i *dane wyjściowe* (ang. *data-out*). Rejestr stanu zawiera bity, które mogą być czytane przez procesor główny. Bitы te wskazują takie stany, jak zakończenie bieżącego polecenia, dostępność bajta do czytania w rejestrze danych wejściowych i wykrycie błędu w urządzeniu. Rejestr sterowania może być zapisywany przez procesor główny w celu rozpoczęcia polecenia lub zmiany try-

bu pracy urządzenia. Na przykład pewien bit w rejestrze sterowania portu szeregowego służy do wybierania między komunikacją półdupleksem i pełnodupleksem, inny umożliwia sprawdzanie parzystości, trzeci bit ustala długość słowa na 7 lub 8 bitów, a jeszcze inne bity służą do wybierania jednej z możliwych szybkości pracy portu szeregowego. Rejestr danych wejściowych jest czytany przez procesor główny w celu pobrania informacji z urządzenia, a wysyłanie danych przez procesor główny polega na zapisaniu ich w rejestrze danych wyjściowych. Rejestry danych mają zazwyczaj od 1 do 4 B. Niektóre sterowniki mają układy FIFO, w których przechowuje się po kilka bajtów danych wejściowych lub wyjściowych, aby rozszerzyć pojemność sterownika ponad rozmiar rejestru danych. Układ FIFO może przejmować małe spiętrzenia danych do chwili, w której urządzenie lub komputer jest w stanie dane te odebrać.

12.2.1 Odpytywanie

Pełny protokół konwersacji między procesorem głównym a sterownikiem urządzenia może być zawiły, lecz elementarne pojęcie *uzgodnienia* (ang. *handshaking*) jest proste. Na czym to uzgadnianie polega, wyjaśnimy na przykładzie. Założmy, że do koordynowania zależności producent-konsument między sterownikiem a procesorem są używane 2 bity. Sterownik zaznacza swój stan za pomocą bitu zajętości (ang. *busy*) w rejestrze stanu. (Przypomnijmy, że „ustawić” bit oznacza nadać mu wartość 1, natomiast „wyczyścić” bit znaczy tyle, co zapisać w nim wartość 0). Sterownik ustawia bit zajętości wtedy, kiedy jest zajęty pracą, i czyści go wtedy, kiedy jest gotów do przyjęcia następnego polecenia. Procesor sygnalizuje swoje życzenia za pośrednictwem bitugotowości polecenia (ang. *command-ready*) w rejestrze polecen (ang. *command register*). Procesor ustawia bitgotowości polecenia wówczas, gdy polecenie do wykonania jest dostępne dla sterownika. W tym przykładzie procesor pisze na wyjściu za pośrednictwem portu, koordynując swoje działanie ze sterownikiem za pomocą następującego ciągu uzgodnionów:

1. Procesor główny powtarza czytanie bitu zajętości depoty, dopóki bit ten nie przyjmie wartości 0.
2. Procesor główny ustawia bit pisania (ang. *write bit*) w rejestrze polecen i wpisuje bajt do rejestrów danych wyjściowych.
3. Procesor główny ustawia bitgotowości polecenia.
4. Gdy sterownik zauważa, że bitgotowości polecenia jest ustawiony, wówczas ustawia bit zajętości.

5. Sterownik czyta rejestr poleceń i rozpoznaje polecenie pisania. Czyta więc bajt z rejestru danych wyjściowych i wykonuje na urządzeniu operację wejścia-wyjścia.
6. Sterownik czyści bit gotowości polecenia oraz bit błędu (ang. *error bit*) w rejestrze stanu, aby powiadomić, że operacja wejścia-wyjścia zakończyła się pomyślnie, po czym czyści bit zajętości, sygnaлизując, że zakończyło się działanie.

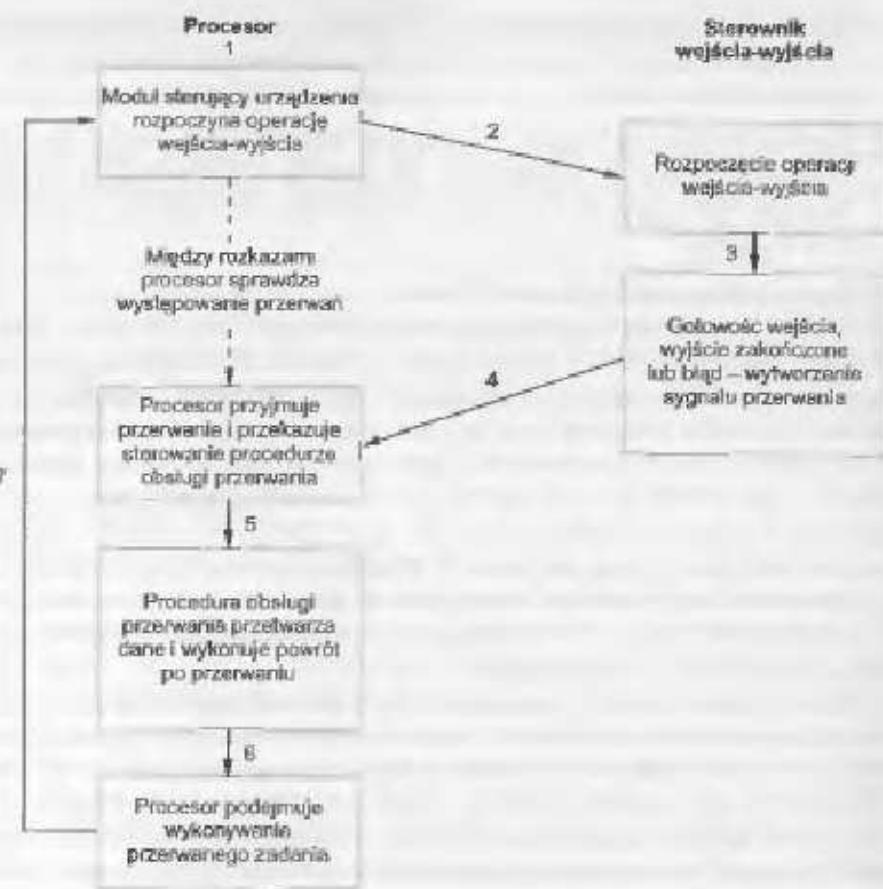
Pętla ta jest powtarzana dla każdego bajtu.

W kroku 1 procesor główny wykonuje *aktynne czekanie* (ang. *busy-waiting*), czyli *odpytywanie* (ang. *polling*) – czyta on nieustannie rejestr stanu, az bit zajętości zostanie wyczyszczony. Jeśli sterownik i urządzenie są szybkie, to metodę tę można uznać za sensowną. Jeśli jednak czekanie miało by być długie, to być może należałoby przełączyć procesor do innego zadania. Jednak w jaki sposób procesor główny pozna wówczas, że sterownik stał się bezczynny? Niektóre urządzenia procesor musi obshigować szybko, w przeciwnym razie dane zostaną utracone. Na przykład strumień danych pojawiający się w porcie szeregowym lub napływających z klawiatury może zapelniać mały bufor sterownika i dane zaczną być tracone, jeżeli procesor zwleka za długo z powrotem do czytania bajtów.

W wielu architekturach komputerów do odpytania urządzenia wystarczą trzy cykle rozkazowe procesora: *czytanie* rejestru urządzenia, operacja *konstrukcji* w celu wydobycia bitu stanu i *skok* przy wartości niezerowej. Jest widoczne, że operacja podstawowego odpytywania jest wydajna. Odpytywanie staje się jednak niewydajne, jeśli jest ponawiane wielokrotnie z powodu rzadkich przypadków gotowości urządzenia do działania, a w tym czasie inne pozytyczne czynności procesora pozostają nie wykonane. W takich przypadkach bardziej wydajna może okazać się taka organizacja działania, w której – zamiast wymagania od procesora ustawniczego odpytywania sterownika o zakończenie operacji wejścia-wyjścia – sterownik sprzęt zawiadamia procesor o powrocie urządzenia do stanu gotowości do działania. Mechanizm sprzętowy umożliwiający urządzeniu zawiadamianie procesora nazywa się *przerwaniem* (ang. *interrupt*).

12.2.2 Przerwania

Podstawowy mechanizm przerwania działa następująco. Osprzęt procesora ma sciejkę nazywaną *linią zgłoszenia przerwania* (ang. *interrupt request line*), którą procesor bada po wykonaniu każdego rozkazu. Jeśli procesor wykryje, że sterownik zasygnalizował coś w linii zgłoszenia przerwania, to przechowa niewielką ilość danych określających stan, takich jak bieżąca wartość licznika



Rys. 12.3 Cykl wejścia-wyjścia obsługiwany za pomocą przerwania

rozkazów, i wykona skok pod ustalony adres w pamięci do *procedury obsługi przerwania* (ang. *interrupt-handler*). Procedura obsługi przerwania rozpoznaje przyczynę przerwania, wykonuje niezbędną czynność, a po nich *rozkaz powrotu z przerwania*, mający na celu przywrócenie procesora do stanu wykonywania sprzed przerwania. Mówimy, że sterownik urządzenia zgłasza przerwanie za pomocą sygnału podawanego w linii zgłoszenia przerwania, procesor przechwytuje przerwanie i wysyła je do procedury obsługi przerwania, a ta czyści je, obsługując urządzenie. Na rysunku 12.3 są pokazane elementy cyklu wejścia-wyjścia obsługiwanej za pomocą przerwania.

Podstawowy mechanizm przerwania umożliwia procesorowi reagowanie na zdarzenia asynchroniczne, takie jak wystąpienie w sterowniku urządzenia gotowości do pracy. W nowoczesnym systemie operacyjnym są potrzebne bar-

dziej złożone możliwości obsługiwanego przerwań. Po pierwsze, potrzebujemy możliwości opóźniania obsługi przerwania wówczas, gdy są wykonywane działania krytyczne. Po drugie, potrzebujemy skutecznego sposobu kierowania przerwania do właściwej dla danego urządzenia procedury obsługi bez uprzedniego odpytywania wszystkich urządzeń w celu wykrycia, które z nich zgłoszą przerwanie. Po trzecie, potrzebujemy przerwań wielopoziomowych, aby system operacyjny mógł odróżnić przerwania niskopriorytetowe od wysokopriorytetowych i reagować z odpowiednim stopniem pilności. W nowoczesnym sprzęcie komputerowym te trzy właściwości są zapewniane przez jednostkę centralną oraz *sprzęt sterownika przerwań* (ang. *interrupt controller*).

Większość procesorów ma dwie linie zgłoszenia przerwań. Jedną z nich docierają *przerwania niemaskowalne*, zarezerwowane dla zdarzeń takich jak nieusuwalne błędy pamięci. Druga linia przerwań jest maskowalna – można ją wyłączyć za pomocą procesora przed wykonaniem krytycznych ciągów instrukcji, których nie wolno przerywać. *Przerwania maskowalne* są używane przez sterowniki urządzeń do zgłoszenia żądań obsługi.

Mechanizm przerwania przyjmuje adres, czyli liczbę, która służy do wyboru konkretnej procedury obsługi przerwania, z pewnego malego zbioru. W większości architektur adres ten oznacza odległość w tablicy nazywanej *wektorem przerwań* (ang. *interrupt vector*). Wektor przerwań zawiera adresy pamięci wyspecjalizowanych procedur obsługi przerwań. Celem mechanizmu wektorowego jest zredukowanie konieczności przeszukiwania przez poszczególne procedury obsługi przerwań wszystkich możliwych źródeł przerwań, aby rozstrzygnąć, które wymaga obsługi. Jednak w praktyce komputery mają więcej urządzeń (a więc i procedur obsługi przerwań) aniżeli pozycji adresowych w wektorze przerwań. Popularnym sposobem rozwiązywania tego problemu jest zastosowanie techniki łańcuchowania przerwań, w której każdy element wektora przerwań wskazuje na czoło listy procedur obsługi przerwań. Po zgłoszeniu przerwania procedury obsługuje z odpowiedniej listy są wywoływanie jedna po drugiej, aż zostanie odnaleziona ta, która potrafi obsłużyć przerwanie. Struktura taka jest kompromisem między kosztem wielkiej tablicy przerwań a nieefektywnością kierowania przerwania do jednej procedury obsługi.

Rysunek 12.4 ilustruje rozwiązanie wektora przerwań przyjęte w procesorze Pentium firmy Intel. Pierwsze 32 niemaskowalne zdarzenia sygnalizują rozmajne sytuacje błędne. Pozycje z przedziału od 32 do 255 są maskowalne i używane do sygnalizowania zdarzeń takich, jak przerwania generowane przez urządzenia.

W skład mechanizmu przerwań wchodzi też *system poziomów priorytetów przerwań* (ang. *interrupt priority levels*). Umozliwia on opóźnianie przez procesor obsługi przerwań niskopriorytetowych bez maskowania wszystkich przerwań i pozwala przerwaniom wysokopriorytetowym wywijażać procedury obsługi przerwań niskopriorytetowych.

Numer wektora	Oznaczenie
0	błąd dzielenia
1	wyjątek diagnostyczny
2	przerwanie pustego
3	punkt kontrolny
4	nadmiar INT0 wykryty
5	wyjątek granicy przedziału
6	niedozwolony kod operacji
7	urządzenie niedostępne
8	błąd podwożenia
9	przepelnienie segmentu koprocesora (zarezerwowane)
10	niedozwolony segment stanu zadania
11	brak segmentu
12	błąd atosu
13	ochrona ogólna
14	błąd strony
15	(zarezerwowane przez firmę Intel – nie używać)
16	błąd zmiennopozycyjny
17	kontrola przylogania
18	kontrola maszyny
19-31	(zarezerwowane przez firmę Intel – nie używać)
32-255	przeniesienie maskowalne

Rys. 12.4 Tablica wektora zdarzeń procesora Intel Pentium

Nowoczesny system operacyjny współpracuje z mechanizmem obsługi przerwań na kilka sposobów. Podczas rozruchu system operacyjny sprawdza szyny sprzętowe w celu określenia, które urządzenia są dostępne, i instalując w wektorze przerwań odpowiednie procedury ich obsługi. Podeczas wykonywania operacji wejścia-wyjścia przerwania są generowane przez rozmaito sterowniki urządzeń, zgłaszające swoją gotowość do pracy. Przerwania te oznaczają zakończenie operacji wyjścia lub dostępność danych wejściowych, albo też wykrycie jakiegoś błędu. Mechanizm przeprowadzania stosuje się również do obsługi różnych sytuacji wyjątkowych (ang. exceptions), takich jak próba dzielenia przez zero, zaadresowanie chronionego lub nie istniejącego obszaru pamięci lub usiłowanie wykonania przez użytkownika rozkazu uprzywilejowanego. Zdarzenia powodujące przerwania muszą oznaczać się wspólną cechą: ich wystąpienie zmusza procesor do wykonania pilnej, samowystarczalnej procedury.

Wydajny, sprzętowy mechanizm przechowywania niewielkich ilości informacji o stanie procesora, połączony z wywołaniem uprzywilejowanej procedury jądra, może znaleźć w systemie operacyjnym jeszcze inne, pozyteczne zastosowania. Na przykład wiele systemów operacyjnych korzysta z mechanizmu przerwania przy stronnicowaniu pamięci wirtualnej. Brak strony jest wyjątkiem powodującym zgłoszenie przerwania. Przerwanie to wstrzymuje

bieżący proces i powoduje skok do procedury obsługi braku strony w jądrze. Procedura taka przechowuje stan procesu, przenosi proces do kolejki oczekujących, wykonuje operacje administracyjne na pamięci podręcznej stron, zaplanowuje operację wejścia-wyjścia w celu pobrania strony, wyznacza inny proces do wznowienia działania, po czym kontynuuje obsługę przerwania.

Inny przykład odnajdujemy w realizacji odwołań do systemu. *Odwółanie do systemu* (ang. *system call*) jest funkcją, którą program użytkowy wywołuje w celu przywołania usługi jądra. W odwołaniu do systemu sprawdza się argumenty podane przez program użytkowy, tworzy strukturę danych do przesunięcia ich do jądra, po czym wykonuje specjalny rozkaz, nazywany *przerwaniem programowym* (ang. *software interrupt*) lub *pętlą* (ang. *trap*^{*}). Ten rozkaz na argument identyfikujący wymaganą usługę jądra. Wykonanie rozkazu pętli podczas odwoływanego się do systemu powoduje przechowanie przez sprzęt obsługi przerwań stanu programu użytkownika, przełączenie procesora w tryb nadzorcę i przejście do wykonania procedury jądra, realizującej zamawianą usługę. Pętla ma stosunkowo niski priorytet przerwania w porównaniu z priorytetami przydzielonymi przerwaniom pochodząącym od urządzeń – wykonanie odwołania do systemu na żądanie aplikacji jest mniej pilne niż obsługa sterownika urządzenia, zanim kolejka FIFO tego ostatniego przepchnie się i nastąpi utrata danych.

Przerwania mogą być też stosowane do zarządzania przebiegiem sterowania w jądrze. Przeanalizujmy na przykład działania wymagane do zakończenia operacji czytania dysku. Jeden krok polega na przekopiowaniu danych z obszaru jądra do bufora użytkownika. Kopiowanie to zajmuje dużo czasu, lecz nie jest pilne – nie powinno tamować obsługi innych przerwań o wysokich priorytetach. W innym kroku należy rozpoczęć następną oczekującą operację wejścia-wyjścia, odnoszącą się do danego dysku. Ten krok ma wyższy priorytet. Jeżeli dysk ma być eksploatowany wydajnie, to należy rozpoczęć następną operację wejścia-wyjścia zaraz po zakończeniu poprzedniej. W efekcie kod jądra kończący czytanie z dysku jest implementowany za pomocą pary procedur obsługujących przerwania. Procedura wysokopriorytetowa odnotowuje stan wejścia-wyjścia, czyści przerwanie pochodzące od urządzenia, rozpoczyna następną ciekającą operację wejścia-wyjścia i w celu dokonania pracy zgłasza przerwanie niskopriorytetowe. Później, gdy procesor nie będzie zajęty pilną pracą, procedura obsługi niskopriorytetowego przerwania zostanie do niego skierowana. Odpowiednia procedura obsługi dokonczy operację wejścia-wyjścia z poziomu użytkownika, kopując dane z buforów jądra do obszaru aplikacji i wywołując planistę przydziału procesora w celu umieszczenia aplikacji w kolejce procesów gotowych do wykonywania.

* Aby dojść do systemu – Przyp. tłum.

Architektura jądra z wątkami dobrze nadaje się do implementowania wielu priorytetów przerwań oraz do narzucania kolejności obsługiwanego przerwań ponad przetwarzaniem drugoplanowym w jądrze i procedurami użytkowymi. Zilustrujemy to za pomocą jądra systemu Solaris. W systemie Solaris procedury obsługi przerwań są wykonywane jako wątki jądra. Dla wątków tego rodzaju jest zarezerwowany przedział wysokich priorytetów. Dzięki tym priorytetom procedury obsługi przerwań zyskują pierwszeństwo przed programami użytkowymi i jądrowymi procedurami administracyjnymi oraz realizują zależności priorytetowe między sobą. Priorytety umożliwiają planowanie wątków systemu Solaris wywłaszczenie niskopriorytetowych procedur obsługi przerwań na rzecz procedur wysokopriorytetowych. Implementacja procedur obsługi przerwań za pomocą wątków umożliwia ponadto wspólnocne wykonywanie wielu tych procedur na sprzecie wieloprocesorowym. Architekturę przerwań w systemach UNIX i Windows NT omawiamy w rozdz. 21 i 23 (odpowiednio).

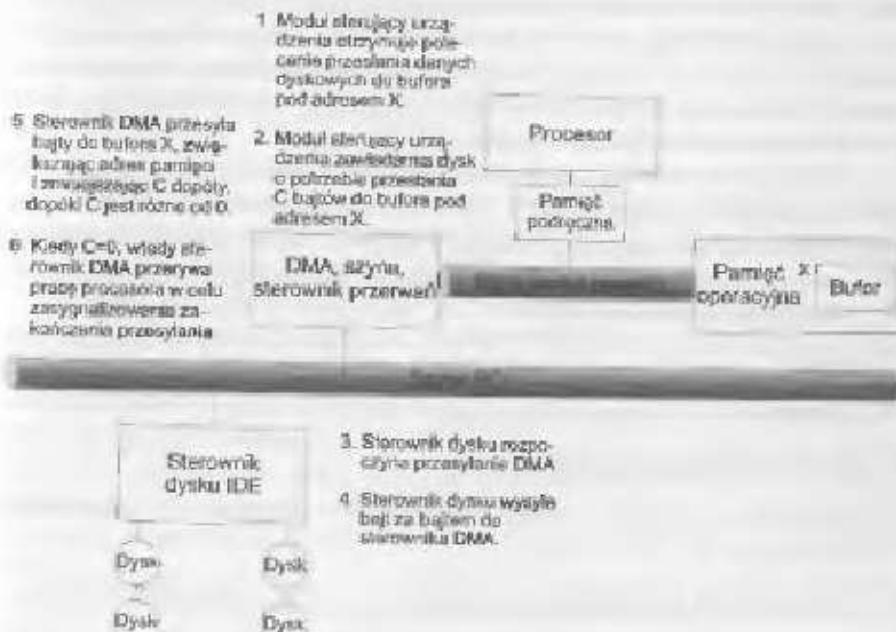
Podsumowując, możemy stwierdzić, że przerwania są stosowane we wszystkich współczesnych systemach operacyjnych, gdzie służą do obsługi asynchronicznych zdarzeń oraz jako środek przechodzenia do procedur wykonywanych w jądrze, w trybie nadzorcy. Aby umożliwić wykonywanie prac najpiękniejszych w pierwszej kolejności, w nowoczesnych systemach operacyjnych stosuje się systemy priorytetów przerwań. Sterowniki urządzeń, błędy sprzętowe oraz odwołania do systemu – wszystkie one powodują przerwania prowadzące do procedur jądra. Ponieważ przerwania są tak intensywnie używane w przetwarzaniu, w którym istotnym czynnikiem jest czas, dobra wydajność systemu jest zależna od sprawnego ich obsługiwanego.

12.2.3 Bezpośredni dostęp do pamięci

W przypadku urządzenia transmitującego wielkie ilości informacji, jak na przykład napęd dysku, zatrudnianie drogiego procesora ogólnego przeznaczenia do obserwowania bitów stanu i przekazywanie danych sterownikowi po jednym bajcie – które to postępowanie nosi nazwę *programowanego wejścia-wyjścia* (ang. *programmed I/O* – PIO) – wydaje się marnotrawstwem. Wielu komputerów, aby nie zakłócać działania głównego procesora stosowaniem metody PIO, przenosi część tej pracy na wyspecjalizowany procesor, nazywany *sterownikiem bezpośredniego dostępu do pamięci* (ang. *direct memory access* – DMA). W celu rozpoczęcia przesyłania w trybie DMA procesor główny zapisuje w pamięci blok sterujący DMA. Blok ten zawiera wskaźnik do źródła przesyłania, wskaźnik miejsca docelowego przesyłania oraz liczbę bajtów do przesłania. Jednostka centralna zapisuje adres bloku sterującego w sterowniku DMA i przechodzi do kontynuowania innych prac. Sterownik

DMA przejmuje wówczas bezpośredni nadzór nad szyną pamięci, umieszczając w niej adresy w celu wykonania przesyłania bez pomocy jednostki centralnej. Prosty sterownik DMA jest standardowym wyposażeniem komputerów osobistych PC, a płyty wejścia-wyjścia komputerów PC służące do zarządzania szyną (ang. *bus-mastering*) zazwyczaj zawierają własny sprzęt DMA o dużej szybkości.

Uzgadnianie przesyłania między sterownikiem DMA a sterownikiem urządzenia dokonuje się za pośrednictwem pary przewodów nazywanych *zamówieniem DMA* (ang. *DMA request*) i *potwierdzeniem DMA* (ang. *DMA acknowledge*). Sterownik urządzenia wytwarza sygnał w przewodzie zamówienia DMA, gdy słowo danych jest gotowe do przesłania. Ten sygnał powoduje, że sterownik DMA przejmuje szynę pamięci, aby umieścić potrzebny adres w liniach adresowych pamięci oraz wytworzyć sygnał w przewodzie potwierdzenia DMA. Gdy sterownik urządzenia odbierze sygnał potwierdzenia DMA, wówczas przesyła słowo danych do pamięci i usuwa sygnał zamówienia DMA. Po zakończeniu całego przesyłania sterownik DMA generuje przerwanie w jednostce centralnej. Proces ten jest przedstawiony na rys. 12.5. Zauważmy, że z chwilą przejęcia przez sterownik DMA szyny pamięci następuje natychmiastowa ochrona pamięci operacyjnej przed dostępem do niej.



Rys. 12.5 Etapy przesyłania w trybie DMA

procesora, choć procesor nadal może sięgać po obiekty danych w jego podstawowej i pomocniczej pamięci podręcznej. Choć taka kradzież cykli (ang. *cycle stealing*) może spowolnić obliczenia procesora, wyzbycie się przesyłania danych na rzecz sterownika DMA z reguły polepsza ogólną wydajność systemu. W niektórych architekturach komputerów oddaje się do dyspozycji trybu DMA adresy pamięci fizycznej, natomiast w innych stosuje się *bezpośredni dostęp do pamięci wirtualnej* (ang. *direct virtual memory access* – DVMA) z wykorzystaniem adresów wirtualnych, które są poddawane tłumaczeniu na adresy pamięci fizycznej. W trybie DVMA można wykonywać przesłania między dwoma urządzeniami odwzorowanymi w pamięci, nie angażując jednostki centralnej i nie używając pamięci operacyjnej.

W jądrach z trybem chronionym system operacyjny z zasady nie dopuszcza, aby procesy bezpośrednio wydawały polecenia urządzeniom. Takie postępowanie chroni dane przed naruszeniem kontroli dostępu i zapobiega bezsensownemu użyciu sterowników urządzeń, które mogliby spowodować załamanie systemu. Zamiast tego system operacyjny udostępnia funkcje, które mogą być używane przez wystarczająco uprzywilejowane procesy do dostępu do niskopoziomowych operacji sprzętowych. W jądrach bez ochrony pamięci procesy mogą kontaktować się ze sterownikami urządzeń bezpośrednio. Ów bezpośredni dostęp umożliwia uzyskiwanie dużej wydajności, gdyż unika się wtedy komunikacji z jądrem, przełączania kontekstu i warstw oprogramowania w jądrze. Nieśćety, pozostaje on w kolizji z wymogami bezpieczeństwa i stabilności systemu. W systemach operacyjnych ogólnego przeznaczenia istnieje tendencja do ochrony pamięci i urządzeń, aby system mógł strzec przed błędnymi lub złośliwymi programami.

Pomimo złożoności sprzętowych aspektów wejścia-wyjścia, rozpatrywanych na poziomie szczegółowości interesującym projektantów sprzętu elektronicznego, opisane przez nas do tej pory pojęcia wystarczają do zrozumienia wielu aspektów systemowych związanych z obsługą wejścia-wyjścia. Przyponijmy jeszcze główne pojęcia:

- szyna;
- sterownik;
- port wejścia-wyjścia i jego rejstry;
- uzgadnianie działania między procesorem głównym a sterownikiem urządzenia;
- egzekwowanie tych uzgodnień za pomocą pętli odpytywania lub przerwań;
- zrzucanie tych obowiązków na sterownik DMA w przypadku przesyłania wielkich ilości informacji.

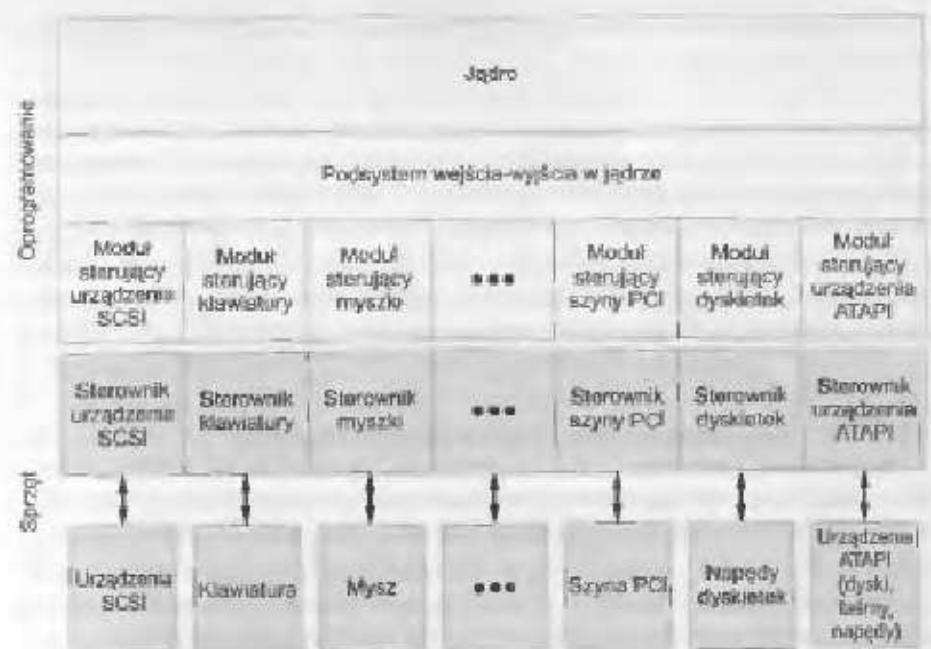
W poprzednim przykładzie pokazaliśmy uzgadnianie działań między sterownikiem urządzenia a procesorem głównym. W rzeczywistości wielka różnorodność dostępnych urządzeń przysparza kłopotów realizatorom systemów operacyjnych. Każdy rodzaj urządzenia ma indywidualny zbiór właściwości, własne definicje bitów sterujących i protokołu współpracy z procesorem głównym, przy czym wszystkie te cechy są za każdym razem inne. W jaki sposób skonstruować system operacyjny, aby można było dołączać do komputera nowe urządzenia bez przepisywania samego systemu? Skoro urządzenia różnią się tak bardzo, jak wobec tego system operacyjny ma tworzyć wygodny, jednolity interfejs wejścia-wyjścia dla swoich aplikacji?

12.3 ■ Użytkowy interfejs wejścia-wyjścia

W tym punkcie omawiamy metody strukturalizacji oraz interfejsy systemu operacyjnego umożliwiające standardowe, jednolite traktowanie urządzeń wejścia-wyjścia. Wyjaśniamy na przykład, w jaki sposób aplikacja może otworzyć plik na dysku, nie znając jego typu, oraz jak można dołączać do komputera nowe rodzaje dysków i innych urządzeń bez naruszania systemu operacyjnego.

Podobnie jak w innych złożonych zagadnieniach inżynierii oprogramowania, korzysta się tu z abstrahowania, obudowywania i uwarstwiania oprogramowania. W szczególności możemy zaniedbać detale różnice między sobą urządzenia wejścia-wyjścia, wydzielając kilka ogólnych ich rodzajów. Dostęp do każdego z tych ogólnych rodzajów urządzeń odbywa się za pomocą ustandardyzowanego zbioru funkcji, czyli *interfejsu*. Rzeczywiste różnice są zamknięte w modułach jądra nazywanych *modułami sterującymi* (programami obsługi urządzeń), które są wewnętrznie dostosowane do każdego z urządzeń, natomiast zas z dostępu mają pewien interfejs standardowy. Na rysunku 12.6 widać, w jaki sposób części oprogramowania dotyczące wejścia-wyjścia układają się w warstwy.

Warstwa modułów sterujących ma za zadanie ukrywać różnice między sterownikami urządzeń przed podsystemem wejścia-wyjścia w jądrze, podobnie jak systemowe wywołania wejścia-wyjścia obudowują specyficzne działania urządzeń w kilku ogólnych klasach, ukrywających różnice sprzętowe przed aplikacjami. Unieczalnienie podsystemu wejścia-wyjścia od sprzętu upraszcza zadanie konstruktorom systemu operacyjnego. Jest ono również na ręce producentom sprzętu. Projektyowane przez nich urządzenia są zgodne z istniejącym w komputerze interfejsem sterowników (np. takim jak SCSI-2) albo są zaopatrywane przez producentów w programowe moduły sterujące tworzące interfejs między nowym sprzętem a popularnymi systemami operacyjnymi. W ten sposób nowe urządzenia zewnętrzne mogą być dołączane do komputera bez czekania, a dostawca systemu operacyjnego zaopatrzy go



Rys. 12.6 Struktura uprogramowania wejścia-wyjścia w jądrze

w odpowiednie możliwości. Niestety, producenci sprzętu stają w obliczu różnorodności standardów interfejsów modułów sterujących, obowiązujących w poszczególnych systemach operacyjnych. Dane urządzenie musi być dostarczane z wieloma modułami sterującymi, na przykład dla systemów: MS-DOS, Windows 95, Windows NT i Solaris.

Urządzenia różnią się pod wieloma względami, co widać na rys. 12.7.

- **Strumień znaków lub bloki:** Urządzenie znakowe (ang. *character-stream device*) przesyła bajty z osobna, jeden po drugim, natomiast urządzenie blokowe przesyła jednorazowo cały blok bajtów.
- **Dostęp sekwencyjny lub swobodny:** Urządzenie o dostępie sekwencyjnym przesyła dane w sposób uporządkowany, zależny od urządzenia, natomiast użytkownik urządzenia o dostępie swobodnym może nakazać urządzeniu odrajdywanie danych w dowolnym dostępnych miejscu ich przechowywania.
- **Synchroniczność lub asynchroniczność:** Urządzenie synchroniczne przesyła dane w przewidywalnym z góry czasie. Urządzenie asynchroniczne ma nieregularne lub nieprzewidywalne czasy odpowiedzi.

Cechy	Odmiany	Przykład
Tryb przesyłania danych	znakowy blokowy	terminal dysk
Sposób dostępu	sekwenncyjny swobodny	modem CD-ROM
Organizacja przesyłania	synchroniczna asynchroniczna	taśma klawiatura
Dzielenie	na zasadzie wyłączności użytkowania wspólnego	taśma dysk
Szybkość urządzenia	zwiększenie powodowane rotacją nośnika danych czas odczytywania danych czas przesyłania opóźnienie między operacjami	
Kierunek przesyłania	tylko czytanie tylko pisanie czytanie i pisanie	CD-ROM slownik graficzny dysk

Rys. 12.7 Właściwości urządzeń wejścia-wyjścia

- Dzielenie lub wyłączność:** Urządzenie dzielone może być używane współbieżnie przez wiele procesów lub wątków. Urządzenie działające na zasadzie wyłączności (dedykowane) nie może tak pracować.
- Szybkość działania:** Szybkości urządzeń wahają się w przedziale od kilku bajtów na sekundę do kilku gigabajtów na sekundę.
- Czytanie i pisanie, tylko czytanie, tylko pisanie:** Niektóre urządzenia wykonują zarówno operacje wejścia, jak i wyjścia; inne umożliwiają przepływ danych tylko w jednym kierunku.

Z uwagi na w programy użytkowe wiele spośród tych różnic jest ukrywanych przez system operacyjny, a urządzenia grupuje się w kilka konwencjonalnych typów. Wynikający z tego sposób dostępu do urządzeń znajduje powszechną akceptację i jest szeroko stosowany. Chociaż same odwołania do systemu mogą być różne w różnych systemach, typy urządzeń można uważać za w miarę ustandardyzowane. Do podstawowych konwencji dostępu należą: wejście-wyjście blokowe, wejście-wyjście znakowe (strumień znaków), dostęp do plików odwzorowywany w pamięci oraz gniazda sieciowe. Systemy operacyjne zawierają również specjalne funkcje systemowe, umożliwiające dostęp do niewielkiej liczby urządzeń dodatkowych, takich jak zegar czasu dobowego i czasomierz. Niektóre systemy operacyjne dostarczają zbioru wywołań do obsługi monitorów graficznych oraz urządzeń wizualnych i dźwiękowych.

W większości systemów operacyjnych występuje także *system obejścia* (ang. *escape system*) lub „*bocznymi drzwiami*” (ang. *back door*), który w sposób przezroczysty przekazuje dowolne polecenia od aplikacji do modułu sterującego. W systemie UNIX wywołanie takie nosi nazwę *ioctl* (z ang. *I/O control*, czyli sterowanie wejściem-wyjściem). Funkcja systemowa *ioctl* umożliwia aplikacji dostęp do dowolnej możliwości, która może być zaimplementowana przez jakikolwiek moduł sterujący, bez konieczności obmyślania nowego odwołania do systemu. Wywołanie systemowe *ioctl* ma trzy argumenty. Pierwszym jest deskryptor pliku łączący aplikację z modelem sterującym przez odniesienie do urządzenia sprzętowego zarządzanego przez ten moduł. Drugim argumentem funkcji *ioctl* jest liczba całkowita, za pomocą której wybiera się jedno z poleceń zrealizowanych w module sterującym. Trzeci argument jest wskaźnikiem. Może on wskazywać dowolną strukturę danych w pamięci, umożliwiając za jej pośrednictwem przekazywanie między aplikacją a modelem sterującym dowolnych informacji sterujących lub danych.

12.3.1 Urządzenia blokowe i znakowe

Interfejs urządzenia blokowego (ang. *block device*) obejmuje wszystkie aspekty niezbędne przy dostępie do napędów dyskowych i innych urządzeń o działaniu blokowym. Oczekuje się, że urządzenie będzie rozpoznawać takie polecenia, jak czytaj (ang. *read*) i pisz (ang. *write*) i – jeśli jest to urządzenie o dostępie bezpośrednim – również polecanie szukaj (ang. *seek*) umożliwiające określenie, który blok ma być przesłany w następnej kolejności. Dostęp do takiego urządzenia odbywa się w aplikacjach zazwyczaj za pomocą interfejsu systemu plików. System operacyjny oraz aplikacje specjalne, jak systemy zarządzania bazami danych, mogą preferować dostęp do urządzenia blokowego traktowanego jak zwykła, liniowa tablica bloków. Ten tryb dostępu jest czasami nazywany *surowym wejściem-wyjściem* (ang. *raw I/O*). Jak można zauważyć, operacje czytania, pisania i szukania obejmują istotne cechy działania blokowych urządzeń pamięci, tak więc aplikacje są odizolowane od niskopoziomowych różnic między urządzeniami tego rodzaju.

Dostęp do plików *odwzorowywanych w pamięci* (ang. *memory-mapped*) może tworzyć warstwę powyżej modułów sterujących urządzeń blokowych. Zamiast udostępniać operacje czytania i pisania, w interfejsie odwzorowanych w pamięci umożliwia się dostęp do pamięci dyskowej za pośrednictwem tablicy bajtów w pamięci głównej. Wywołanie systemowe, które odwzorowuje plik w pamięci, powoduje przekazanie adresu wirtualnego tablicy znaków zawierającej kopię pliku. Rzeczywiście przesyłanie danych zachodzi tylko wtedy, kiedy trzeba zapewnić dostęp do obrazu pamięci. Wejście-wyjście odwzorowywane w pamięci jest sprawne, gdyż przesyłanie odbywa się za

pomocą tego samego mechanizmu co dostęp do pamięci wirtualnej stronionowej na żądanie. Odwzorowywanie informacji w pamięci jest także wygodne dla osób programujących, ponieważ dostęp do pliku odwzorowanego w pamięci jest równie prosty jak czytanie i zapisywanie pamięci operacyjnej. W systemach operacyjnych z pamięcią wirtualną często spotyka się zastosowanie interfejsu odwzorowań do realizowania usług jądra. Aby na przykład wykonać program, system operacyjny odwzorowuje plik wykonywalny w pamięci, po czym przekazuje sterowanie do adresu wejściowego w tym odwzorowaniu. Interfejs odwzorowań jest także często stosowany wówczas, gdy jądro dokonuje wymiany obszaru z użyciem dysku.

Przykładem urządzenia dostępnego za pomocą interfejsu strumienia znaków jest klawiatura. Podstawowe odwołanie do systemu w interfejsie tego rodzaju umożliwia aplikacji pobranie (ang. *get*) lub przekazanie (ang. *put*) jednego znaku. Powyżej tego interfejsu oprogramowanie biblioteczne może tworzyć dostęp na zasadzie jednorazowego przesyłania całych wierszy tekstu z możliwością ich buforowania i edytowania (np. naciśnięcie przez użytkownika klawisza `backspace` powoduje usunięcie ze strumienia wejściowego poprzedniego znaku). Ten rodzaj dostępu jest wygodny w przypadku takich urządzeń wejściowych, jak klawiatury, myszki i modemy, w których dane wejściowe powstają „spontanicznie”, tj. w chwilach niekoniecznie możliwych do określenia przez aplikację. Jest to również dobry rodzaj dostępu do urządzeń wyjściowych, jak drukarki lub karty dźwiękowe, które w sposób naturalny pasują do koncepcji liniowego strumienia bajtów.

12.3.2 Urządzenia sieciowe

Pod względem wydajności i właściwości adresowania sieciowe wejście-wyjście znacznie różni się od wejście-wyjścia dyskowego, toteż większość systemów operacyjnych udostępnia sieciowy interfejs wejścia-wyjścia, odmienny od interfejsu `czytaj-pisz-szukaj` odnoszącego się do dysków. W wielu systemach operacyjnych, w tym w systemach UNIX i Windows NT, jest dostępny interfejs gniazda sieciowego.

Pomyślimy o ściennym gniazdku elektrycznym. Można do niego podłączyć dowolne urządzenie elektryczne. Analogicznie, odwołania do systemu w interfejsie gniazda umożliwiają aplikacji utworzenie gniazda, połączenie lokalnego gniazda ze zdalnym adresem (łącząc aplikację z gniazdem utworzonym przez inną aplikację), nashuciwanie, czy któraś ze zdalnych aplikacji podłączyła się do gniazda lokalnego, oraz wysyłanie i odbieranie pakietów za pomocą tego połączenia. Aby ułatwić implementowanie serwerów, interfejs gniazda zawiera też funkcję nazywaną `wybierz` (ang. `select`) zarządzającą zbiorem gniazd. Wywołanie funkcji `wybierz` powoduje przekazanie informacji

o tym, które z gniazda mają nie odebrane pakiety, a które mają miejsce, aby przyjąć nowy pakiet. Posłużenie się funkcją *wybierz* uwalnia od konieczności odpisywania i aktywnego czekania, które w przeciwnym razie należałoby stosować w sieciowym wejściu-wyjściu. Te funkcje obudowują istotne aspekty działania sieci, znacznie ułatwiając opracowywanie rozproszonych aplikacji, w których można zastosować dowolny sprzęt sieciowy i stos protokołów.

Urzeczywiastniono wiele innych podejść do komunikacji międzyprocesowej i komunikacji sieciowej. Na przykład system Windows NT rozporządza interfejsem do kontaktowania się z kartą interfejsu sieciowego oraz interfejsem do protokołów sieciowych (zob. p. 23.6). W systemie UNIX, który z racji swojej długiej historii przyczynił się do wypróbowania podstawowych technologii sieciowych, odnajdujemy półdupleksowe potoki, pełnodupleksowe kolejki FIFO, pełnodupleksowe strumienie, kolejki komunikatów i gniazda. Informacje dotyczące sieciowych aspektów systemu UNIX podajemy w p. 21.9.

12.3.3 Zegary i czasomierze

Większość komputerów jest zaopatrzonych w sprzętowe zegary i czasomierze, spełniające trzy podstawowe funkcje:

- podawanie bieżącego czasu,
- podawanie upływającego czasu,
- powodowanie przez odpowiednio nastawiony czasomierz wykonania operacji X w chwili T .

Funkcje te są intensywnie używane przez system operacyjny oraz w aplikacjach uzależnionych od czasu. Niestety, wywołania systemowe realizujące te funkcje nie są ustandaryzowane w systemach operacyjnych.

Sprzęt służący do pomiaru upływającego czasu oraz do powodowania wykonywania operacji nazywa się *czasomierzem programowalnym* (ang. *programmable Interval timer*). Można go nastawić na czekanie przez pewien okres, po upływie którego powoduje on przerwanie. Czasomierz można nastawić na jednorazowe spowodowanie tej czynności lub na jej powtarzanie i okresowe generowanie przerwań. Z mechanizmu czasomierza korzysta planista przydziału procesora, generując przerwania wywłaszczające proces na końcu przyznanego mu kwantu czasu. Korzysta z niego podsystem dyskowego wejścia-wyjścia, aby powodować okresowe opróżnianie na dysk zabrudzonych buforów podręcznych, a także podsystem sieciowy w celu kasowania operacji, które są wykonywane za wolno wskutek przeładowania sieci lub awarii. System operacyjny może również udostępniać interfejs czasomierza

procesem użytkowym. Symulując zegary wirtualne, system operacyjny może obsługiwać więcej zamówień dotyczących pomiaru czasu niż wynosi liczba posiadanych przez niego kanałów w sprzętowym wyposażeniu czasomierza. W tym celu jądro (lub moduł sterujący czasomierzem) utrzymuje wykaz przerwań zgłoszonych przez jego własne procedury i zamówienia użytkowników, uporządkowany w kolejności od najwcześniejszego. Jądro nastawia czasomierz na najwcześniejszy termin z wykazu. Gdy wystąpi przerwanie, jądro sygnalizuje ten fakt zamawiającemu i przeladowuje rejestr czasomierza na następny, najbliższy termin.

W wielu komputerach częstotliwość przerwań generowanych przez tylkinię zegara sprzętowego wynosi od 18 do 60 impulsów na sekundę. Jest to słaba rozdzielcość, zważywszy że nowoczesny komputer może wykonywać setki milionów rozkazów na sekundę. Dokładność wyzwalaczy operacji jest ograniczona przez tę słabą rozdzielcość czasomierza, do czego dochodzą nakłady związane z utrzymywaniem zegarów wirtualnych. Jeżeli więc impulsy czasomierza są stosowane do utrzymywania zegara czasu dobowego w systemie, to zegar systemowy może się spóźniać lub spieszyć. W większości komputerów zegar sprzętowy jest zbudowany z licznika wysokiej częstotliwości. W niektórych komputerach wartość tego licznika można odczytywać z rejestru sprzętowego, a wówczas licznik taki może być uważany za zegar o wysokiej rozdzielcości. Choć zegar ten nie generuje przerwań, można za jego pomocą dokonywać dokładnych pomiarów odcinków czasu.

12.3.4 Wejście-wyjście z blokowaniem oraz bez blokowania

Pozostał nam do omówienia jeszcze jeden aspekt interfejsu odwołanego do systemu – chodzi o wybór między wejściem-wyjściem z blokowaniem lub bez blokowania (asynchronicznym). Gdy program użytkowy wykonuje *Blokowanie* (ang. *blocking*) wywołanie systemowe, wówczas jego działanie zostaje wstrzymane. Aplikacja zostaje przeniesiona z systemowej kolejki procesów gotowych do kolejki procesów czekających. Po zakończeniu funkcji systemowej aplikacja wraca do kolejki procesów gotowych, z której kandyduje do wznowienia działania i z chwilą jego podjęcia otrzymuje wartości przekazane przez tę funkcję. Fizyczne czynności wykonywane przez urządzenia wejścia-wyjścia są z reguły asynchroniczne; zajmują one nieokreśloną ilość czasu. Niemniej jednak w interfejsach użytkowych większości systemów operacyjnych występują odwołania do systemu powodujące blokowanie, gdyż kod blokowanych aplikacji jest łatwiejszy do zrozumienia niż kod aplikacji działających bez blokowania.

Niektóre procesy poziomu użytkowego wymagają wejścia-wyjścia bez blokowania. Jednym z przykładów jest interfejs użytkownika, w którym sy-

gnały z klawiszy i myszki przeplatają się z przetwarzaniem i wyświetlaniem danych na ekranie. Innym przykładem jest videoaplikacja czytająca klatki z pliku na dysku, a jednocześnie rozpakowująca i wyświetlająca informacje na ekranie.

Jednym ze sposobów, za pomocą których twórcy aplikacji może uzyskać zachodzenie na siebie obliczeń i operacji wejścia-wyjścia, jest napisanie aplikacji wielowątkowej. Część wątków będzie się odwoływać do systemu w sposób blokowany, podczas gdy inne będą kontynuować działanie. Projektanci systemu Solaris zastosowali tę metodę do realizacji biblioteki poziomu użytkownika, obsługującej wejście-wyjście asynchronicznie i uwalniającej od realizacji tego zadania twórców aplikacji. Systemowe operacje wejścia-wyjścia w niektórych systemach operacyjnych działają bez blokowania. Nieblokowane wywołanie nie wstrzymuje wykonywania aplikacji na dłuższy czas. Przeciwnie – kończy się ono szybko, przekazując aplikacji informację o liczbie przesłanych bajtów.

Alternatywą dla wywołania systemowego bez blokowania jest asynchroniczne odwołanie do systemu^{*}. Powrót po asynchronicznym odwołaniu do systemu następuje natychmiast, bez czekania na zakończenie operacji wejścia-wyjścia. Aplikacja kontynuuje wykonywanie swojego kodu, a zakończenie operacji wejścia-wyjścia, następujące w jakiś czas potem, zostaje jej zakończeniem komunikowane przez ustawienie pewnej zmiennej w przestrzeni adresowej aplikacji albo przez przekazanie jej sygnału, albo przerwania programowego, bądź też za pomocą przywołania (ang. *callback routine*) wykonywanego poza porządkiem obowiązującym w aplikacji. Zwrócmy uwagę na różnicę między wywołaniami systemowymi bez blokowania a asynchronicznymi odwołaniami do systemu. Nieblokowana operacja **czytaj** zwraca sterowanie natychmiast, niezależnie od tego, ile zdola przekazać danych: pełną zamówioną liczbę, mniejszą lub wcale. Przesłanie zamówione przez asynchroniczne wywołanie **czytaj** zostanie wykonane w całości, lecz zostanie zakończone po jakimś czasie.

Dobrym przykładem działania bez blokowania jest wywołanie systemowe **wybierz** (ang. *select*) dotyczące gniazd sieciowych. Ma ono jeden argument, który określa maksymalny czas oczekiwania. Nadając temu argumentowi wartość 0, aplikacja może pytać o pracę sieci bez blokowania. Jednak stosowanie funkcji **wybierz** pociąga za sobą dodatkowe koszty, ponieważ wywołanie **wybierz** sprawdza tylko, czy wejście-wyjście jest możliwe. W celu przesłania danych po wywołaniu **wybierz** musi następować jakaś operacja czytania lub pisania. Odmianę tego podejścia można napotkać w systemie Mach w postaci

^{*} Stosuje się również określenia: działania bez blokowania i działania asynchroniczne – Przyp. tłum.

zwielokrotnionego czytania z blokowaniem. W jednym odwołaniu do systemu określa się potrzebną liczbę operacji czytania odnoszących się do kilku urządzeń, a powrót z wywołania następuje wówczas, gdy którakolwiek z tych operacji zakonczy działanie.

12.4 ■ Podsystem wejścia-wyjścia w jądrze

Jądra systemów operacyjnych udostępniają wiele usług dotyczących wejścia-wyjścia. W tym punkcie omawiamy kilka usług realizowanych przez podsystem wejścia-wyjścia jądra oraz rozważamy sposób, w jaki korzystają one z infrastruktury tworzonej przez sprzęt i moduły sterujące urządzeń.

Omawiamy następujące usługi: planowanie wejścia-wyjścia, buforowanie, przechowywanie podręczne, spooling, rezerwowanie urządzeń i obsługę błędów.

12.4.1 Planowanie wejścia-wyjścia

Przez zaplanowanie zbioru zamówień na operacje wejścia-wyjścia rozumie się określenie dobrego porządku ich wykonywania. Porządek, w którym programy użytkowe odwołują się do systemu, rzadko kiedy jest najlepszym z możliwych. Planowanie może poprawić ogólną wydajność systemu, polepszyć wspólne korzystanie z urządzeń przez procesy i zmniejszyć średni czas oczekiwania na zakończenie operacji wejścia-wyjścia. Oto prosty przykład ilustrujący te możliwości. Przypuśćmy, że ramię dysku jest blisko początku dysku* oraz że trzy aplikacje zamawiają na tym dysku operacje czytania z blokowaniem. Aplikacja 1 żąda bloku położonego blisko końca dysku, aplikacja 2 zamawia blok u początku dysku, a aplikacja 3 zgłasza zapotrzebowanie na blok w środku dysku. Jest jasne, że system operacyjny może zmniejszyć drogi, którą pokona ramię dysku, obsługując te aplikacje w porządku 2, 3, 1. Tego rodzaju zmiana porządku obsługiwanego jest tym, o co chodzi w planowaniu wejścia-wyjścia. Twórcy systemów operacyjnych implementują planowanie przez utrzymywanie kolejek zamówień do każdego urządzenia. Gdy aplikacja wywołuje blokowaną, systemową operację wejścia-wyjścia, wówczas umieszcza się ją w kolejce do danego urządzenia. Planista wejścia-wyjścia zmienia porządek w kolejce, mając na względzie polepszanie ogólnej sprawności systemu i czasu odpowiedzi doświadczanego przez aplikacje. System operacyjny może też próbować być sprawiedliwym, tzn. dbać o to, aby żadna z aplikacji nie była źle obsługiwana, lub może dawać pierw-

* Tzn. w pobliżu jego zewnętrznzej krawędzi – Przyp. tłum.

szeństwo obsłudze pilnych zamówień. Na przykład zamówienia pochodzące od podsystemu pamięci wirtualnej mogą mieć pierwszeństwo przed zadaniami aplikacji. Kilka algorytmów planowania dostępu do dysku jest szczegółowo omówionych w p. 13.2.

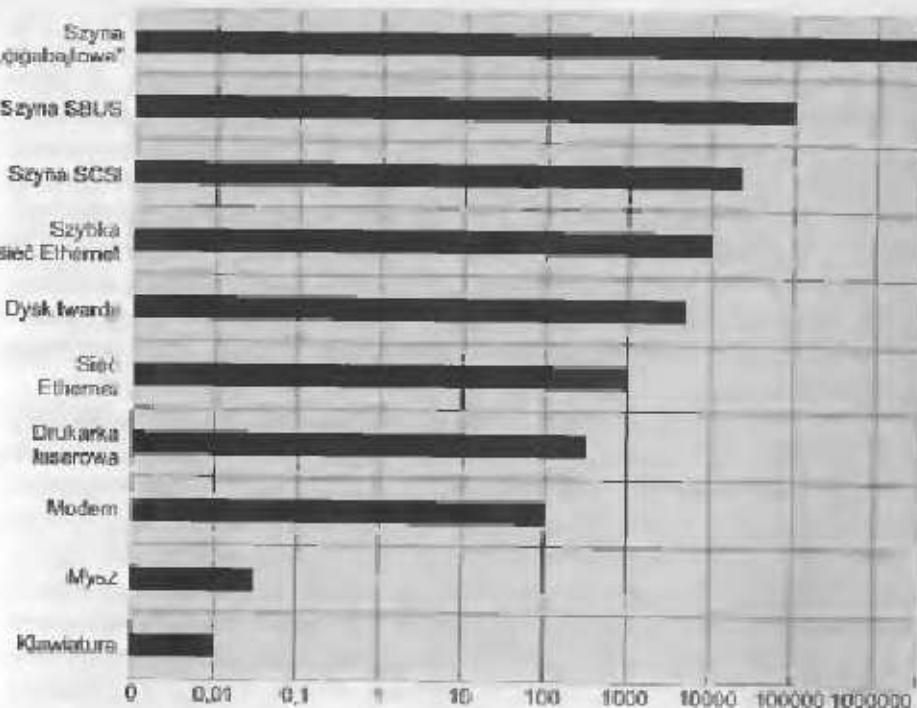
Jednym ze sposobów, w jaki podsystem wejścia-wyjścia polepsza wydajność komputera, jest planowanie operacji wejścia-wyjścia. Inny sposób polega na wykorzystywaniu miejsca w pamięci operacyjnej lub na dysku za pomocą metod nazywanych buforowaniem, przechowywaniem podręcznym oraz spoolingiem.

12.4.2 Buferowanie

Bufor (ang. *buffer*) jest obszarem pamięci, w którym przechowuje się dane przesypane między dwoma urządzeniami lub między urządzeniem a aplikacją. Istnieją trzy powody uzasadniające buforowanie. Pierwszym jest konieczność radzenia sobie z dysproporcjami między szybkościami producenta i konsumenta strumienia danych. Założymy na przykład, że plik zapamiętywany na dysku dociera do niego za pomocą modemu. Modem jest około tysiąc razy wolniejszy niż dysk twardy. Dlatego w pamięci operacyjnej jest tworzony bufor, aby gromadzić bajty nadchodzące z modelem. Kiedy cały bufor zostanie zapelniony, można go zapisać na dysku za pomocą jednej operacji. Ponieważ pisanie na dysku nie jest natychmiastowe, a modem wszędzie potrzebuje miejsca do zapamiętywania kolejnych danych, korzysta się z dwóch buforów. Po zapelnieniu przez modem pierwszego bufora następuje zamówienie operacji pisania na dysku. Zanim modem zapieśni drugi bufor, operacja zapisania zawartości pierwszego bufora na dysku powinna się zakończyć, więc modem można znów przełączyć do pierwszego bufora, podczas gdy dysk zapisuje informacje z drugiego bufora. Takie *podwójne buforowanie* (ang. *double buffering*) oddziela producenta danych od konsumenta, osłabiając tym samym zachodzące między nimi związki czasowe. Potrzebę tego oddzielenia widać na rys. 12.8, na którym ukazano wyjątkowe różnice szybkości urządzeń typowego sprzętu komputerowego.

Drugi powód uzasadniający buforowanie to konieczność dopasowania urządzeń o różnych rozmiarach przesyłanych jednostek danych. Niedopasowania tego rodzaju są szczególnie powszechnie w pracy sieciowej, kiedy to bufore znajdują szerokie zastosowanie przy fragmentowaniu i składaniu komunikatów. Po stronie wysyłającej dzieli się duże komunikaty na małe pakiety sieciowe. Pakiety przesyła się przez sieć i po stronie odbiorczej zestawia się je z powrotem w buforze, tworząc obraz danych źródłowych.

Trzeci powód stosowania buforowania to potrzeba zapewnienia semantyki kopii w na wejściu i wyjściu aplikacji. Pojęcie „semantyki kopii” wyjaśnimy



Rys. 12.8 Szybkości przesyłania w urządzeniach komputera Sun Enterprise 6000
(w skali logarytmicznej)

na przykładzie. Przypuśćmy, że aplikacja ma bufor danych, które chce zapisać na dysku. Wywołuje w tym celu funkcję systemową `pisz`, podając wskaźnik do bufora i wartość całkowitą określającą liczbę bajtów do zapisania. Co się stanie, jeśli po wywołaniu funkcji systemowej aplikacja zmieni zawartość bufora? W semantyce kopii gwarantuje się, że wersja danych zapisana na dysku będzie wersją z chwili odwołania się przez aplikację do systemu, niezależnie od jakichkolwiek późniejszych zmian w buforze aplikacji. Prosta metoda zapewnienia przez system operacyjny semantyki kopii polega na przekopiowaniu danych aplikacji przez funkcję systemową `pisz` do bufora w jądrze, zanim nastąpi przekazanie sterowania do aplikacji. Pisanie na dysku odbywa się z użyciem bufora w jądrze, zatem dalsze zmiany w buforze aplikacji nie wywołają żadnych skutków. Kopiowanie danych między buforami jądra a przestrzenią danych aplikacji jest w systemach operacyjnych czynnością typową pomimo kosztów, jakie operacja ta wprowadza – chodzi o czystość semantyki. Ten sam skutek można uzyskać wydajniej za pomocą sprytnych odwzorowań w pamięci wirtualnej i ochrony strony w trybie kopiowania przy zapisie.

12.4.3 Przechowywanie podręczne

Pamięć podręczna (ang. *cache*) jest obszarem szybkiej pamięci, w której przechowuje się kopię danych. Dostęp do kopii przechowywanej podręcznie jest szybszy niż dostęp do oryginału. Na przykład instrukcje bieżąco wykonywanego procesu są pamiętane na dysku, podręcznie przechowywane w pamięci operacyjnej, a ponadto kopiowane do pomocniczej i podstawowej pamięci podręcznej procesora. Różnica między buforem a pamięcią podręczną polega na tym, że bufor może zawierać jedyną, istniejącą kopię danych, natomiast przez przechowywanie podręczne rozumie się po prostu utrzymywanie w szybszej pamięci kopii obiektu danych, który przebywa gdzie indziej.

Przechowywanie podręczne i buforowanie to dwa odmienne działania, lecz niekiedy dany obszar pamięci może być używany w obu tych celach. Aby na przykład zachować semantykę kopii i umożliwiać skuteczne planowanie operacji dyskowych, system operacyjny stosuje bufory w pamięci operacyjnej, w których pamięta się dane. Z tych buforów korzysta się również jak z pamięci podręcznej, aby polepszać wydajność operacji wejścia-wyjścia na plikach dzielonych przez aplikacje lub takich plikach, które są zapisywane i zaraz potem czytane. Gdy jądro otrzyma zamówienie na plikową operację wejścia-wyjścia, wówczas najpierw sprawdza, czy w buforze pamięci podręcznej organizowanym w pamięci operacyjnej nie ma potrzebnego fragmentu pliku. Jeśli tak, to można uniknąć fizycznej transmisji dyskowej lub ją opóźnić. W podobny sposób na kilka sekund gromadzi się w pamięci podręcznej buforów wyniki dyskowych operacji pisania, aby zbierać duże porcje informacji do przesłania, co umożliwia wykonywanie wydajnych planów operacji pisania. Strategię opóźniania pisania w celu polepszania efektywności operacji wejścia-wyjścia omawiamy w kontekście dostępu do plików zdalnych w p. 17.3.

12.4.4 Spooling i rezerwowanie urządzeń

Mianem *spoolingu*⁶ określa się użycie bufora do przechowywania danych przeznaczonych dla urządzenia, które nie dopuszcza przeplatań danych w przeznaczonym dla niego strumieniu. Przykładem takiego urządzenia jest drukarka. Choć drukarka w konkretnej chwili może obsługiwać tylko jedno zadanie, wiele aplikacji może być gotowych do drukowania wyników współbieżnie. Należy więc zadbać, żeby wyniki te nie zostały ze sobą pomieszczone. System

⁶ Nazwa jest skrótem od słów *simultaneous peripheral operation on-line* (jednoczesna bezpośrednią pracę urządzeń), zob. p. 12 – Przyp. tłum.

operacyjny rozwiązuje ten problem przez przechwytywanie wszystkich informacji kierowanych na drukarkę. Dane wyjściowe każdej aplikacji są gromadzone w osobnym pliku-buforze. Gdy aplikacja skończy drukować, system organizujący spooling ustawia plik z obrazem drukowanych wyników w kolejce do drukarki. Pliki te są kopiowane na drukarce przez system spoolingu po kolej. W niektórych systemach operacyjnych spooling jest zarządzany przez proces systemowego demona⁷. W innych systemach spooling jest obsługiwany przez wątek w jądrze. W obu przypadkach system operacyjny dostarcza interfejsu sterującego, który umożliwia użytkownikom i administratorem systemu wyświetlanie kolejki, usuwanie niepotrzebnych zadań przed ich wydrukowaniem, zawieszanie drukowania na czas obsługi drukarki itp.

Niektóre urządzenia, takie jak przewijaki taśmy i drukarki, nie potrafią skutecznie przeplać obsługi zamówień wejścia-wyjścia pochodzących z wielu współbieżnych aplikacji. Spooling jest jednym ze sposobów koordynowania przez systemy operacyjne współbieżnych operacji wejście-wyjście. Inny sposób postępowania ze współbieżnym dostępem do urządzeń polega na zorganizowaniu jawnych środków koordynacji. Niektóre systemy operacyjne (w tym – system VMS) umożliwiają dostęp do urządzeń na zasadzie wyłączności; proces może przydzielić sobie bezczynne urządzenie i zwolnić je wówczas, gdy przestanie mu być potrzebne. W innych systemach operacyjnych wymusza się ograniczenie, aby do tego rodzaju urządzeń mógł istnieć tylko jeden uchwyt otwartego pliku. Wiele systemów operacyjnych zawiera funkcje, które pozwalają procesom koordynować wyłączność dostępu. Na przykład w systemie Windows NT istnieją funkcje systemowe umożliwiające czekanie na udostępnienie urządzenia. W systemie tym wywołanie systemowe open otwierające plik ma parametr określający rodzaj dostępu, na który zezwala się innym, współbieżnym wątkom. Unikanie zakleszczeń należy w tych systemach do obowiązków aplikacji.

12.4.5 Obsługa błędów

System operacyjny korzystający z pamięci chronionej może strzec przed wieloma rodzajami błędów sprzętowych i błędów popełnianych przez programy użytkowe, nie dopuszczając, aby jakaś drobna, mechaniczna niesprawność prowadziła do awarii całego systemu. Urządzenia i operacje wejścia-wyjścia mogą ulegać różnorodnym awariom z przyczyn doraźnych, jak przeładowanie sieci, lub „trwałych” – takich jak uszkodzenie sterownika z dysku. Niejednokrotnie systemy operacyjne potrafią skutecznie przeciwdziałać awariom prze-

⁷ Demon (ang. *daemon*) to trwały proces systemowy, na którym poświęcający w uśpieniu (oczekujący na zadanie) lub pobudzany do działania przez zdarzenie systemowe. – Przyp. tłum.

ściowym. Na przykład na nieudane czytanie z dysku można odpowiedzieć powtóżeniem operacji czytania, a błąd powstający przy przesyłaniu sieci może być kompensowany wykonaniem operacji powtórnego przesłania, jeśli przewidziano ją w protokole. Niestety, jeżeli ważna część systemu ulegnie trwałej awarii, to szansa na pokonanie przez system tej przeszkoły jest nika.

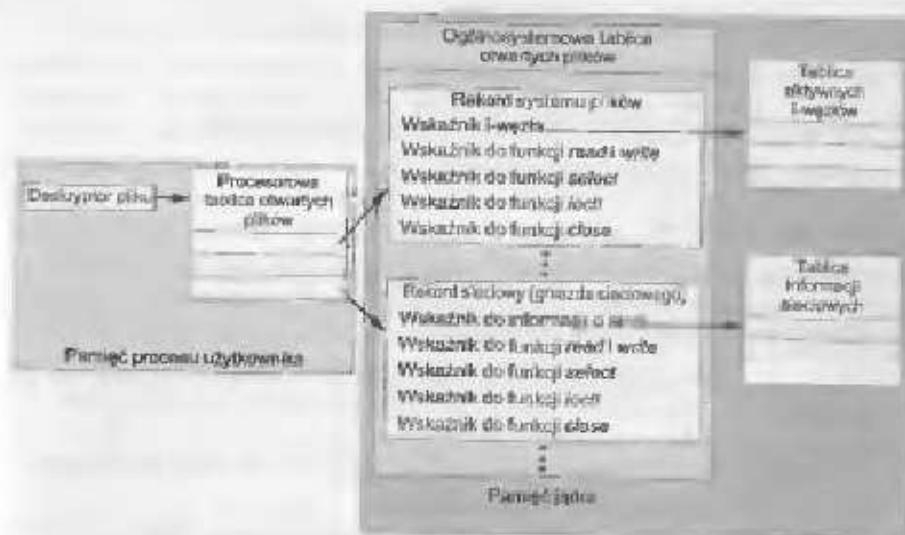
Przyjmuje się ogólnie, że systemowe wywołanie wejścia-wyjścia spowoduje zwrotne przekazanie jednego bitu informacji określającej skutek wywołania: sukces lub niepowodzenie. W systemie operacyjnym UNIX zastosowano dodatkowo zmienną całkowitą o nazwie *errno*^{*}, za pośrednictwem której jest przekazywany kod błędu, czyli jedna z około stu wartości informujących ogólnie o charakterze błędu (np. przekroczenie dopuszczalnego przedziału przez wartość argumentu, użycie złego wskaźnika lub nieotworzenie pliku). W porównaniu z tym niektóre rodzaje sprzętu mogą dostarczać bardzo szczegółowych informacji o błędach, jednak w wielu dzisiejszych systemach operacyjnych nie przewidziano możliwości przenoszenia tych informacji do programu użytkowego. Na przykład awaria urządzenia standardu SCSI jest komunikowana za pomocą protokołu SCSI w formie „klucza diagnostycznego” (ang. *sense key*), który identyfikuje ogólny rodzaj błędu, jak błąd sprzętowy lub niedozwolone zamówienie. Dodatkowy kod diagnostyczny (ang. *additional sense code*) informuje o kategorii błędu, na przykład o złym parametrze polecenia lub niepowodzeniu samotestowania. Jeszcze więcej szczegółów zawiera określnik dodatkowego kodu diagnostycznego (ang. *additional sense code qualifier*); podaje on na przykład, który parametr polecenia był błędny lub który z podsystemów sprzętowych nie przeszedł pomyślnie samotestowania. Ponadto wiele urządzeń SCSI rejestruje na wewnętrznych stronach informacje diagnostyczne, o które procesor główny może poprosić, lecz korzysta się z tego rzadko.

12.4.6 Struktury danych jądra

Jądro musi przechowywać informacje o stanie używanych składowych wejścia-wyjścia. Robi to za pomocą rozmaitych, wewnętrznych struktur danych, takich jak tabela otwartych plików z p. 11.1. Jądro korzysta z wielu podobnych struktur, które służą mu do połączeń sieciowych, komunikacji z urządzeniami znakowymi i innych działań na wejściu i wyjściu.

W uniksowym systemie plików udostępnia się wiele różnych obiektów, na przykład pliki użytkowników, surowe urządzenia i przestrzenie adresowe procesów. Chociaż każdy z tych obiektów umożliwia czytanie, semantyki tej operacji są różne. Na przykład w celu czytania pliku użytkownika jądro musi sprawdzić bufor pamięci podręcznej, zanim podejmie decyzję o wykonaniu

* Jak: error number – numer błędu – Przyp. tłum.



Rys. 12.9 Struktura wejścia-wyjścia w jądrze UNIX

operacji czytania z dysku. Aby czytać z dysku, jądro musi się upewnić, czy rozmiar zamówienia jest wielokrotnością rozmiaru sektora dyskowego i czy leży w granicach sektorów. Do przeczytania obrazu procesu wystarczy przekopiować dane z pamięci operacyjnej. System UNIX obudowuje te różnice jednolitą strukturą za pomocą metod obiektowych. Przedstawiony na rys. 12.9 rekord otwartego pliku zawiera tabelę odesłań ze wskaźnikami do odpowiednich procedur – zależnie od typu pliku.

W niektórych systemach operacyjnych metody obiektowe znajdują znacznie większe zastosowanie. W systemie Windows NT na przykład wejście-wyjście jest zrealizowane na zasadzie przekazywania komunikatów. Zamówienie operacji wejścia-wyjścia jest przekształcane na komunikat wysypany za pośrednictwem jądra do zarządcy wejścia-wyjścia, a stamtąd do modułu sterującego, przy czym każdy z tych pośredników może zmienić zawartość komunikatu. W celu wykonania operacji wyjścia komunikat zawiera dane do pisania. Komunikat na wejściu zawiera bufor przeznaczony na przyjęcie danych. Podejście polegające na przekazywaniu komunikatów może być kosztowniejsze w porównaniu z metodami proceduralnymi korzystającymi z dzielonych struktur danych, lecz upraszcza projektowanie i budowę systemu wejścia-wyjścia oraz zwiększa jego elastyczność.

Podsumowując, można powiedzieć, że podsystem wejścia-wyjścia koordynuje szeroki zbiór usług dostępnych dla aplikacji i pewnych części jądra. Podsystem wejścia-wyjścia nadzoruje:

- zarządzanie przestrzenią nazw plików i urządzeń;
- przebieg dostępu do plików i urządzeń;
- poprawność operacji (np. modem nie może przeszukiwać);
- przydzielanie miejsca w systemie plików;
- przydział urządzeń;
- buforowanie, przechowywanie podręczne oraz spooling;
- planowanie operacji wejścia-wyjścia;
- doglądanie stanu urządzeń, obsługę błędów oraz czynności naprawcze po awarii;
- konfigurowanie i wprowadzanie w stan początkowy modułu sterującego.

Dostęp do urządzeń na górnym poziomie podsystemu wejścia-wyjścia odbywa się za pomocą jednolitego interfejsu dostarczanego przez programowe moduły sterujące.

12.5 ■ Przekształcanie zamówień wejścia-wyjścia na operacje sprzętowe

Omówiliśmy wcześniej przebieg uzgodnień dokonywanych między modelem sterującym urządzeniem a jego sterownikiem sprzętowym. Nie wyjaśniliśmy jednak, w jaki sposób system operacyjny wiąże pochodzące od aplikacji zamówienie ze zbiorem kabli sieciowych lub konkretnym sektorem dysku. Rozważmy na przykład czytanie pliku z dysku.

Program użytkowy odwołuje się do danych za pomocą nazwy pliku. Odwozowanie nazwy pliku za pomocą katalogów systemu plików, mając na celu uzyskanie w obrębie dysku informacji o miejscu przydzielonym plikowi, należy do zadań systemu plików. Na przykład w systemie MS-DOS nazwa jest odwzorowywana na liczbę wskazującą pozycję w tablicy dostępu do pliku^{*}, a zawarty w tej pozycji wpis określa, które bloki dyskowe są przydzielone do pliku. W systemie UNIX nazwa jest odwzorowywana na numer i-węzła, a odpowiedni i-węzeł zawiera informacje o przydzielaniu miejsca na dysku.

W jaki sposób odbywa się powiązanie nazwy pliku z sterownikiem dysku (adresem portu sprzętowego lub odwzorowanymi w pamięci rejestrami sterownika)?

^{*} Czyli w tablicy FAT – Przyp. tłum.

Rozpatrzymy najpierw system MS-DOS – stosunkowo prosty system operacyjny. Pierwsza część nazwy pliku systemu MS-DOS (poprzedzająca dwukropka) jest napisem identyfikującym konkretną jednostkę sprzętu. Na przykład **e:** stanowi pierwszą część każdej nazwy odnoszącej się do podstawowego dysku twardego. To, że **e:** oznacza podstawowy dysk twardy, jest wbudowane w system operacyjny; napis ten jest odwzorowany na określony adres portu za pomocą tablicy urządzeń. Dwukropka oddziela przestrzeń nazw urządzeń od przestrzeni nazw systemu plików w obrębie każdego urządzenia. Takie odseparowanie ułatwia systemowi operacyjnemu przypisywanie dodatkowych funkcji poszczególnym urządzeniom. Łatwo jest na przykład objąć spoolingiem dowolne pliki przeznaczone dla drukarki.

Jeśli natomiast przestrzeń nazw urządzeń jest zaliczona do przestrzeni regularnych nazw systemu plików, jak to ma miejsce w systemie UNIX, to automatycznie uzyskuje się zwykłe usługi nazewnicze. Jeżeli system plików realizuje prawa własności i kontrolę dostępu w odniesieniu do wszystkich nazw, to urządzenia też uzyskują właścicieli i kontrolę dostępu. Ponieważ pliki są przechowywane na urządzeniach, więc interfejs tego typu umożliwia dostęp do systemu plików na dwóch poziomach. Za pomocą nazw można uzyskiwać dostęp do samych urządzeń lub do zgromadzonych na nich plików.

W systemie UNIX nazwy urządzeń są pamiętane w regularnej przestrzeni nazw systemu plików. W odróżnieniu od nazwy pliku systemu MS-DOS, w której występuje separator w postaci dwukropka, unikowa nazwa ścieżki nie wyróżnia wyraźnie części dotyczącej urządzenia. W rzeczywistości żadna część nazwy ścieżki nie jest przeznaczona na nazwę urządzenia. System UNIX ma *tablice montażu*, która wiąże przedrostki nazw ścieżek z nazwami poszczególnych urządzeń. Aby przetłumaczyć nazwę ścieżki, system UNIX dopasowuje do niej w tablicy montażu największy przedrostek, odpowiadający mu wpis w tablicy montażu zawierający nazwę urządzenia. Ta nazwa również ma postać nazwy z przestrzeni systemu plików. W wyniku poszukiwania jej przez system UNIX w strukturze katalogowej systemu plików zamiast numeru i-węzła zostaje odnaleziony numer urządzenia w postaci pary **<starszy, młodszy>**. Starszy numer urządzenia identyfikuje moduł sterujący, który należy wywołać w celu obsługi operacji wejścia-wyjścia tego urządzenia. Młodszy numer urządzenia jest przekazywany do modułu sterującego, aby posłużyć jako indeks w tablicy urządzeń. Odpowiedni wpis w tablicy urządzeń zawiera poszukiwany adres portu lub odwzorowany w pamięci adres sterownika urządzenia.

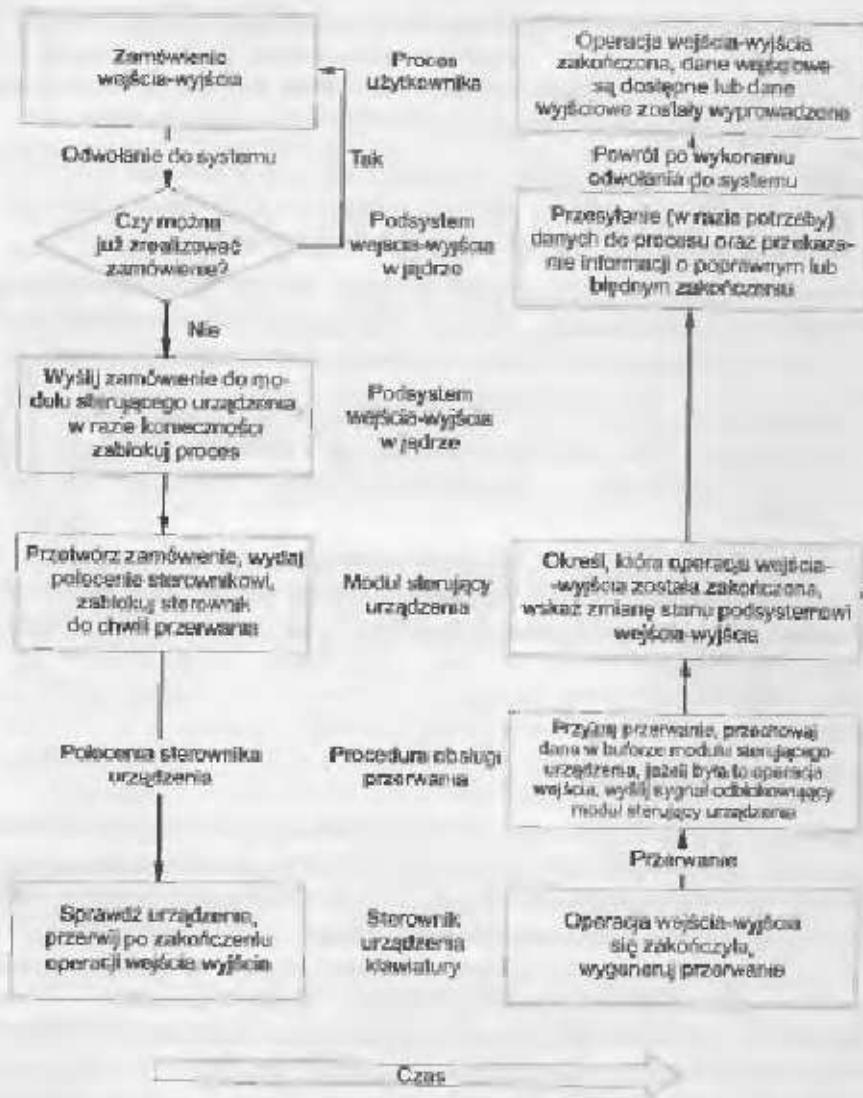
Dzięki wieloetapowemu przeszukiwaniu tablic na drodze od zamówienia do fizycznego sterownika urządzenia nowoczesne systemy operacyjne zyskują większą elastyczność. Mechanizmy przekazywania zamówień od aplikacji do modułów sterujących są ogólne. Nowe urządzenia i moduły ich ob-

slugi można więc wprowadzać do komputera bez powtórzonych komplikacji jądra. W istocie, pewne systemy operacyjne mają możliwość indywidualizowania modułów sterujących na życzenie. Podeczas rozruchu system sprawdza najpierw szyny sprzętowe, aby określić, jakie urządzenia są obecne, po czym wprowadza do pamięci niezbędne moduły sterujące, robi to natychmiast lub przy pierwszym wystąpieniu zamówienia na operację wejścia-wyjścia.

System UNIX w wersji V ma ciekawy mechanizm nazywany strumieniami, który umożliwia aplikacji dynamiczne załatwianie kodu modułów obsługi w potoku. *Strumień* (ang. *stream*) jest połiduplexowym połączeniem między modelem sterującym a procesem poziomu użytkownika. Składa się z *czola strumienia* (ang. *stream head*), będącego interfejsem z procesem użytkownika, i *zakończenia sterującego* (ang. *driver end*) nadzorującego urządzenie. Między nimi może ponadto występować kilka *modułów strumienia* (ang. *stream modules*). Moduły można umieszczać w strumieniu w celu dodawania do niego funkcji w sposób warstwowy. Proces może na przykład otworzyć port szeregowy urządzenia za pośrednictwem strumienia, do którego jest wstawiony moduł umożliwiający redagowanie nadchodzących danych. Strumieni można używać do komunikacji międzyprocesowej i sieciowej. Mechanizm gniazda w Systemie V jest naprawdę zrealizowany za pomocą strumieni.

Omówimy teraz typową drogę zamówienia czytania z blokowaniem, przedstawioną na rys. 12.10. Z rysunku wynika, że operacja wejścia-wyjścia wymaga wiele kroków, na których wykonanie zużywa się olbrzymią liczbę cykli procesora.

1. Proces zamawia w systemie blokującą go operację czytania, powołując się na deskryptor uprzednio otwartego pliku.
2. W kodzie należącej do jądra funkcji systemowej następuje sprawdzenie poprawności parametrów. Jeżeli zamawiana operacja dotyczy wejścia i potrzebne dane znajdują się akurat w buforze pamięci podręcznej, to następuje przekazanie ich do procesu i zamówienie wejścia-wyjścia zostaje spełnione.
3. W przeciwnym razie powstaje konieczność wykonania fizycznej operacji wejścia-wyjścia. Proces zostaje usunięty z kolejki procesów gotowych do wykonywania i umieszczony w kolejce procesów czekających na dane urządzenie. Potrzebna operacja wejścia-wyjścia podlega zaplanowaniu. Ostatecznie podsystem wejścia-wyjścia pośle zamówienie do modułu sterującego. W zależności od systemu operacyjnego zamówienie zostaje przesłane za pomocą wywołania podprogramu lub za pośrednictwem wewnętrznego komunikatu jądra.



Rys. 12.10 Etapy wykonania zamówień w e-commerce.

- Moduł sterujący rezerwuje miejsce na przyjęcie danych w buforze jądra i planuje operację wejścia-wyjścia. W odpowiedniej chwili moduł sterujący wysyła polecenia do sterownika, zapisując je w rejestrach sterujących urządzenia.
 - Sterownik urządzenia nakazuje sprzętowym podzespołom urządzenia wykorzystanie przesłania danych.

6. Moduł sterujący może odpytywać o stan urządzenia i dane lub może zorganizować przesyłanie w trybie DMA do pamięci jądra. Zakładamy, że przesyłanie to jest obsługiwane przez sterownik bezpośredniego dostępu do pamięci, który po zakończeniu przesyłania powoduje przerwanie.
7. Właściwa procedura obsługi przerwania odbiera przerwanie za pośrednictwem tablicy wektorów przerwań, zapamiętuje niezłe dane, przekazuje sygnał do modułu sterującego i kończy obsługę przerwania.
8. Moduł sterujący urządzenia odbiera sygnał, określa, która operacja wejścia-wyjścia się skończyła, określa stan zamówienia i powiadamia podsystem wejścia-wyjścia w jądrze, że zamówienie zostało zakończone.
9. Jądro przesyła dane lub przekazuje umowne kody do przestrzeni adresowej procesu, który złożył zamówienie, i przemieszcza proces z kolejki procesów oczekujących z powrotem do kolejki procesów gotowych do działania.
10. Przeniesienie procesu do kolejki procesów gotowych powoduje jego odblokowanie. Gdy planista przydziału procesora przydzieli proces do procesora, wówczas nastąpi wznowienie jego pracy po zakończonym odwołaniu do systemu.

12.6 ■ Wydajność

Wejście-wyjście ma istotny wpływ na wydajność systemu. Nakłada ono ostre wymagania na procesor w związku z wykonywaniem kodu modułu sterującego i dążeniem do harmonijnego, skutecznego planowania blokowania i odblokowywania procesów. Wynikające z tego zmiany kontekstów obciążają procesor i jego sprzętową pamięć podręczną. Operacje wejścia-wyjścia ujawniają także wszelkie nieefektywności zawartego w jądrze mechanizmu obsługi przerwań, a obłożenie operacjami wejścia-wyjścia spowalnia kopiowanie danych szyną między sterownikami a pamięcią fizyczną oraz kopiowanie między buforami jądra a przestrzenią danych aplikacji. Zgrabne podanie wszystkim tym wymaganiom jest jednym z najważniejszych zadań budowniczego systemu komputerowego.

Choć współczesne komputery mogą obsługiwać setki przerwań na sekundę, obsługa przerwania pozostaje zadaniem względnie kosztownym. Każde przerwanie powoduje zmianę stanu systemu, wykonanie procedury obsługi przerwania i odtworzenie stanu. Programowane wejście-wyjście może być wydajniejsze niż wejście-wyjście sterowane przerwaniami, jeżeli liczba cykli zużywanych na aktywne czekanie nie jest zbyt duża. Zakończeniu operacji

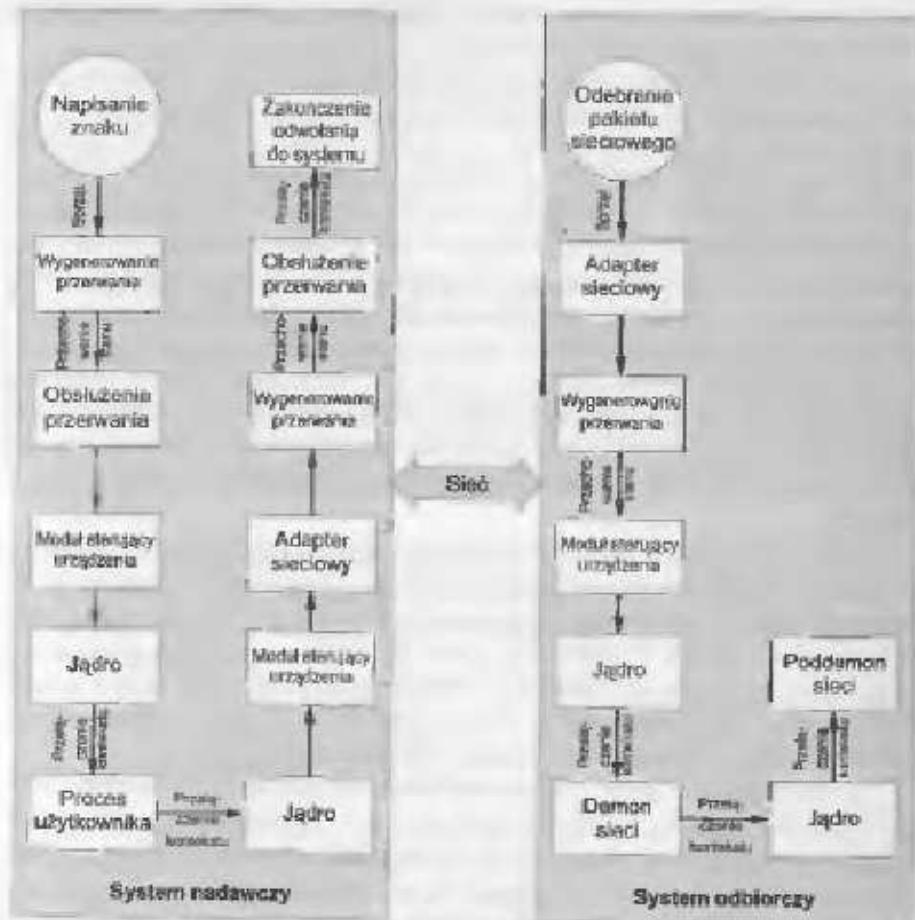
wejścia-wyjścia zazwyczaj towarzyszy odblokowanie procesu-wraz z całym kosztownym przełączeniem kontekstu.

Ruch w sieci może również powodować częste zmiany kontekstu. Rozważmy na przykład zdalne rejestrowanie się na odległej maszynie za pomocą innej maszyny. Każdy znak pisany na lokalnej maszynie musi być przesłany do maszyny zdalnej. Znak zostaje napisany na maszynie lokalnej: następuje przerwanie klawiaturowe, procedura obsługi przerwania przekazuje znak do modułu sterującego w jądrze, a następnie do procesu użytkownika. W celu przesłania znaku do maszyny zdalnej proces użytkownika wywołuje systemową operację sieciowego wejścia-wyjścia. Znak przechodzi wówczas do lokalnego jądra, a stamtąd, przez warstwy sieciowe konstruujące pakiet sieciowy, trafia do modułu obsługi urządzenia sieciowego. Moduł sterujący urządzenia sieciowego przekazuje pakiet do sterownika sieci, który wysyła znak i generuje przerwanie. Przerwanie jest przekazywane z powrotem przez jądro, aby spowodować zakończenie systemowej operacji sieciowego wejścia-wyjścia.

W tej chwili sprzęt sieciowy zdalnego systemu odbiera pakiet i powoduje przerwanie. Znak zostaje wydobyty z protokołów sieciowych i przekazany do odpowiedniego demonu sieci. Demon sieci identyfikuje, o którą zdalną sesję rejestrowania chodzi, i przekazuje pakiet do właściwego poddemonu danej sesji. Temu przepływowi informacji towarzyszą zmiany kontekstów i stanów (rys. 12.11). Zazwyczaj odbiorca wysyła jeszcze do nadawcy potwierdzenie odebrania znaku, a wtedy praca zostaje wykonana dwukrotnie.

Konstruktorzy systemu Solaris zaimplementowali na nowo demonu usługi *telnet*, korzystając z wątków jądra w celu wyeliminowania przełączeń kontekstu przy każdym przeniesieniu znaku między demonami a jadrem. Według oszacowań firmy Sun ulepszenie to umożliwiło zwiększenie na dużym serwerze maksymalnej liczby usług rejestrowania w sieci z kilkuset do kilku tysięcy.

W innych systemach do operacji wejścia-wyjścia dotyczących terminali używa się oddzielnych procesorów *członowych* (ang. *front-end processors*), aby zmniejszyć obciążenie jednostki centralnej powodowane obsługą przerwań. Na przykład *koncentrator terminali* (ang. *terminal concentrator*) może obsługiwać w sposób przeplatany w jednym porcie dużego komputera dane pochodzące z setek zdalnych terminali. *Kanał wejścia-wyjścia* (ang. *I/O channel*) jest wydzielonym, specjalizowanym procesorem występującym w komputerach głównych i innych wysokowydajnych systemach. Praca kanału polega na przejmowaniu od procesora głównego obciążen związkanych z operacjami wejścia-wyjścia. Z założenia kanały służą do obsługi przepływu danych, podczas gdy procesor główny zwalnia się do przetwarzania danych. W porównaniu ze sterownikami urządzeń i sterownikami bezpośredniego



Rys. 12.11 Komunikacja międzykomputerowa i jej koszty

dostępu do pamięci, występującymi w mniejszych komputerach, kanał może wykonywać bardziej ogólne i złożone programy; karty dają się więc stroić stosownie do specyficznych obciążeń.

Aby poprawiać wydajność wejścia-wyjścia, możemy korzystać z kilku zasad:

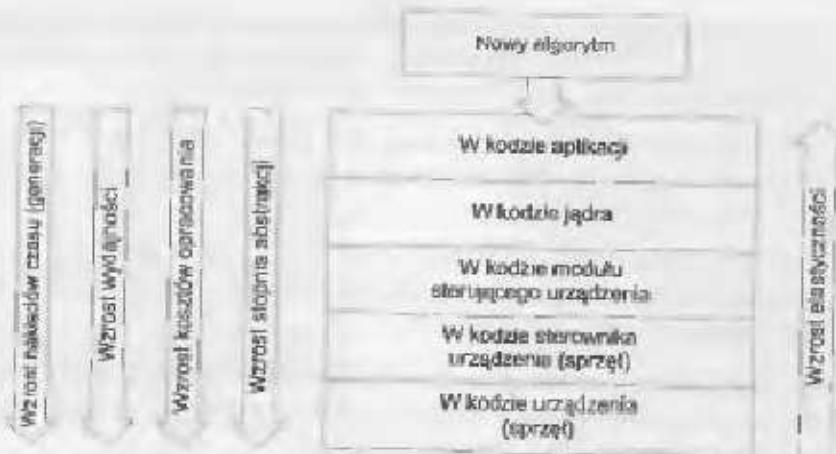
- Zmniejszać liczbę przełączeń kontekstu.
- Zmniejszać liczbę kopowania danych w pamięci podczas przekazywania ich od urządzenia do aplikacji.

- Zmniejszaćczęstośćwystępowaniaprzerwańprzezstosowanie wielkich przestran i przemyślnych sterowników, jak również za pomocą odpytywania (jeśli można zminimalizować aktywne czekanie).
- Zwiększaćwspółbieżnośćza pomocą sterowników pracujących w trybie DMA lub kanałów w celu uwalniania jednostki centralnej od zwykłego przesyłania danych.
- Realizowaćelementardziałaniaza pomocą sprzętu i pozwalać na ich współbieżne wykonywanie w sterownikach urządzeń – wraz z jednoczesnym działaniem szyny i procesora.
- Równoważyćwydajnośćprocesora, podsystemów pamięci, szyny i operacji wejścia-wyjścia, ponieważ przeciążenie w jednym miejscu będzie powodować bezczynność w innych miejscach.

Urządzenia różnią się znacznie pod względem złożoności. Myszka jest przykładem prostego urządzenia. Ruchy wykonywane myszką oraz naciśkanie jej przycisków zamieniają się na wartości liczbowe i przekazuje od sprzętu, przez moduł sterujący myszki do aplikacji. W porównaniu z tym funkcje modułu sterującego dysku używanego w systemie NT są złożone. Moduł ten nie tylko zarządza poszczególnymi dyskami, lecz również implementuje tablice RAID (zob. p. 13.5). W tym celu zamienia on pochodzące od aplikacji zamówienie czytania lub pisania na uporządkowany zbiór dyskowych operacji wejścia-wyjścia. Co więcej, moduł sterujący realizuje też wyszukaną obsługę błędów oraz algorytmy odtwarzania danych, a także podejmuje działania mające na celu optymalizowanie pracy dysku z uwagi na znaczenie, jakie wydajność pamięci pomocniczej ma dla efektywności całego systemu.

Gdzie należy implementować funkcje wejścia-wyjścia: w sprzęcie, w module sterującym czy w oprogramowaniu aplikacji? Niejednokrotnie deje się zaoferować drogę przedstawioną na rys. 12.12.

- Początkowo implementujemy eksperymentalne algorytmy wejścia-wyjścia na poziomie aplikacji, ponieważ kod aplikacji jest elastyczny, a występujące w nim błędy nie powodują swarii systemu. Opracowując kod na poziomie aplikacji, unikamy ponadto konieczności rozpoczęcia od nowa pracy systemu lub wymiany modułów sterujących po wykonaniu każdej zmiany w kodzie. Jednak implementacja na poziomie aplikacji może być mało wydajna z powodu kosztów przełączania kontekstu, jak również dlatego, że aplikacja nie może korzystać z wewnętrznych struktur danych jądra i jego funkcji (takich jak wydajne przekazywanie komunikatów wewnątrz jądra, stosowanie wątków i blokowanie zasobów).



Rys. 12.12 Lokalizacja funkcji urządzeń zewnętrznych

- Jeśli algorytm zrealizowany na poziomie aplikacji okaza się godny uwagi, to można zaimplementować go w jądrze. Pozwoli to polepszyć wydajność, lecz w jego opracowaniu trzeba włożyć znacznie więcej wysiłku, gdyż jądro systemu operacyjnego jest dużym i skomplikowanym modelem oprogramowania. Ponadto implementację w jądrze należy starannie przetestować, aby uniknąć uszkodzeń danych i załamań systemu.
- Największą wydajność można uzyskać przy udziale specjalizowanej implementacji sprzętowej – w urządzeniu lub jego sterowniku. Do wad implementacji sprzętowej zalicza się trudności i wydatki związane z jej dalszymi ulepszeniami lub usuwaniem błędów, zwiększyony czas opracowywania (miesiące zamiast dni) oraz zmniejszoną elastyczność. Na przykład sprzętowy sterownik RAID⁷ może ograniczać możliwość wpływania jądra na kolejność i położenie poszczególnych, czytanych lub zapisywanych bloków, pomimo że jądro dysponuje specjalnymi danymi dotyczącymi obciążenia, co mogłoby mu umożliwić poprawienie wydajności operacji wejścia-wyjścia.

12.7 ■ Podsumowanie

Do podstawowych elementów sprzętowych stosowanych na wejściu i wyjściu należą: szyny, sterowniki urządzeń i same urządzenia. Przenoszenie danych od urządzeń do pamięci głównej jest wykonywane przez procesor jako wej-

⁷ Nadmiarowej tablicy niezałącznych dysków (ang. redundant array of independent disks) -- Przyp. tłum.

ście-wyjście programowane lub przerzuca się je na sterownik DMA (bezpośredniego dostępu do pamięci). Moduł jądra nadzorujący działanie urządzenia nazywa się modelem sterującym. Udostępniany aplikacjom interfejs odwołań do systemu jest zaprojektowany tak, aby można było za jego pomocą obsługiwać kilka podstawowych kategorii sprzętu, takich jak urządzenia blokowe, znakowe, pliki odwzorowywane w pamięci, gniazda sieciowe i programowane czasomierze. Odwołania do systemu zazwyczaj powodują zablokowanie korzystającego z nich procesu, ale samo jądro oraz aplikacje, których nie wolno usypiać na czas oczekiwania na zakończenie operacji wejścia-wyjścia, stosują wywołania bez blokowania i wywołania asynchroniczne.

Podsystem wejścia-wyjścia w jądrze dostarcza wielu usług. Należą do nich m.in.: planowanie wejścia-wyjścia, buforowanie, spooling, obsługa błędów i rezerwowanie urządzeń. Inną usługą jest tłumaczenie nazw, w wyniku którego następuje powiązanie urządzeń sprzętowych z symbolicznymi nazwami plików, stosowanymi w aplikacjach. Obejmuje ono kilka poziomów odwzorowań, na których nazwy są tłumaczone z postaci napisowej na określony moduł sterujący lub adres urządzenia, a potem na adresy fizyczne portów wejścia-wyjścia lub sterowników szyn. Odwzorowanie takie może występować wewnętrz przerwanej przestrzeni nazw systemu plików, jak to się dzieje w systemie UNIX, albo w oddzielnej przestrzeni nazw urządzeń – jak w systemie MS-DOS.

Wywołania systemowe realizujące operacje wejścia-wyjścia, ze względu na zatrudnianie jednostki centralnej, są kosztowne. Przyczyną tego jest istnienie wielu warstw oprogramowania między urządzeniem fizycznym a aplikacją. Warstwy te powodują nakłyki wynikające z przełączania kontekstu, niezbędnego do przekraczania granic ochronnych jądra, obsługi sygnałów i przerwań, wymaganej przez urządzenia wejścia-wyjścia, oraz z obciążeniem procesora i systemu pamięci powodowanych kopowaniem danych między buforami jądra i przestrzenią aplikacji.

■ Ćwiczenia

- 12.1** Wymień trzy zalety lokalizowania funkcji w sterowniku urządzenia, a nie w jądrze. Wymień trzy wady takiej decyzji.
- 12.2** Rozważmy następujące scenariusze obsługi wejścia-wyjścia dotyczące komputera PC przeznaczonego dla jednego użytkownika:
 - (a) myszka używana z graficznym interfejsem użytkownika;
 - (b) przewijak taśmy w wielozadaniowym systemie operacyjnym (zakładmy, że nie dokonuje się wstępnych przydziałów urządzeń);

- (c) napęd dysku zawierający pliki użytkownika;
- (d) karta graficzna z bezpośrednim podłączeniem do szyny, dostępna za pomocą wejścia-wyjścia odwzorowywanego w pamięci.

Któż z metod warto byłoby wybrać w systemie operacyjnym dla każdego z tych scenariuszy: buforowanie, spooling, przechowywanie podręczne czy ich kombinację? Czy w operacjach wejścia-wyjścia wskazane byłoby zastosować odpytywanie, czy sterowanie za pomocą przerwań? Uzasadnij swoje wybory.

- 12.3** W przykładzie uzgodnień podanym w p. 12.2 zastosowano dwa bity: bit zajętości i bit gotowości do wykonania polecenia. Czy można by zrealizować te uzgodnienia za pomocą tylko jednego bitu? Jeżeli tak, to opisz odpowiedni protokół. Jeżeli nie, to wyjaśnij, dlaczego jeden bit to za mało?
- 12.4** Opisz trzy sytuacje, w których powinno się używać operacji wejścia-wyjścia z blokowaniem. Opisz trzy sytuacje, w których powinno się używać operacji wejścia-wyjścia bez blokowania. Czy nie wystarczyłoby realizować operacje wejścia-wyjścia bez blokowania i organizować w procesach aktywne czekanie na zakończenie pracy używanych przez nie urządzeń?
- 12.5** Dlaczego do obsługi pojedynczego portu szeregowego można by zastosować system sterowany przerwaniami, a do obsługi procesora czolowego, takiego jak koncentrator terminali, powinno się użyć metody odpytywania o stan rejestrów wejścia-wyjścia?
- 12.6** Wykrywanie zakończenia operacji wejścia-wyjścia za pomocą odpytywania może powodować marnowanie wielkiej liczby cykli jednostki centralnej, jeśli procesor wielokrotnie powtarza wykonanie pętli aktywnego czekania, zanim operacja wejścia-wyjścia się zakończy. Jeśli jednak urządzenie wejścia-wyjścia jest gotowe do pracy, to odpytywanie może być znacznie wydajniejsze niż przechwytywanie i obróbka przerwań. Opisz mieszanką strategię obsługi urządzenia wejścia-wyjścia, łączącą odpytywanie, usypianie i przerwania. Dla każdej z tych strategii (samo odpytywanie, same przerwania i metoda mieszana) przedstaw środowisko obliczeniowe, w którym dana strategia jest wydajniejsza od każdej z pozostałych.
- 12.7** System UNIX koordynuje działanie składowych jądra za pomocą manipulacji na wspólnych, jądrowych strukturach danych, natomiast w systemie Windows NT stosuje się obiektowe przekazywanie komunikatów między elementami jądra dotyczącymi wejścia-wyjścia. Przedstaw trzy argumenty za i trzy argumenty przeciw każdemu z tych podejść.

- 12.8** W jaki sposób bezpośredni dostęp do pamięci (DMA) zwiększa współbieżność? Jak wpływa on na złożoność sprzętu?
- 12.9** Napisz (w pseudokodzie) implementację zegarów wirtualnych, uwzględniając organizację kolejek i zarządzanie zamówieniami na odliczanie czasu w jądrze i aplikacjach. Przyjmij, że sprzęt ma trzy kanały czasomierzy.
- 12.10** Dlaczego możliwość zwiększania szybkości szyny systemowej i urządzeń jest ważna w przypadku wzrostu szybkości procesora?

Uwagi bibliograficzne

Dobry przegląd zagadnień wejścia-wyjścia i operacji sieciowych w systemie UNIX podaje Vahalia w książce [435]. Leffler i in. w książce [245]^{*} przedstawiają w szczegółach struktury i metody wejścia-wyjścia zastosowane w systemie UNIX BSD. Książka Milenkovica [293] zawiera omówienie złożoności metod wejścia-wyjścia i ich implementacji. Zastosowanie i programowanie różnorodnych protokołów komunikacji międzyprocesowej i sieciowej w systemie UNIX przedstawia Stevens w książce [408]. Brain w książce [51] dokumentuje interfejs aplikacji systemu Windows NT. Implementację wejścia-wyjścia w przykładowym systemie MINIX OS opisali w swej książce [418] Tanenbaum i Woodhull. Helen Custer w książce [90] zamieszcza szczegółowe informacje na temat implementacji operacji wejścia-wyjścia w systemie Windows NT, opartej na przekazywaniu komunikatów.

Do najlepszych źródeł zawierających szczegóły obsługi wejścia-wyjścia na poziomie sprzętowym oraz metod odwzorowań w pamięci należą podręczniki dotyczące procesorów firm Motorola [302] i Intel [194]. Hennessy i Patterson w książce [169] opisują systemy wieloprocesorowe i zagadnienia spójności pamięci. Tanenbaum w książce [415] przedstawia zasady projektowania urządzeń wejścia-wyjścia na niskim poziomie, a książka [371] Sargenta i Shoemakera stanowi przewodnik programisty niskopoziomowych właściwości sprzętu i oprogramowania komputerów osobistych typu PC. Odwzorowanie adresów urządzeń wejścia-wyjścia stosowanych w komputerach zgodnych ze standardem IBM PC zawiera podręcznik [186]. Wydanie [187] miesięcznika *IEEE Computer* jest poświęcone omówieniu zaawansowanego sprzętu i oprogramowania wejścia-wyjścia.

* Istnieje nowsza książka poświęcona tej tematyce, odnosząca się do systemu 4.4BSD – McKusick M. K., Bostic K., Karels M. J., Quarterman J. S.: *The Design and Implementation of the 4.4BSD Operating System*. Reading, MA, Addison-Wesley 1996. – Przyp. tłum.



Rozdział 13

STRUKTURA PAMIĘCI POMOCNICZEJ

Z logicznego punktu widzenia w systemie plików można wyodrębnić trzy części. W rozdziale 10 zapoznaliśmy się z interfejsem systemu plików przeznaczonym dla osób korzystających z jego usług oraz zajmujących się programowaniem. W rozdziale 11 omówiliśmy wewnętrzne struktury danych oraz algorytmy używane w systemie operacyjnym do implementowania tego interfejsu. W tym rozdziale omawiamy najniższy poziom systemu plików: strukturę pamięci pomocniczej. Najpierw opiszemy algorytmy planowania ruchu głowic, które określają porządek dyskowych operacji wejścia-wyjścia w celu zwiększenia wydajności. Następnie zajmiemy się formatowaniem dysku i zarządzaniem blokami rozruchowymi, blokami uszkodzonymi oraz obszarem wymiany. Na zakończenie zwróciimy uwagę na niezawodność dysków oraz na implementowanie pamięci trwałej.

13.1 ■ Struktura dysku

W nowoczesnych systemach komputerowych dyski dostarczają olbrzymiej ilości pamięci pomocniczej. Jako pierwsze nośniki pamięci pomocniczej były stosowane taśmy magnetyczne, lecz ich czas dostępu jest znacznie dłuższy niż dysków. Dlatego taśm używa się dzisiaj głównie w celach archiwalnych do składowania rzadko używanych informacji, jako nośników do przenoszenia informacji z jednego systemu do drugiego oraz do przechowywania tak wielkich ilości informacji, że nie opłaca się ich trzymać w systemach dyskowych. Więcej wiadomości na temat pamięci taśmowej podajemy w rozdz. 14.

Nowoczesne napędy dysków są adresowane niczym wielkie, jednowymiarowe tablice *bloków logicznych*, przy czym blok logiczny jest najmniejszą jednostką przesyłania. Rozmiar bloku logicznego wynosi zazwyczaj 512 B, chociaż niektóre dyski można formatować na *niskim poziomie*, wybierając różne rozmiary bloków logicznych, na przykład 1024 B. Możliwość tę omawiamy szerzej w p. 13.3.1.

Jednowymiarowa tablica bloków logicznych jest sekwencyjnie odwzorowywana na sektory dysku. Sektor 0 jest pierwszym sektorem na pierwszej ścieżce najbardziej zewnętrznego cylindra. Dalsze odwzorowanie odbywa się po kolej wzduż tej ścieżki, następnie wzduż pozostałych ścieżek cylindra, a potem postępuje w głąb następnych cylindrów – od najbardziej zewnętrznego do najbardziej wewnętrznego.

Za pomocą takiego odwzorowania powinna być możliwa zamiana numeru bloku logicznego na adres dyskowy w starym stylu, składający się z numeru cylindra, numeru ścieżki w obrębie tego cylindra i numeru sektora w obrębie ścieżki. W praktyce wykonywanie takiego tłumaczenia jest trudne z dwóch powodów. Po pierwsze, na dyskach trafiają się sektory uszkodzone, jednak omawiane odwzorowanie ukrywa to, zastępując sektory uszkodzone sektorami rezerwowymi, które mogą występować w dowolnym miejscu na dysku. Po drugie, liczba sektorów przypadających na ścieżkę nie jest stała. Im ścieżka jest dalej położona od środka dysku, tym większa jest jej długość, więc może zawierać więcej sektorów. Dlatego nowoczesne dyski są organizowane w grupy cylindrów (ang. *zones of cylinders*). Liczba sektorów na ścieżce jest stała w obrębie grupy. Jednak w miarę przemieszczania się od grupewnętrznych ku zewnętrznemu wzrasta liczba sektorów przypadających na ścieżkę. Ścieżki w najbardziej zewnętrznnej grupie na ogół zawierają o 40% więcej sektorów niż ścieżki w grupie najbardziej wewnętrznej.

Liczba sektorów na ścieżce wzrosła wraz z rozwojem technologii pamięci dyskowych i występowanie 100 sektorów na jednej ścieżce w zewnętrznzej grupie cylindrów jest rzeczą normalną. Z podobnych przyczyn wzrosła też liczba cylindrów przypadających na dysk i niczym niezwykłym nie jest dziś kilka tysięcy cylindrów na jednym dysku.

13.2 ■ Planowanie dostępu do dysku

Jednym z zadań systemu operacyjnego jest ekonomiczne użytkowanie sprzętu. W odniesieniu do napędów dysków oznacza to troskę o szybki dostęp do dysku i szybkie przesyłanie danych dyskowych. Na czas dostępu mają wpływ dwa ważne składniki (zob. też p. 2.3.2). Przez *czas szukania* (ang. *seek time*) rozumie się czas potrzebny na przemieszczenie ramienia dysku do pozycji,

w której głowice ustawiają się w cylindrze zawierającym potrzebny sektor. *Opóźnienie obrotowe* (ang. *rotational latency*) oznacza dodatkowy czas zużywany na obrót dysku do pozycji, w której potrzebny sektor trafia pod głowicę dysku. *Szerokość pasma* (ang. *bandwidth*) dysku nazywa się łączną liczbą przesyłanych bajtów, podzieloną przez łączny czas, jaki upływa od pierwszego zamówienia na usługę dyskową do chwili zakończenia ostatniego przesłania. Planując wykonywanie dyskowych operacji wejścia-wyjścia w odpowiednim porządku, możemy polepszać zarówno czas dostępu, jak i szerokość pasma.

Zgodnie z tym, co powiedzieliśmy w rozdz. 2, proces potrzebujący wykonać dyskową operację wejścia-wyjścia każdorazowo odwołuje się do systemu operacyjnego. W zamówieniu określa się kilka informacji:

- czy jest to operacja wejścia czy wyjścia;
- dyskowy adres przesyłania;
- adres pamięci operacyjnej dotyczący przesyłania;
- liczbę bajtów do przesłania.

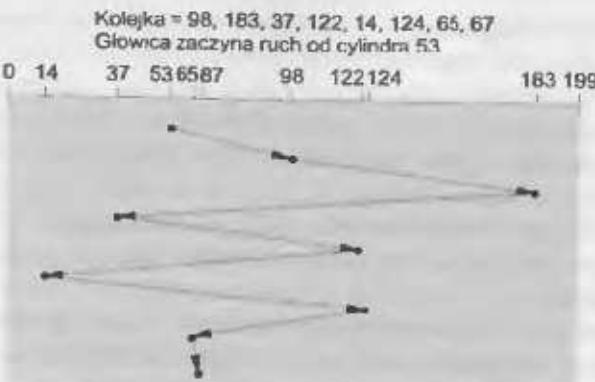
Jeżeli potrzebny napęd dysku i jego sprzętowy sterownik są gotowe do pracy, to zamówienie można spełnić natychmiast. Jeśli napęd lub sterownik są zajęte, to każde nowe zamówienie usługi będzie musiało być ustawione w kolejce zamówień oczekujących na dany napęd. W systemie wieloprogramowym z wieloma procesami kolejka dyskowa może często mieć po kilka czekających zamówień. Wobec tego po zakończeniu jednego zamówienia system operacyjny ma możliwość wyboru zamówienia, które zostanie obsłużone w następnej kolejności.

13.2.1 Planowanie metodą FCFS

Najprostszą metodą planowania dostępu do dysku jest – oczywiście – algorytm „pierwszy zgłoszony – pierwszy obsłużony” (ang. *first-come, first-served* – FCFS). Algorytm ten jest z natury sprawiedliwy, lecz na ogół nie zapewnia najszybszej obsługi. Rozważmy na przykład dyskową kolejkę zamówień na operacje wejścia-wyjścia odnoszące się do bloków w cylindrach

98, 183, 37, 122, 14, 124, 65, 67,

obsługiwianą w tej kolejności. Jeśli głowica dysku znajduje się początkowo w cylindrze 53, to najpierw przemieści od cylindra 53 do cylindra 98, a następnie do cylindrów 183, 37, 122, 14, 124, 65 i na koniec do cylindra 67, przechodząc łącznie 640 cylindrów. Plan ten widać na rys. 13.1.



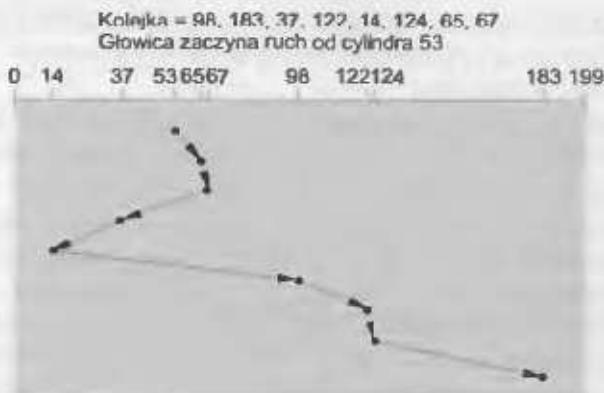
Rys. 13.1 Planowanie dostępu do dysku metodą FCFS

Wada tego zaplanowania ujawnia się w gwałtownych wychyleniach głowicy od cylindra 122 do 14 i z powrotem do 124. Gdyby zamówienia odnoszące się do cylindrów 37 i 14 można było obsłużyć razem, przed lub po zamówieniach na cylindry 122 i 124, to łączny ruch głowicy zmalałby istotnie, co mogłoby polepszyć wydajność.

13.2.2 Planowanie metodą SSTF

Wydaje się uzasadnione, aby dążyć do łącznej obsługi wszystkich zamówień sąsiadujących z bieżącym położeniem głowicy, zanim nastąpi jej przemieszczenie w odleglejsze rejony w celu realizacji innych zamówień. To założenie jest podstawą algorytmu „najpierw najkrótszy czas przeszukiwania” (ang. *shortest-seek-time-first* – SSTF). W algorytmie SSTF wybiera się zamówienie z minimalnym czasem przeszukiwania względem bieżącej pozycji głowicy. Ponieważ czas przeszukiwania wzrasta proporcjonalnie do liczby cylindrów odwiedzanych przez głowicę, w algorytmie SSTF wybiera się zamówienie najbliższe bieżącemu położeniu głowicy.

W naszej przykładowej kolejce zamówień najbliższe zamówienie względem początkowego położenia głowicy (53) dotyczy cylindra 65. Gdy znajdziemy się w cylindrze 65, wówczas następne najbliższe zamówienie odnosi się do cylindra 67. Z tamtego miejsca bliżej jest do zamówienia cylindra 37 niż do 98, więc zamówienie 37 będzie obsłużone najpierw. W dalszym ciągu obsługujemy zamówienie 14, potem 98, 122, 124 i na koniec 183 (rys. 13.2). Ta metoda planowania owocuje łącznym przemieszczeniem głowicy o zaledwie 236 cylindrów, czyli niewiele ponad jedną trzecią odległości koniecznej przy planowaniu metodą FCFS. Algorytm ten daje znaczne polepszenie wydajności.



Rys. 13.2 Planowanie dostępu do dysku metodą SSTF

Planowanie metodą SSTF jest w istocie odmianą planowania metodą „najpierw najkrótsze zadanie” (SJF) i – podobnie jak planowanie metodą SJF – może powodować głodzenie (ang. *starvation*) pewnych zamówień. Pamiętajmy, że zamówienia mogą nadchodzić w dowolnych chwilach. Założymy, że mamy w kolejce dwa zamówienia – odnoszące się do cylindra 14 i 186. Jeśli podczas obsługi zamówienia w cylindrze 14 nadejdzie zamówienie dotyczące cylindra bliskiego 14, to zostanie ono obsłużone jako następne, powodując dalsze oczekiwanie zamówienia do cylindra 186. Podeczas obsługi tego zamówienia może pojawić się kolejne zamówienie bliskie cylindrowi 14. Teoretycznie może nastąpić w tej sytuacji ciąg zamówień położonych w sąsiedztwie i spowodować nieskoronczone oczekiwanie zamówienia do cylindra 186. Szenariusz takí jest tym bardziej prawdopodobny, im dłuższa będzie się stawać kolejka nie obsłużonych zamówień.

Algorytm SSTF, choć znacznie lepszy od algorytmu FCFS, nie jest optymalny. W podanym przykładzie postąpilibyśmy lepiej, gdybyśmy przed przejściem do obsługi cylindrów 65, 67, 98, 122, 124 i 183, przemieścili głowicę z cylindra 53 na 37, pomimo że ten ostatni nie jest najbliższy pozycji 14. Taka strategia zmniejszyłaby ruch głowic do 208 cylindrów.

13.2.3 Planowanie metodą SCAN

W algorytmie SCAN ramię dysku rozpoczyna od jednej krawędzi dysku i przemieszcza się w kierunku krawędzi przeciwległej, obsługując zamówienia po osiągnięciu każdego kolejnego cylindra, aż dotrze do skrajnego cylin-

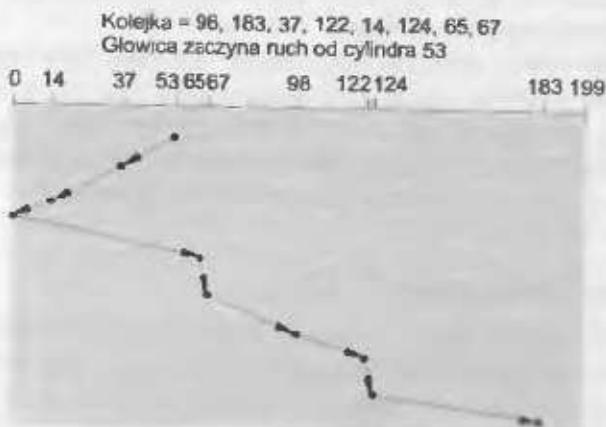
* Nazwa nie jest skrótem, lecz pochodzi od ang. *scan* (tu: *omiatać*). – Przyp. tłum.

dra. Wtedy zmienia się kierunek ruchu głowicy i obsługa jest kontynuowana. Głowica nieprzerwanie przeszukuje cały dysk tam i z powrotem. Znów posłużymy się naszym przykładem.

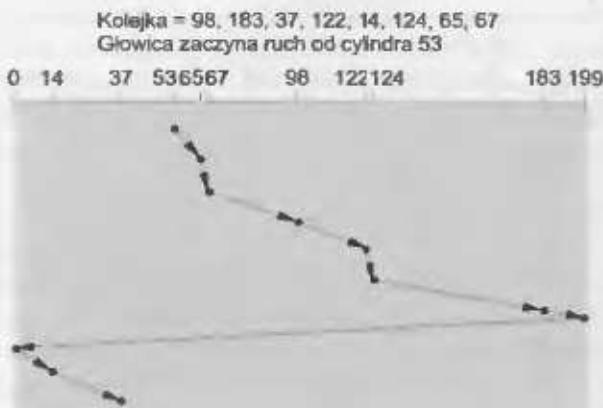
Przed zastosowaniem algorytmu SCAN do zaplanowania zamówień na cylindry 98, 183, 37, 122, 14, 124, 65 i 67, oprócz bieżącej pozycji głowicy (53), jest nam potrzebna znajomość kierunku jej ruchu. Jeśli ramię dysku przemieszcza się w kierunku cylindra 0, to głowica obsługuje najpierw zamówienia w cylindrach 37 i 14. Przy cylindrze 0 ramię zmieni kierunek ruchu i zacznie się przemieszczać w kierunku przeciwległej krawędzi dysku, obsługując zamówienia w cylindrach 65, 67, 98, 122, 124 i 183 (rys. 13.3). Jeśli w kolejce pojawiłyby się zamówienie odnoszące się do cylindra tuż przed głowicą, to zostałyby zrealizowane niemal natychmiast, natomiast zamówienie pojawiające się tuż za głowicą musiałoby poczekać na obsługę, aż głowica dojdzie do końca, zmieni kierunek i wróci.

Algorytm SCAN bywa czasami nazywany *algorytmem windy* (ang. *elevator algorithm*), ponieważ ramię dysku zachowuje się jak winda w budynku, obsługując najpierw wszystkie zamówienia w kierunku do góry, a potem zmieniając kierunek obsługi zamówień na przeciwny.

Przy założeniu równomiernego rozkładu zamówień na cylindry można rozważyć gęstość zamówień w chwili, gdy głowica osiąga skrajne położenie i zmienia kierunek. W takiej chwili będzie względnie mało zamówień w bezpośrednim sąsiedztwie głowicy, gdyż cylindry te zostały dopiero co obsłużone. Największe zagęszczenie zamówień dotyczyć będzie przeciwległego „konca”. Zamówienia te również najdłużej czekają, dlaczego by zatem nie pojść tam najpierw? Na tym polega pomysł następnego algorytmu.



Rys. 13.3 Planowanie dostępu do dysku metodą SCAN



Rys. 13.4 Planowanie dostępu do dysku metodą C-SCAN

13.2.4 Planowanie metodą C-SCAN

Algorytm C-SCAN (ang. *circular SCAN* – omijanie cykliczne) jest odmianą algorytmu SCAN zaprojektowaną w trosce o bardziej równomierny czas czekania. Tak jak w przypadku planowania metodą SCAN w algorytmie C-SCAN przesuwa się głowicę od krawędzi dysku do jego środka, obsługując napotykane po drodze zamówienia. Jednak gdy głowica osiągnie skrajne wychylenie na dysku, natychmiast wraca do przeciwnego położenia, bez podejmowania obsługi jakiegokolwiek zamówienia w drodze powrotnej (rys. 13.4). W algorytmie C-SCAN w istocie cylindry traktuje się jak listę cykliczną, na której ostatni cylinder spotyka się z pierwszym.

13.2.5 Planowanie metodą LOOK

Zauważmy, że zgodnie z podanym przez nas opisem zarówno w planowaniu SCAN, jak i C-SCAN głowica przemieszcza się zawsze od jednego skrajnego położenia na dysku do drugiego. W praktyce żaden z tych algorytmów nie jest implementowany w ten sposób. Najczęściej głowica przesuwa się do skrajnego zamówienia w każdym kierunku. Natychmiast potem głowica wykonuje zwrot, nie dochodząc do skrajnego położenia na dysku. Takie wersje algorytmów SCAN i C-SCAN nazywają się odpowiednio LOOK i C-LOCK, ponieważ przed kontynuowaniem ruchu „patrzy się” w nich, czy w danym kierunku znajduje się jakieś zamówienie (rys. 13.5).



Rys. 13.5 Planowanie dostępu do dysku metodą C-LOOK

13.2.6 Wybór algorytmu planowania dostępu do dysku

Jak, mając tyle algorytmów planowania dostępu do dysku, wybrać najlepszy z nich? Planowanie metodą SSTF jest dość powszechnie i wygląda naturalnie. Algorytmy planowania SCAN i C-SCAN są odpowiedniesze w systemach, w których jest dużo zamówień na operacje dyskowe, ponieważ mniejsze jest w nich prawdopodobieństwo występowania omówionego wcześniej problemu głodzenia. Dla dowolnej, konkretnej listy zamówień można zdefiniować optymalny porządek obsługi, ale wielkość obliczeń potrzebnych do planowania optymalnego może nie dać się usprawiedliwić uzyskanymi oszczędnościami w stosunku do metod planowania SSTF lub SCAN.

Wydajność każdego algorytmu planowania zależy jednak przede wszystkim od liczby i rodzaju zamówień. Przypuśćmy na przykład, że kolejka ma zazwyczaj tylko jedno nie obsłużone zamówienie. Wówczas wszystkie algorytmy planowania dadzą ten sam rezultat, gdyż mają tylko jeden wybór co do przesunięcia głowicy dysku. Zachowanie ich wszystkich byłoby takie samo jak przy planowaniu metodą FCFS.

Zauważmy też, że zamówienia na usługi dyskowe mogą w znacznym stopniu zależeć od metody przydziału pliku. Program czytający plik przydzielony w sposób ciągły wygeneruje kilka zamówień odnoszących się do sąsiednich miejsc na dysku, co ograniczy ruch głowicy. Natomiast plik listowy lub indeksowy może zawierać bloki szeroko rozrzucone po dysku, powodując większy ruch głowic.

Ważna jest również lokalizacja katalogów i bloków indeksowych. Ponieważ każdy plik musi być przed użyciem otwarty, a otwarcie pliku wymaga

przeszukania struktury katalogowej, więc dostępy do katalogów będą częste. Założymy, że wpis katalogowy znajduje się w pierwszym cylindrze, a dane pliku są w ostatnim cylindrze. W tym przypadku głowica musi przebyć całą szerokość dysku. Gdyby wpis katalogowy znajdował się w środkowym cylindrze, to głowica miałaby do pokonania co najwyżej połowę drogi. Podręczne przechowywanie katalogów i bloków indeksowych w pamięci głównej może również pomóc w zmniejszaniu ruchu ramienia dysku, szczególnie przy realizacji zamówień czytania.

Z uwagi na te czynniki algorytm planowania dostępu do dysku powinien być napisany jako osobny moduł systemu operacyjnego, tak aby można go było w razie konieczności zastąpić innym algorymem. Zarówno algorytm SSTF, jak i algorytm LOOK może z powodzeniem pełnić funkcję algorytmu obieranego domyślnie.

Zauważmy, że w przedstawionych tu algorytmach planowania bierze się pod uwagę tylko odległość szukania. W nowoczesnych dyskach opóźnienie obrotowe może być prawie tak samo duże jak średni czas szukania. Planowanie mające na celu poprawianie parametru opóźnienia obrotowego napotyka jednak w systemie operacyjnym trudności, ponieważ nowoczesne dyski nie ujawniają fizycznego położenia bloków logicznych. Producenci dysków zaradzili temu problemowi, implementując algorytmy planowania dostępu do dysku w sterowniku wbudowanym w sprzęt napędu dysku. Jeśli system operacyjny wyśle pakiet zamówień do sterownika, to zostaną one przez sprzęt ustawione w kolejkę i zaplanowane w sposób poprawiający zarówno czas szukania, jak i opóźnienie obrotowe. Gdyby wydajność operacji wejścia-wyjścia była jedyną sprawą do załatwienia, system operacyjny mógłby z powodzeniem przerzucić odpowiedzialność za planowanie dostępu do dysku na sprzęt. Jednak w praktyce system operacyjny może mieć do czynienia z innymi ograniczeniami na porządek obsługiwanego zamówienia. Stronicowanie na żądanie może na przykład wymagać pierwszeństwa nad operacjami wejścia-wyjścia dotyczącymi aplikacji, z kolei operacje pisania są pilniejsze niż operacje czytania, jeżeli w pamięci podręcznej zaczyna brakować wolnych stron. Może być także pożądane gwarantowanie porządku w zbiorze dyskowych operacji pisania, aby uodpornić system plików na wypadek załamania systemu. Zważmy, co by się stało, gdyby system operacyjny przydzielił stronę na dysku do pliku, a program użytkowy zdążył zapisać na niej dane zanim system operacyjny miałby szansę wprowadzić zmieniony i-węzeł oraz wykaz wolnych obszarów z powrotem na dysk. Aby dostosować się do takich wymagań, system operacyjny może realizować własne algorytmy planowania dostępu do dysku i „dawkować” zamówienia sterownikowi dysku jedno za drugim.

13.3 ■ Zarządzanie dyskiem

System operacyjny odpowiada również za kilka innych aspektów zarządzania dyskiem. Tutaj omówimy przygotowanie dysku do pracy, rozruch systemu z użyciem dysku oraz postępowanie z blokami błędymi.

13.3.1 Formatowanie dysku

Nowy dysk magnetyczny jest „nie zapisaną tablicą” – są to po prostu tarcze wykonane z materiału magnetycznego. Zanim na dysku będzie można zapamiętywać dane, należy go podzielić na sektory, które sterownik dysku potrafi czytać i zapisywać. Ten proces nazywa się *formatowaniem niskiego poziomu* (ang. *low-level formatting*) albo *formatowaniem fizycznym* (ang. *physical formatting*). Formatowanie niskiego poziomu polega na umieszczeniu specjalnej struktury danych we wszystkich miejscach na dysku odpowiadających sektorom. Struktura danych sektora zazwyczaj składa się z nagłówka, obszaru danych (na ogół o długości 512 B) i zakończenia. Nagłówek i zakończenie zawierają informacje, z których korzysta sprzętowy sterownik dysku, takie jak numer sektora i *kod korygujący* (ang. *error-correcting code* – ECC). Gdy sterownik zapisuje sektor danych podczas zwykłej operacji wejścia-wyjścia, wówczas kod ECC zostaje uaktualniony za pomocą wartości obliczonej na podstawie bajtów obszaru danych. Przy czytaniu sektora kod korygujący jest obliczany ponownie i porównywany z wartością zapamiętaną. Jeśli liczby – obliczona i zapamiętana – są różne, to wskazuje to na uszkodzenie obszaru danych sektora i rodzi podejrzenie, że dysk ma wadę w tym sektorze (zob. p. 13.3.3). Kod ECC jest samokorygujący, ponieważ zawiera dość informacji, aby w sytuacji, gdy tylko 1 lub 2 bity danych są uszkodzone, sterownik mógł te zmienione bity zidentyfikować i obliczyć ich poprawną wartość. Przetwarzanie ECC jest wykonywane przez sterownik automatycznie podczas każdego zapisywania lub czytania sektora.

Większość dysków twardych zostaje sformatowana na niskim poziomie już w fabryce – czynność ta jest częścią procesu ich wytwarzania. Formatowanie umożliwia producentowi przetestowanie dysku i naniesienie odwzorowania numerów bloków logicznych na wolne od błędów sektory dyskowe. Sterowniki wielu dysków twardych mogą w ramach formatowania niskiego poziomu być informowane, ile bajtów na obszar danych mają zostawić między nagłówkami a zakończeniami wszystkich sektorów. Można zazwyczaj wybierać między rozmiarami takimi jak 256, 512 lub 1024 B. Formatowanie dysku z użyciem większych sektorów oznacza, że na każdej ścieżce będzie się ich mieścić mniej, lecz oznacza to także, że mniej będzie na ścieżce nagłówków i zakończeń sektorów, wskutek tego wzrasta ilość miejsca dostępnego

dla danych użytkownika. W niektórych systemach operacyjnych można stosować tylko sektory o długości 512 B.

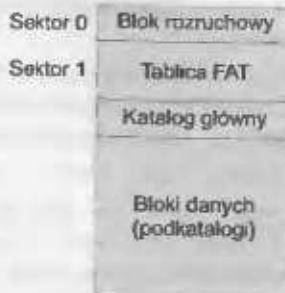
Aby zastosować dysk do przechowywania plików, system operacyjny musi jeszcze zapisać na nim własne struktury danych. Wykonuje to w dwu etapach. Pierwszy etap polega na podzieleniu dysku na jedną lub więcej grup cylindrów. System operacyjny może traktować każdą z tak powstałych stref (tzw. partycji; ang. *partition*) jak osobny dysk. W jednej ze stref można na przykład utrzymywać kopię wykonywalnego kodu systemu operacyjnego, a w innej – pliki użytkownika. Po podzieleniu dysku następuje etap drugi, nazywany *formatowaniem logicznym* lub „tworzeniem systemu plików”. W tej fazie system operacyjny zapamiętuje na dysku początkowe struktury danych systemu plików. W skład tych struktur może wchodzić mapa wolnych i przydzielonych obszarów (tablica FAT lub i-węzły) oraz początkowy, pusty katalog.

Niektóre systemy operacyjne uprawniają specjalne programy do traktowania strefy dysku jako wielkiej, liniowej tablicy bloków logicznych, bez żadnych struktur danych systemu plików. Możliwość działania na tej tablicy jest czasem nazywana *surowym* (ang. *raw*) wejściem-wyjściem. Z operacji surowego wejścia-wyjścia korzystają na przykład pewne bazy danych, gdyż umożliwia im to dokładne określanie miejsca przechowywania na dysku poszczególnych rekordów bazy. Surowe operacje wejścia-wyjścia są wykonywane z pominięciem wszelkich usług systemu plików, takich jak bufore podrzczne, wstępne sprawdzanie, przydział przestrzeni, nazwy plików i katalogi. Implementując indywidualnie specjalne usługi magazynowania informacji z użyciem surowej strefy, możemy zwiększyć wydajność pewnych aplikacji, lecz większość aplikacji działa lepiej z użyciem zwyczajnych usług systemu plików.

13.3.2 Blok rozruchowy

Komputer, który rozpoczyna pracę – na przykład w chwili włączenia go do sieci zasilającej lub podczas wznowiania działania – musi wykonać jakiś program wstępny. Ten wstępny *program rozruchowy* (ang. *bootstrap*) jest zazwyczaj prosty. Ustawia on stan początkowy wszystkich elementów systemu – od procesora aż po sterowniki urządzeń i zawartość pamięci – po czym uruchamia system operacyjny. Aby wykonać swoje zadanie, program rozruchowy znajduje jądro systemu operacyjnego na dysku, umieszcza je w pamięci i wykonuje skok pod adres początkowy w celu rozpoczęcia wykonywania systemu operacyjnego.

W większości komputerów program rozruchowy jest przechowywany w pamięci przeznaczonej tylko do czytania (ang. *read only memory* – ROM). Jest to wygodne miejsce, ponieważ pamięć ROM nie wymaga żadnych zabiegów wstępnych i nie zmienia swojej lokalizacji, aby procesor mógł zacząć



Rys. 13.6 Układ informacji na dysku MS-DOS

działanie po włączeniu zasilania lub ponownym rozruchu. Ponadto ze względu na to, że pamięć ROM można tylko czytać, nie jest ona narażona na zainfekowanie wirusem komputerowym. Kłopot polega natomiast na tym, że zmiana kodu programu rozruchowego wymaga wymiany układów scalonych pamięci ROM. Z tego powodu większość systemów przechowuje małutki program ładowania programu rozruchowego (ang. *bootstrap loader*) w rozruchowej pamięci ROM. Jedynym zadaniem tego programu jest sprowadzenie pełnego programu rozruchowego z dysku. Pełny program rozruchowy można łatwo wymienić – nową wersję zapisuje się po prostu na dysku. Pełny program rozruchowy jest przechowywany w strefie nazywanej *blokami rozruchowymi* (ang. *boot blocks*) w ustalonym miejscu na dysku. Dysk mający strefę rozruchową zwie się *dyskiem rozruchowym* (ang. *boot disk*) lub *systemowym*.

Kod zawarty w rozruchowej pamięci ROM poleca sterownikowi dysku przeczytać bloki rozruchowe do pamięci (w tym czasie nie ładuje się żadnych modułów sterujących urządzeniami), a następnie rozpoczyna wykonywanie tego kodu. Pełny program rozruchowy jest bardziej skomplikowany niż jego program ładowający w rozruchowej pamięci ROM i potrafi załadować cały system operacyjny z dowolnego miejsca na dysku oraz rozpoczęć jego działanie. Niezależnie od tych funkcji pełny program rozruchowy może być mały. Na przykład program rozruchowy systemu MS-DOS zajmuje tylko jeden 512-bajtowy blok (rys. 13.6).

13.3.3 Bloki uszkodzone

Ponieważ dyski mają części ruchome i małe tolerancje* (pamiętajmy, że głowica dysku unosi się tuż nad jego powierzchnią), więc są podatne na awarie. Czasami

* Granice dozwolonych odchyлеń parametrów działania. — Przyp. tłum.

awaria jest zupełna i dysk wymaga wymiany, a jego zawartość musi być odtworzona z nośników archiwalnych na nowym dysku. Znacznie częściej uszkodzeniu ulega jeden lub więcej sektorów. Większość dysków już z fabryki przychodzi z *uszkodzonymi blokami* (ang. *bad blocks*). W zależności od dysku i zastosowanego w nim sterownika z blokami tymi postępuje się w różny sposób.

Na nieskomplikowanych dyskach, takich jak niektóre dyski ze sterownikami IDE, uszkodzone bloki eliminuje się ręcznie. Na przykład polecenie **format** systemu MS-DOS wykonuje formatowanie logiczne, a także – jako część tego procesu – analizuje dysk w poszukiwaniu wadliwych bloków. Jeżeli polecenie **format** napotka blok uszkodzony, to zapisuje w odpowiedniej pozycji tablicy FAT specjalną wartość, aby zaznaczyć, że procedury przydzielania miejsca na dysku nie powinny go używać. Jeżeli bloki ulegają uszkodzeniu podczas normalnego działania, to należy ręcznie nakazać wykonanie specjalnego programu (np. **chkdsk**) w celu odnalezienia uszkodzonych bloków i odcięcia do nich dostępu, tak jak poprzednio. Dane zapisane w blokach uszkodzonych są zazwyczaj tracone.

W bardziej złożonych dyskach, w rodzaju dysków SCSI stosowanych w wysokiej klasy komputerach PC i w większości stacji roboczych, z uszkodzonymi blokami postępuje się w sposób bardziej elegancki. Wykaz uszkodzonych bloków na dysku utrzymuje sterownik danego dysku. Wykaz ten jest zakładany podczas formatowania niskiego poziomu wykonywanego w fabryce i jest aktualniany przez cały okres eksploatacji dysku. Przy formatowaniu niskiego poziomu zostawia się również pewien zapas sektorów niewidocznych dla systemu operacyjnego. Sterownikowi można nakazać logiczne zastąpienie każdego uszkodzonego sektora za pomocą sektora pobranego ze zbioru sektorów zapasowych. Schemat ten jest znany pod nazwą *metody sektorów zapasowych lub przenoszenia sektorów* (ang. *sector sparing or forwarding*).

Typowe transakcja dotycząca wystąpienia uszkodzonego sektora może wyglądać następująco:

- System operacyjny próbuje czytać blok logiczny o numerze 87.
- Sterownik oblicza kod ECC i wykrywa uszkodzenie sektora. Melduje o tym odkryciu systemowi operacyjnemu.
- Podczas następnego rozruchu systemu wykonuje się specjalne polecenie nakazujące sterownikowi SCSI zastąpić uszkodzony sektor sektorem zapasowym.
- Po wykonaniu tych czynności każde odwołanie systemu do bloku 87 zostanie przetłumaczone przez sterownik na adres sektora zapasowego.

Zwrócmy uwagę, że takie przeadresowanie wykonane przez sterownik może zniweczyć każdą optymalizację (!) poczynioną przez systemowy algorytm

planowania dostępu do dysku. Z tego powodu większość dysków formatuje się tak, aby pozostawić po małym zapasie sektorów na każdym cylindrze oraz osobny cylinder zapasowy. Podczas odwzorowywania uszkodzonego bloku sterownik stara się w miarę możliwości użyć sektora z zapasu znajdującego się na tym samym cylindrze.

Niektóre sterowniki zamiast korzystania z sektorów zapasowych mogą wykonywać polecenia zastąpienia uszkodzonego bloku metodą *przeciągania sektorów* (ang. *sector slipping*)^{*}. Oto przykład: Założmy, że blok logiczny o numerze 17 ulega uszkodzeniu, a pierwszy sektor zapasowy występuje po sektorze 202. Wówczas przeciągnięcie sektorów spowoduje nowe odwzorowanie wszystkich sektorów o numerach od 17 do 202 w ten sposób, że zostaną one przesunięte o jedno miejsce. Tak więc sektor 202 zostanie przekopiowany na miejsce zapasowe, następnie sektor 201 przejdzie na miejsce sektora 202 itd. aż do sektora 18, który zajmie miejsce sektora 19. Przeciąganie sektorów tą metodą doprowadzi do zwolnienia sektora 18, dzięki czemu sektor 17 można będzie odwzorować na jego miejscu.

Na ogół zastępowanie wadliwego bloku nie jest czynnością całkowicie automatyczną, ponieważ dane w uszkodzonym bloku są zazwyczaj tracone. Pociąga to za sobą konieczność naprawienia pliku, w którym blok ten był użyty (np. przez zrekonstruowanie go z taśmy zawierającej jego kopię zapasową), a to wymaga zabiegów ręcznych.

13.4 ■ Zarządzanie obszarem wymiany

Zarządzanie obszarem wymiany jest kolejnym niskopoziomowym zadaniem systemu operacyjnego. Pamięć wirtualna korzysta z przestrzeni dyskowej jako z rozszerzenia pamięci głównej. Ponieważ dostęp do dysku jest znacznie wolniejszy niż do pamięci głównej, zastosowanie obszaru wymiany ma duży wpływ na wydajność systemu. Podstawowym celem w projektowaniu obszaru wymiany jest umożliwienie najlepszej przepustowości systemowej pamięci wirtualnej. W tym punkcie omawiamy sposób użycia obszaru wymiany, jego umiejscowienie na dysku oraz sposób, w jaki się nim zarządza.

13.4.1 Wykorzystanie obszaru wymiany

Obszar wymiany jest używany na wiele sposobów przez różne systemy operacyjne, zależnie od zrealizowanych algorytmów zarządzania pamięcią. Na przykład systemy implementujące wymianę mogą korzystać z obszaru wy-

* Dosłownie: przesuwanie się sektora. – Przyp. tłum.

miany do przechowywania obrazów całych procesów, włącznie z ich kodem i segmentami danych. Systemy stronicujące mogą po prostu przechowywać strony usunięte z pamięci głównej. Ilość przestrzeni wymiany potrzebnej w systemie może się więc zmieniać w zależności od ilości pamięci fizycznej, ilości pamięci wirtualnej, którą należy składować, oraz sposobu wykorzystania pamięci wirtualnej. Ilość ta może się ważyć w przedziale od kilku megalabajtów do setek megalabajtów lub więcej.

Niektóre systemy operacyjne, takie jak UNIX, umożliwiają stosowanie wielu obszarów wymiany. Te obszary są zazwyczaj rozmieszczane na różnych dyskach, zatem obciążenie systemu wejścia-wyjścia operacjami stronicowania i wymiany można rozkładać między należące do systemu urządzenia wejścia-wyjścia.

Zauważmy, że bezpieczniej jest nadmiernie oszacować wielkość obszaru wymiany niż oszacować ją w stopniu niewystarczającym, bo gdyby systemowi zabrakło obszaru wymiany, wówczas mógłby on być zmuszony do zaniechania procesów lub nawet mógłby ulec awarii. W wyniku przeszacowania dochodzi do marnowania miejsca na dysku, które można by spożytkować na pliki, lecz nie powoduje to innych dolegliwości.

13.4.2 Umiejscowienie obszaru wymiany

Obszar wymiany może rezydować w dwu miejscach: przestrzeń wymiany można wyciąć ze zwykłego systemu plików lub może się ona znajdująć w osobnej strefie dyskowej. Jeżeli obszar wymiany jest po prostu wielkim plikiem w obrębie systemu plików, to do jego utworzenia, nazwania i przydzielenia mu miejsca można użyć zwykłych procedur systemu plików. Jest to zatem podejście łatwe do realizacji. Niestety, jest ono również mało wydajne. Nawigowanie w strukturze katalogowej i strukturach danych związanych z przydzielaniem miejsca na dysku jest czasochłonne i wymaga (potencjalnie) dodatkowych kontaktów z dyskiem. Fragmentacja zewnętrzna może znacznie wydłużyć czas wymiany, powodując konieczność wielokrotnych przeszukań podczas czytania lub zapisywania obrazu procesu. Wydajność można poprawić, przechowując podręcznie w pamięci operacyjnej informacje o położeniu bloków oraz stosując specjalne narzędzia do przydzielania obszaru wymiany jako fizycznie ciągłych grup bloków. Jednak wciąż będzie pozostawał koszt przechodzenia przez struktury danych systemu plików.

Znacznie częściej obszar wymiany jest tworzony w osobnej strefie dyskowej. Nie lokalizuje się w niej żadnego systemu plików ani nie buduje struktury katalogowej. Zamiast tego do przydzielania i zwalniania bloków stosuje się zarządcę pamięci obszaru wymiany. Nie optymalizuje on zużycia pamięci, lecz korzysta z algorytmów optymalizowanych ze względu na szyb-

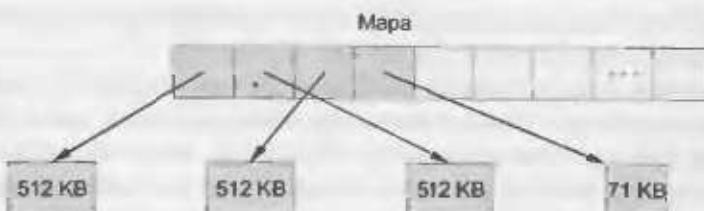
kosc. Wewnętrzna fragmentacja może rosnąć, lecz jest to kompromis do przyjęcia, ponieważ w obszarze wymiany dane pozostają przez znacznie krótsze okresy niż pliki w systemie plików, natomiast dostęp do obszaru wymiany może być znacznie częstszy. Niestety, takie podejście powoduje utworzenie podczas dzielenia dysku na strefy obszaru wymiany o stałej wielkości. Zwiększenie ilości miejsca w obszarze wymiany wymaga więc powtórnego podziału dysku na strefy (co pociąga za sobą przemieszczanie lub zniszczenie i odtwarzanie innych stref z systemami plików z kopii rezerwowych). Nowy obszar wymiany można też dodać gdziekolwiek do systemu.

Niektóre systemy operacyjne są elastyczne i mogą dokonywać wymian zarówno w strefach surowych, jak i w przestrzeni systemu plików. Przykładem jest system Solaris. Kwestię polityki oddzielono w nim od implementacji, co umożliwia administratorowi maszyny decydować o wyborze rodzaju obszaru wymiany. Kompromis zawiera się między wygodą przydzielania miejsca i zarządzania systemem plików a wydajnością wymian w surowych strefach.

13.4.3 Zarządzanie obszarem wymiany

Aby zilustrować metody stosowane do zarządzania obszarem wymiany, przyjrzymy się rozwojowi metod wymiany i stronicowania w systemie UNIX. W systemie UNIX (omawiamy go w pełni w rozdz. 21) implementowanie wymiany rozpoczęto od kopiowania całych procesów z pamięcią operacyjną do ciągłych obszarów dysku i z powrotem. Wraz z pojawieniem się sprzętu stronicującego system UNIX wyewoluował w kierunku kombinacji wymiany i stronicowania.

W systemie 4.3BSD uruchamianemu procesowi przydziela się obszar wymiany. W pamięci pomocniczej rezerwuje się miejsce w ilości wystarczającej na pomieszczenie programu (ten obszar nazywa się *stronami tekstu* (ang. *text pages*) lub *segmentem tekstu* (ang. *text segment*)) i segmentu danych procesu. Dzięki wstępнемu przydziałowi całej potrzebnej przestrzeni z reguły chroni się proces przed brakiem obszaru wymiany podczas działania. Gdy rozpoczyna się wykonywanie procesu, wówczas strony jego tekstu zostają sprowadzone z systemu plików. Te strony są w razie potrzeby wysyłane do obszaru wymiany i są z niego czytane z powrotem, więc kontakt z systemem plików przypada tylko po razie na każdą stronę tekstu. Strony z segmentu danych są czytane z systemu plików lub są tworzone (jeśli nie zawierają wartości początkowych) i podlegają wymianie i sprowadzaniu, stosownie do potrzeb. Jednym z ulepszeń (np. w sytuacji, gdy dwu użytkowników korzysta z tego samego edytora) jest wspólne użytkowanie stron tekstu identycznych w kilku procesach – zarówno w pamięci fizycznej, jak i w obszarze wymiany.

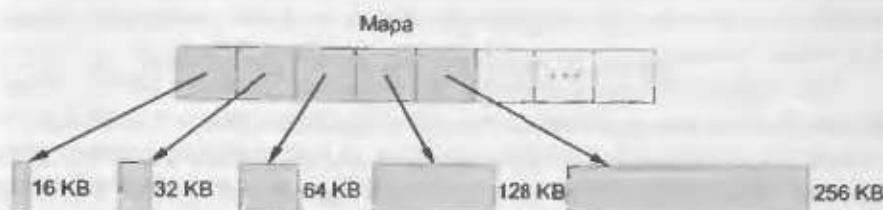


Rys. 13.7 Mapa wymiany segmentu tekstu w systemie 4.3BSD

Jądro korzysta z dwóch procesowych *map wymiany* (ang. *swap maps*) w celu doglądania użytkowania obszaru wymiany. Segment tekstu ma ustalony rozmiar, więc jego obszar wymiany jest przydzielany kawałkami po 512 KB, z wyjątkiem ostatniego kawałka, wymierzanego w przyrostach 1 K, w którym przechowuje się resztę stron (rys. 13.7).

Mapa wymiany segmentu danych jest bardziej skomplikowana, ponieważ segment danych może rosnąć z upływem czasu. Mapa ma ustalony rozmiar, ale zawiera adresy bloków wymiany o zmiennych rozmiarach. Blok wskazywany przez wpis w mapie wymiany z pozycji i ma rozmiar $2^i \times 16$ KB, przy czym jego maksymalna wielkość może wynieść 2 MB. Ta struktura danych jest zilustrowana na rys. 13.8. (Minimalne wartości bloków są określone przez zmienne, których wartości można zmienić przy wznawianiu pracy systemu). Jeśli proces próbuje powiększyć segment danych poza ostatnio przydzielony blok w obszarze wymiany, to system operacyjny przydziela mu nowy blok – dwa razy większy niż blok przydzielony poprzednio. Dzięki takiemu postępowaniu małe procesy używają małych bloków. Zmniejsza się również fragmentacja. Bloki wielkich procesów można szybko odnajdywać, a mapa wymiany pozostaje mała.

Konstruktorzy systemu Solaris 1 (SunOS 4) wprowadzili do standardowych metod uniksowych modyfikacje mające polepszyć wydajność i uwzględnić zmiany technologiczne. Podczas wykonywania procesu strony segmentu tekstu są sprowadzane z systemu plików do pamięci operacyjnej, w której się je udostępnia i z której są usuwane, jeśli zostaną do tego zakwalifikowane. Po-



Rys. 13.8 Mapa wymiany segmentu danych w systemie 4.3BSD

wtórne czytanie strony z systemu plików jest wydajniejsze niż zapisywanie jej w obszarze wymiany i czytanie jej stamtąd.

W systemie Solaris 2 dokonano dalszych zmian. Największa z nich polega na tym, że system Solaris 2 przydziela obszar wymiany tylko wówczas, gdy strona jest wyrzucana z pamięci fizycznej, a nie podczas pierwszego utworzenia strony pamięci wirtualnej. Dzięki tej zmianie wzrasta wydajność nowoczesnych komputerów, które mają więcej pamięci operacyjnej niż starsze systemy i charakteryzują się mniejszą intensywnością stronicowania.

13.5 ■ Niezawodność dysku

Dyski zawsze były najbardziej zawodną częścią systemu. Nadal charakteryzują się one względnie dużą awaryjnością, a ich awarie powodują utratę danych i długie przestoje związane z wymianą dysku i odtwarzaniem danych. Rekonstrukcja po awarii dysku może trwać godzinami, ponieważ odnajdywanie na taśmach kopii rezerwowych utraconych danych i proces ich odtwarzania na nowym dysku zajmuje wiele czasu. Odtworzony dysk z reguły nie jest wiernym obrazem dysku zepsutego, gdyż wszelkie zmiany danych wykonane po ostatnim składowaniu są utracone. Z tego powodu powiększanie niezawodności systemów dyskowych jest ważnym przedmiotem badań.

Zaproponowano kilka ulepszeń metod eksploatacji dysków. Wśród tych metod znajdują się takie, które organizują kilka dysków do pracy kooperatywnej. W celu powiększenia szybkości stosuje się metodę *paskowania dysku* (ang. *disk striping*), inaczej – *przeplotu*, w której grupa dysków jest traktowana jako jedna jednostka pamięci. Każdy blok danych jest podzielony na kilka podbloków, a każdy podblok jest pamiętany na innym dysku. Czas potrzebny na przesłanie bloku do pamięci skraca się niezwykle, ponieważ wszystkie dyski przesyłają swoje podbloki równolegle. Jeżeli obroty dysków są zsynchronizowane, to wydajność polepsza się jeszcze bardziej, gdyż wszystkie dyski są gotowe do przesyłania podbloków w tym samym czasie, więc czekanie powodowane najdłuższym opóźnieniem obrotowym zostaje wyeliminowane. Im większa liczba „pociętych na paski” dysków współpracuje ze sobą, tym większa jest łączna szybkość przesyłania osiągana w takim systemie.

Taka organizacja jest zazwyczaj nazywana nadmiarową tablicą niezależnych dysków (ang. *redundant array of independent disks* – RAID). Oprócz zwiększenia wydajności rozmaite schematy RAID polepszają także niezawodność wskutek pamiętania nadmiarowych danych. Nadmiarowość (redundancję) można organizować różnymi sposobami, charakteryzującymi się różną wydajnością i kosztem.

W najprostszej organizacji typu RAID, nazywanej *odbiciem lustrzanym* (ang. *mirroring*) lub *tworzeniem cienia* (ang. *shadowing*), utrzymuje się po prostu kopię każdego dysku. Rozwiążanie to jest drogie, ponieważ do pamiętania tej samej ilości danych stosuje się dwukrotnie więcej dysków. Jednak zapewnia ono też około dwa razy szybsze czytanie, ponieważ do każdego dysku można kierować połowę zamówień czytania. W innej organizacji RAID, nazywanej *przeplataniem bloków parzystości* (ang. *block interleaved parity*), używa się znacznie mniej冗undancji. Mały ulamek obszaru dysku jest wykorzystywany do przechowywania bloków parzystości. Założymy na przykład, że w tablicy jest dziewięć dysków. Wówczas dla każdych ośmiu bloków danych pamiętanych w tablicy zapamiętuje się również jeden blok parzystości. Każdy bit w bloku parzystości będzie zawierać parzystość odnoszącą się do odpowiednich pozycji bitów w każdym z ośmiu bloków danych. Odbywa się to podobnie jak obliczanie dziewiątego bitu parzystości w pamięci głównej dla każdego 8-bitowego bajtu. Jeśli jeden z bloków dyskowych ulegnie uszkodzeniu, to wszystkie jego bity danych są praktycznie zatarte. Powstaje *błąd zatarcia danych* (ang. *erasure error*). Bity te można jednak obliczyć na nowo na podstawie innych bloków danych oraz bloku parzystości. Zatem awaria jednego dysku nie powoduje już utraty danych.

Szybkość systemu RAID z kontrolą parzystości wynika z połączenia wielu dysków i ich sterowników. Jednak wydajność jest wystawiana na próbę podczas pisania, ponieważ uaktualnienie dowolnego, pojedynczego podbloku danych zmusza do ponownego obliczenia i zapisania bloku parzystości. Rozmieszczenie podbloków parzystości na wszystkich dyskach, zamiast przeznaczania na dane dotyczące parzystości osobnego dysku, umożliwia równomierne rozłożenie obciążen. Ponieważ wynikowa szybkość, koszt i niezawodność techniki RAID z kontrolą parzystości wydają się oticujące, wiele firm produkuje sprzęt i oprogramowanie RAID. Jeśli w systemie występują setki dysków, to awarie mogą powodować utratę danych kilka razy w roku. Dzięki zastosowaniu do rekonstrukcji po awariach nadmiarowych danych na dyskach RAID parametr ten polepsza się – jedna awaria występuje na kilkadziesiąt lat.

13.6 ■ Implementowanie pamięci trwałej

W rozdziale 6 przedstawiliśmy pojęcie rejestru zapisów wyprzedzających (ang. *write-ahead log*)^{*}, który wymagał dostępności pamięci trwałej. Informacje przechowywane w pamięciowej nigdy – z mocy definicji – nie ulegają

* Spotyka się również termin *lista zamiarów* (ang. *intention list*). – Przyp. tłum.

utracie. Aby zrealizować taką pamięć, musimy z wielokrotnią potrzebne informacje na wielu urządzeniach pamięci, niezależnych od siebie pod względem awaryjności. Zapisywanie uaktualnień należy koordynować w taki sposób, aby awaria zaistniała w trakcie aktualizacji nie spowodowała uszkodzenia wszystkich kopii, a usuwanie jej skutków umożliwiałoby przywrócenie wszystkim kopiom wartości spójnych i poprawnych nawet wówczas, gdyby podczas rekonstrukcji doszło do następnej awarii. W pozostałej części tego punktu pokazujemy, jak można spełnić te wymagania.

Operacja pisania na dysku kończy się na jeden z trzech sposobów:

- **Pomyślne zakończenie:** dane zapisano na dysku poprawnie.
- **Awaria częściowa:** podczas przesyłania wystąpił błąd, wskutek czego tylko w części sektorów zapisano nowe dane, a sektor zapisywany w chwili awarii mógł ulec uszkodzeniu.
- **Awaria całkowita:** do uszkodzenia doszło przed rozpoczęciem pisania na dysku, więc poprzednie wartości danych dyskowych pozostały nie tknięte.

Oczekujemy, że system wykryje każdą awarię występującą podczas zapisywania bloku i wywoła procedurę rekonstrukcji w celu przywrócenia blokowi spójnego stanu. W tym celu dla każdego bloku logicznego system musi utrzymywać dwa bloki fizyczne. Operacja wyjścia przebiega następująco:

1. Informacje są zapisywane w pierwszym bloku fizycznym.
2. Po pomyślnym zakończeniu pierwszego pisania te same informacje zapisuje się w drugim bloku fizycznym.
3. Operację uważa się za zakończoną tylko wtedy, kiedy drugie pisanie zakończy się pomyślnie.

Podeczas usuwania skutków awarii sprawdza się każdą parę bloków fizycznych. Jeśli oba są takie same i nie udało się w nich wykryć żadnych błędów, to nie są potrzebne żadne dalsze działania. Blok, w którym wykryto błąd, zastępuje się zawartością drugiego bloku. Jeżeli oba bloki nie zawierają wykrywalnych błędów, lecz ich zawartości są różne, to zawartość pierwszego bloku zastępujemy zawartością drugiego bloku. Taka procedura odtwarzania zapewnia, że zapisywanie pamięci trwalej kończy się pełnym powodzeniem albo nie następują żadne zmiany.

Procedurę tę można łatwo rozszerzyć na dowolną liczbę kopii każdego bloku w pamięci trwalej. Chociaż duża liczba kopii jeszcze bardziej zmniejsza

prawdopodobieństwo awarii, zazwyczaj jako rozsądne rozwiązanie przyjmuje się symulowanie pamięci trwałej za pomocą tylko dwóch kopii. Dane w pamięci trwałej są bezpieczne dopóty, dopóki nie nastąpi awaria, która zniszczy obie kopie.

13.7 ■ Podsumowanie

Napędy dysków są ważnymi urządzeniami wejścia-wyjścia pamięci pomocniczej w większości komputerów. Zamówienia na operacje dyskowe są generowane przez system plików oraz przez system pamięci wirtualnej. Każde zamówienie określa w postaci numeru bloku logicznego adres na dysku, do którego ma nastąpić odniesienie.

Algorytmy planowania dostępu do dysku mogą poprawić wynikową przepustowość oraz średni czas odpowiedzi i jego wariancję. Stosując algorytmy takie jak SSTF, SCAN, C-SCAN, LOOK i C-LOOK, można uzyskać te ulepszenia przez zmianę kolejności zamówień w celu zmniejszenia łącznego czasu przeszukiwania dysku.

Niekorzystny wpływ na działanie systemu ma fragmentacja zewnętrzna. Jednym ze sposobów reorganizacji dysku w celu zmniejszenia fragmentacji jest wykonanie składowania i odtworzenia zawartości całego dysku lub jego strefy. Czyta się rozrzucone po dysku bloki danych i odtwarza przez zapisanie ich z powrotem w sposób ciągły. Niektóre systemy dysponują narzędziami, za pomocą których można przeanalizować system plików w celu wykrycia pokakowanych plików. Programy takie przemieszczają następnie bloki w celu zmniejszenia fragmentacji. Zlikwidowanie fragmentacji szkodliwie pofragmentowanego systemu plików może znacznie polepszyć wydajność, ale podczas przebiegu procesu scalania system jest z reguły nie do użycia. Złożone systemy plików, w rodzaju uniksowego, szybkiego systemu plików (ang. *Fast File System – FFS*), zawierają wiele strategii panowania nad fragmentacją podczas przydziału miejsca na dysku: reorganizacja dysku staje się więc zbędna.

System operacyjny zarządza blokami dyskowymi. Dysk musi być najpierw sformatowany na niskim poziomie, aby na surowym nośniku danych utworzyć sektory. Następnie można dysk podzielić na strefy i utworzyć na nim system plików, jak również wydzielić bloki rozruchowe do zapamiętania programu rozruchu systemu. System musi mieć też możliwość wykreślania bloków lub logicznego zastępowania ich zapasowymi blokami na wypadek uszkodzenia bloku.

Ponieważ wydajny obszar wymiany jest kluczem do dobrego działania, systemy na ogół obchodzą system plików i korzystają z surowej strefy dyskowej w celu przechowywania stron pamięci nie mieszczących się w pamięci

fizycznej. W niektórych systemach używa się w tym celu pliku z systemu plików, jednak może się to odbić na wydajności. W innych systemach zezwala się użytkownikowi lub administratorowi na podejmowanie decyzji w tej sprawie, dostarczając mu możliwości wymiany.

Schemat utrzymywania rejestru zapisów wyprzedzających wymaga dostępności pamięci trwałej. Aby zrealizować taką pamięć, należy zwielokrotnić informacje na wielu nieulotnych urządzeniach pamięci (zwykle są to dyski), które w wypadku awarii są od siebie niezależne. Uaktualnianie informacji musi również przebiegać w sposób kontrolowany, aby zapewnić, że trwałe dane będzie można zrekonstruować po każdej awarii występującej podczas przesyłania lub rekonstrukcji.

■ Ćwiczenia

13.1 Z wyjątkiem algorytmu FCFS żaden z algorytmów planowania dostępu do dysku nie jest naprawdę sprawiedliwy (może wystąpić głodzenie).

- (a) Wyjaśnij, dlaczego powyższe stwierdzenie jest prawdziwe.
- (b) Opisz sposób zmodyfikowania algorytmów takich jak SCAN w celu zapewnienia ich sprawiedliwego działania.
- (c) Wyjaśnij, dlaczego sprawiedliwość algorytmu jest ważna w systemie czasu rzeczywistego.
- (d) Podaj co najmniej trzy przykłady sytuacji, w których jest istotne, aby system operacyjny nie postępował „sprawiedliwie” przy obsłudze operacji wejścia-wyjścia.

13.2 Przypuśćmy, że napęd dysku ma 5000 cylindrów ponumerowanych od 0 do 4999. W danej chwili napęd obsługuje zamówienie w cylindrze 143, a poprzednie zamówienie dotyczyło cylindra 125. Kolejka oczekujących zamówień w porządku FIFO przedstawia się następująco:

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Poczytając od bieżącego położenia głowicy, określ łączny dystans (wyrażony liczbą cylindrów), który przebywa ramię dysku w celu spełnienia wszystkich oczekujących zamówień dla każdego z następujących algorytmów planowania dostępu do dysku:

- (a) FSCS,
- (b) SSTF.

- (c) SCAN,
 (d) LOOK,
 (e) C-SCAN.
- 13.3** Z elementarnych praw fizyki wiadomo, że jeśli obiekt ma stałe przyspieszenie a , to zależność między odległością d i czasem t jest określona wzorem: $d = at^2/2$. Założmy, że podczas szukania dysk z ćw. 13.2 jednostajnie przyspiesza ruch swojego ramienia przez pierwszą połowę szukania, a następnie zwalnia w tym samym tempie przez drugą połowę szukania. Przyjmijmy, że dysk może wykonać przemieszczenie do sąsiedniego cylindra w 1 ms oraz że całe przejście wszystkich 5000 cylindrów trwa 18 ms.
- Odległość szukania określamy jako liczbę cylindrów mijanych przez głowicę. Wyjaśnij, dlaczego czas szukania jest proporcjonalny do pierwiastka kwadratowego z odległością szukania.
 - Napisz równanie czasu szukania w funkcji odległości szukania. Równanie to powinno mieć postać $t = x + y \sqrt{L}$, gdzie t jest czasem w milisekundach, a L jest odlegością szukania wyrażoną liczbą cylindrów.
 - Oblicz łączny czas szukania dla każdego z zaplanowań z ćw. 13.2. Określ, który z planów jest najszybszy (ma najmniejszy łączny czas szukania).
 - Procent polepszenia szybkości* (ang. *percentage speedup*) jest ilorazem czasu zaoszczędzonego i czasu pierwotnego. Ile wynosi procent polepszenia szybkości najszybszego zaplanowania w porównaniu z metodą FCFS?
- 13.4** Założmy, że dysk z ćw. 13.3 wiruje z prędkością 7200 obr/min.
- Ile wynosi średnie opóźnienie obrotowe tego napędu dyskowego?
 - Na jaką odległość można by wykonać szukanie w czasie obliczonym w części (a)?
- 13.5** Szukanie z przyspieszaniem, opisane w ćw. 13.3, jest typowe dla napędów dysków twardych. Z kolei dyski elastyczne (i wiele dysków twardych wyprodukowanych w pierwszej połowie lat osiemdziesiątych) wykonują zazwyczaj szukanie ze stałą prędkością. Przypuśćmy, że dysk w ćw. 13.3 działa ze stałą prędkością szukania zamiast ze stałym przyspieszeniem, tak więc czas szukania wyraża się wzorem: $t = x + vL$,

gdzie t jest czasem w milisekundach, a L – odległością szukania. Założymy, że czas odszukania sąsiedniego cylindra wynosi 1 ms, tak jak przedtem, a odnalezienie każdego następnego cylindra zajmuje 0,5 ms.

- Napisz równanie na czas szukania w zależności od odległości szukania w tych warunkach.
- Korzystając z funkcji czasu szukania z części (a), oblicz łączny czas szukania dla każdego z zaplanowań z ćw. 13.2. Czy Twoja odpowiedź jest taka sama jak w ćwiczeniu 13.3(c)?
- Ille w tym przypadku wynosi procent polepszenia szybkości najszybszego zaplanowania w stosunku do metody FCFS?

- Napisz program typu monitor (zob. rozdz. 6) służący do planowania dostępu do dysku z użyciem algorytmów SCAN i C-SCAN.
- Porównaj wydajność algorytmów planowania C-SCAN i SCAN przy założeniu równomiernego rozkładu zamówień. Rozważ średni czas odpowiedzi (czas między nadaniem zamówienia a zakończeniem jego obsługi), wariancję czasu odpowiedzi oraz wynikową przepustowość. Jaka jest zależność wydajności od względnych wielkości czasu szukania i opóźnienia obrotowego?
- Czy planowanie dostępu do dysku metodą inną niż FCFS jest przydatne w środowisku z jednym użytkownikiem? Odpowiedź uzasadnij.
- Wyjaśnij, dlaczego w planowaniu dostępu do dysku metodą SSTF środkowe cylindry są uprzywilejowane w stosunku do skrajnych cylindrów wewnętrznych i zewnętrznych?
- Zamówienia nie są na ogół rozmieszczone równomiernie. Na przykład cylinder zawierający tablicę FAT systemu plików albo i-węzły będzie z reguły częściej odwiedzany niż cylindry, które zawierają tylko pliki. Zakładamy, że wiemy, iż 50% zamówień dotyczy małej, ustalonej liczby cylindrów.
 - Czy któryś z algorytmów omówionych w tym rozdziale byłby szczególnie korzystny w tych warunkach? Odpowiedź uzasadnij.
 - Zaproponuj jeszcze wydajniejszy algorytm planowania dostępu do dysku, korzystając z wiedzy o tym „gorącym miejscu” na dysku.
 - Blok danych w systemach plików jest zazwyczaj znajdowany za pomocą pośredniej tablicy adresów, takiej jak tablica FAT w systemie MS-DOS lub i-węzły w systemie UNIX. Opisz jeden lub więcej

sposobów wykorzystania tego pośrednictwa do poprawienia wydajności dysku.

- 13.11** Dlaczego opóźnienie obrotowe nie jest zwyklebrane pod uwagę w planowaniu dostępu do dysku? W jaki sposób należałoby zmienić metody SSTF, SCAN i C-SCAN, aby objęły one optymalizacją również opóźnienie obrotowe?
- 13.12** Jaki wpływ miałoby zastosowanie RAM-dysku na Twój wybór algorytmu planowania dostępu do dysku? Jakie czynniki należałoby uwzględnić? Czy te same rozważania odnoszą się do planowania dostępu do dysku twardego przy założeniu, że system plików zapamiętuje ostatnio używane bloki w buforach podręcznych w pamięci głównej?
- 13.13** Dlaczego jest ważne, aby w systemie tworzącym środowisko wielozadaniowe równoważyć obciążenia dysków i sterowników operacjami wejścia-wyjścia pochodząymi z systemu plików?
- 13.14** Jakie są zalety i wady wielokrotnego czytania stron kodu z systemu plików w porównaniu z pamiętaniem ich w obszarze wymiany?
- 13.15** Czy istnieje jakiś sposób realizacji naprawdę trwałej pamięci? Odpowiedź uzasadnij.
- 13.16** Niezawodność napędu dysku twardego określa się zazwyczaj wielkością średniego czasu międzyawaryjnego (ang. *mean time between failure* – MTBF). Choć wielkość ta jest nazywana „czasem”, w istocie jest mierzoną liczbą godzin pracy dysku przypadającą na jedną awarię.
- (a) Jeśli w „stadzie” dysków znajduje się 1000 napędów, z których każdy ma średni czas międzyawaryjny wynoszący 750 000 godzin, to które z poniższych określeń najlepiej odzwierciedla częstość występowania awarii w tym stadzie: raz na tysiąc lat, raz na sto lat, raz na dziesięć lat, raz na rok, raz na miesiąc, raz na dzień, raz na godzinę, na minutę czy raz na sekundę?
 - (b) ze statystyk umieralności wynika, że mieszkaniec USA ma średnio 1 szansę na 1000, aby umrzeć między 20 a 21 rokiem życia. Określ, ile godzin wynosi „czas” MTBF dla amerykańskich dwudziestolatków. Zamień tę liczbę godzin na lata. Jakie informacje odnośnie do oczekiwanej czasu życia dwudziestolatków wynikają dla Ciebie z wartości parametru MTBF?
 - (c) Wytwórca oświadcza, że średni czas międzyawaryjny (MTBF) pewnego modelu napędu dysku wynosi milion godzin. Co możesz

powiedzieć o liczbie oczekiwanych lat niezakłóconej eksploatacji jednego z tych napędów?

13.17 Termin „szybka, szeroka szyna SCSI-II” (ang. *fast wide SCSI-II*) oznacza szynę SCSI przesyłającą pakiet między komputerem głównym a urządzeniem z szybkością 20 MB/s. Założmy, że dysk w napędzie podłączonym do szybkiej, szerokiej szyny SCSI-II obraca się z prędkością 7200 obr/min. jego sektory mają po 512 B, a na ścieżce mieści się 160 sektorów.

- (a) Oszacuj (w MB/s) szybkość przesyłania tego napędu na długim odcinku czasu.
- (b) Założmy, że napęd ten ma 7000 cylindrów, 20 ścieżek w cylindrze, czas przełączania jego głowicy (z jednej powierzchni na drugą) wynosi 0.5 ms, a czas przechodzenia głowicy na sąsiedni cylinder równa się 2 ms. Korzystając z tych dodatkowych informacji, określ dokładnie stałą szybkość przesyłania w przypadku wielkiej transmisji.
- (c) Przypuśćmy, że średni czas szukania na tym dysku równa się 8 ms. Oszacuj liczbę operacji wejścia-wyjścia w ciągu 1 s oraz efektywną szybkość przesyłania w przypadku obciążenia dostępem losowym, przy którym czyta się poszczególne sektory rozrzucone na dysku.
- (d) Określ liczbę losowych operacji wejścia-wyjścia przypadających na 1 s oraz szybkość przesyłania dla porcji długości: 4 KB, 8 KB i 64 KB.
- (e) Jeśli w kolejce znajduje się wiele zamówień, to algorytm planowania dostępu do dysku, taki jak SCAN, powinien zmniejszyć średnią odległość szukania. Założmy, że obciążenie dysku polega na losowym czytaniu stron o rozmiarze 8 KB, średnia długość kolejki wynosi 10 zamówień, a algorytm planowania zmniejsza średni czas szukania do 3 ms. Dla tych danych znajdź liczbę operacji wejścia-wyjścia przypadających na 1 s oraz efektywną szybkość przesyłania dysku.

13.18 Do szyny SCSI można podłączyć więcej niż jeden napęd dysków. W szczególności, do szybkiej, szerokiej szyny SCSI-II (zob. ćw. 13.17) można podłączyć do 15 napędów dysków. Przypomnijmy, że szyna ta ma przepustowość 20 MB/s. Między wewnętrzną pamięcią któregoś z dysków a komputerem głównym można w każdej chwili przesyłać tą szyną tylko 1 pakiet. Jednak dysk może przemieszczać swoje ramię, w czasie gdy inny dysk przesyła pakiet szyną. Podczas gdy jeden z dys-

ków przesyła pakiet za pomocą szyny, inny dysk może też przesyłać dane ze swoich powierzchni roboczych do wewnętrznej pamięci podręcznej. Biorąc pod uwagę szybkości przesyłania obliczone w cw. 13.17, określ liczbę dysków, które mogą być skutecznie używane za pośrednictwem jednej szybkiej i szerokiej szyny SCSI-II.

- 13.19** Fakt ponownego odwzorowania uszkodzonych bloków metodą korzystania z zapasu sektorów lub przeciągania sektorów może mieć wpływ na wydajność. Założmy, że napęd z cw. 13.17 ma ogółem 100 wadliwych sektorów rozmieszczonych losowo i każdy z uszkodzonych sektorów został odwzorowany za pomocą sektora rezerwowego, zlokalizowanego na innej ścieżce, lecz w tym samym cylindrze. Oszacuj liczbę operacji wejścia-wyjścia wykonywanych w ciągu 1 s oraz efektywną szybkość przesyłania dla losowego czytania obszarów o długości 8 KB, przy założeniu, że długość kolejki zamówień wynosi 1 (więc wybór dokonywany przez algorytm planowania dostępu do dysku nie ma znaczenia). Jaki wpływ na wydajność mają uszkodzone sektory?
- 13.20** Omów względne zalety metody sektorów zapasowych oraz metody przeciągania sektorów.

Uwagi bibliograficzne

Technologie dysków magnetycznych opisali Freedman [141] oraz Harker i in. [164]. Dyski elastyczne przedstawili Pechura i Schoeffler [328] oraz Sarisky [372]. Nadmiarowe tablice niezależnych dysków (RAID) zostały omówione przez Pattersona w materiałach konferencyjnych [325] oraz w szczegółowym przeglądzie dokonanym przez Chena i in. [69]. Architektury dysków dla wysokowydajnych obliczeń zaprezentowali Katz i in. w artykule [203].

Artykuł Teoreya i Pinkertona [421] zawiera wcześniejszą analizę porównawczą algorytmów planowania dostępu do dysków. Autorzy zastosowali symulację modelu dysku, w którym czas szukania zależy liniowo od liczby mijanych cylindrów. Dla takiego dysku i kolejki o długości mniejszej niż 140 zamówień dobrze jest wybrać algorytm LOOK, a w przypadku kolejki dłuższej niż 100 zamówień odpowiedni okazał się algorytm C-LOOK. Singhania i Tonge [394] przedstawili oczekiwany czas szukania jako funkcję rozmiaru kolejki w odniesieniu do różnych algorytmów planowania dostępu do dysku, zakładając liniową funkcję czasu szukania, i opisali sparametryzowany algorytm, który można łatwo dostosować do działania w trybie FCFS, SCAN i C-SCAN. Geist i Daniel w artykule [147] zaprezentowali algorytm z parametrami, który może płynnie przechodzić między metodami SSTF i SCAN.

Pośrednie wartości parametru sterującego wskazują na uzyskiwanie lepszej wydajności niż z osobna stosowane metody SSTF i SCAN. King w publikacji [214] omawia sposoby polepszenia czasu szukania za pomocą przemieszczania ramienia dysku w chwilach bezczynności dysku. W materiałach z konferencji [443] Worthington i in. zaprezentowali algorytmy planowania dostępu do dysku, w których – oprócz czasu szukania – uwzględniono opóźnienie obrotowe, oraz omówili wpływ postępowania z uszkodzeniami na wydajność. Znaczenie umiejscowienia intensywnie używanych danych na czas szukania było przedmiotem rozważań autorów takich, jak Wong [442], Ruemmler i Wilkes [359] oraz Akyurek i Salem [7].

Ruemmler i Wilkes w artykule [361] omawiają dokładny model działania nowoczesnego napędu dysku. Worthington i in. [444] wyjaśniają, w jaki sposób podejmować decyzje o niskopoziomowych właściwościach dysku, takich jak struktura stref.

Rozmiar wejścia-wyjścia oraz losowość obciążenia zamówieniami mają istotny wpływ na wydajność dysku. Ousterhout i in. [319] oraz Ruemmler i Wilkes [360] dają przegląd wielu interesujących charakterystyk obciążen, uwzględniając fakt, że pliki w większości są małe, nowo otwarte pliki najczęściej są szybko usuwane, większość plików jest otwieranych do czytania i czytanych sekwencyjnie od początku do końca, a cdległości szukania są na ogół krótkie. McKusick i in. w artykule [278] omawiają szybki system plików (FFS) z Berkeley, w którym zastosowano wiele przemyślnych metod w celu uzyskania dobrej wydajności dla szerokiego repertuaru obciążen. McVoy i Kleiman w artykule [280] przedstawili dalsze ulepszenia podstawowego systemu FFS.

Systemy plików o budowie rejestrowej (ang. *log-structured file systems* – LFS) omówili Finlayson i Cheriton [136], Dougis i Ousterhout [117] oraz Rosenblum i Ousterhout [357]. Systemy te zaprojektowano w celu poprawiania wydajności dyskowych operacji pisania przy założeniu, że optymalizowanie czytania za pomocą buforów podręcznych w pamięci głównej jest mniej ważne niż optymalizowanie w ten sposób operacji pisania. Dobrze zestrojona wersja systemu LFS w praktyce spisuje się dla różnych obciążzeń dysku magnetycznego niemal tak samo dobrze jak system FFS, przy czym podejście LFS może okazać się cenne w odniesieniu do urządzeń pamięci, które można zapisywać tylko jeden raz lub takich, na których jest dozwolone tylko dopisywanie informacji. Urządzeniami takimi są na przykład dyski optyczne.

Rozdział 14

STRUKTURA PAMIĘCI TRZECIORZĘDNEJ

W rozdziale 2 wprowadziliśmy pojęcia pamięci podstawowej, pomocniczej (czyli drugorzędnej) i trzeciorzędnej. W tym rozdziale omawiamy szczegółowo pamięć trzeciorzędową. Najpierw opisujemy rodzaje urządzeń stosowanych jako pamięci trzeciorzędne. Następnie omawiamy problemy, które powstają, gdy system operacyjny korzysta z pamięci trzeciorzędnej. Na koniec rozważamy aspekty wydajności systemów pamięci trzeciorzędnej.

14.1 ■ Urządzenia pamięci trzeciorzędnej

Czy kupilby ktoś magnetowid mający w środku tylko jedną taśmę, której nie wolno by było wyjąć ani zastąpić inną? Albo magnetofon kasetowy lub odtwarzacz płyt kompaktowych tylko z jedną kasetą lub płytą zamontowaną na stałe? Oczywiście – nie. W magnetowidzie lub odtwarzaczu płyt kompaktowych powinno być możliwe używanie wielu taniach kaset za pomocą jednego napędu, aby obniżyć ogólne koszty.

Niskie koszty to element definiujący pamięć trzeciorzędową. Tak więc w praktyce pamięć trzeciorzędna jest budowana z nośników wymiennych (ang. *removable media*). Najpopularniejszymi przykładami nośników wymiennych są dyskietki i płyty CD-ROM, choć istnieje też wiele innych rodzajów urządzeń pamięci trzeciorzędnej.

14.1.1 Dyski wymienne

Dyski wymienne są rodzajem pamięci trzeciorzędnej. Dyski elastyczne (dyskietki) są przykładem wymiennych dysków magnetycznych. Są one wykonane z cieniutkich tarcz pokrytych materiałem magnetycznym i ochronianych plastikowymi kopertami. Chociaż na typowych dyskach elastycznych można przechować około 1 MB informacji, podobną technologię stosuje się do wymiennych dysków magnetycznych mogących pomieścić więcej niż 1 GB. Wymienne dyski magnetyczne mogą być niemal tak samo szybkie jak dyski twarde, choć ich powierzchnie robocze są w większym stopniu narażone na uszkodzenia wskutek porysowania.

Inną odmianą dysku wymiennego jest *dysk magnetoptyczny*. Dane są zapamiętywane na twardej płycie powleczonej materiałem magnetycznym, lecz sposób ich zapisywania jest zupełnie inny niż na dysku magnetycznym. Głowica magnetoptyczna unosi się na znacznie większej odległości od powierzchni dysku niż głowica magnetyczna, a materiał magnetyczny jest pokryty grubą, ochronną warstwą plastiku lub szkła. Tak wykonany dysk jest znacznie odporniejszy na awarie głowicy.

Napęd dysku ma cewkę wytwarzającą pole magnetyczne – w temperaturze pokojowej pole to jest za rozległe i za słabe, aby namagnesować jeden bit na dysku. W celu zapisania bitu głowica dysku błyska promieniem laserowym nad jego powierzchnią roboczą. Laser jest wycelowany w malerki obszar, w którym ma być zapisany bit. Laser podgrzewa ten obszar, czyniąc go podatnym na pole magnetyczne. W ten sposób za pomocą rozległego i słabego pola magnetycznego można zapisać malerki bit.

Głowica magnetoptyczna jest za daleko od powierzchni dysku, aby odczytać dane przez wykrycie małych pól magnetycznych w sposób, w jaki to robi głowica dysku magnetycznego. Dane są odczytywane za pomocą światła laserowego z wykorzystaniem efektu *Kerra* (ang. *Kerr effect*). Gdy promień lasera odbija się od namagnesowanego miejsca, wówczas polaryzacja promienia laserowego ulega skręceniu zgodnie lub przeciwnie do ruchu wskazówek zegara – zależnie od kierunku pola magnetycznego. To skręcenie jest rozpoznawane przez głowicę jako przeczytany bit.

Jeszcze innym rodzajem dysków wymiennych są *dyski optyczne*. W tego rodzaju dyskach w ogóle nie korzysta się z magnetyzmu. Używa się w nich specjalnych materiałów, które można zmieniać za pomocą światła laserowego tak, aby nabierały plamek względem siebie jaśniejszych i ciemniejszych. Każda plamka reprezentuje bit. Przykładami dwu technologii dysków optycznych są dyski zmiennofazowe i dyski wykonane z barwionego polimeru.

Dysk zmiennofazowy (ang. *phase-change disk*) jest powlekany materiałem, który przy oziębianiu może krystalizować lub przechodzić w stan bezpo-

staciowy (zmieniać fazy). Te dwa stany powstają przy oświetleniu promieniem lasera o różnej mocy. Napęd dysku używa światła laserowego o różnej mocy, aby powodować topnienie i wtórne krzepnięcie plamek materiału na dysku, różnicując je pod względem uzyskiwanego przez nie stanu – krystalicznego lub bezpostaciowego.

W przypadku dysku z barwionego polimeru (ang. *dye-polymer disk*) korzysta się z innej właściwości fizycznej. Dane na dysku zapisuje się, wytwarzając na nim grudki. Dysk jest powleczony plastikiem zawierającym barwnik, który pochłania światło lasera. Promieniem lasera można podgrzać małutki obszar, wskutek czego ten obszar się „wybrusza” i tworzy grudkę. Laserem można też podgrzać grudkę, tak aby zmiękczała i skurczyła się do swego pierwotnego kształtu.

Technologie zmian fazy oraz barwionego polimeru nie są tak popularne jak metoda magnetoptyczna z powodu ich wysokiej ceny i małej wydajności. Ilustrują jednak poszukiwania nowych i lepszych metod zapisywania.

Dyski omówionych tutaj rodzajów mają tę cechę, że można ich używać wciąż na nowo. Nazywa się je dyskami zdatnymi do czytania i pisania. Inną klasę tworzą dyski do jednokrotnego zapisu i wielokrotnego czytania (ang. *write-once read-many-times* – WORM). Jeden ze sposobów wykonania dysku WORM polega na obłożeniu cienkiej folii aluminiowej dwiema szklanymi lub plastиковymi płytami. Do zapisania bitu używa się światła laserowego, za pomocą którego przepala się w folii aluminiowej maleńkie otwory. Ponieważ otworu nie można zlikwidować, każdy sektor na dysku może być zapisany tylko jeden raz. Chociaż informacje na dysku WORM można zniszczyć, przepalając otwory wszędzie, zmienianie danych jest na nim w istocie niemożliwe, ponieważ można tylko dodawać nowe otwory, a kod ECC skojarzony z każdym sektorem zapewne wykryliby takie dodatki. Dyski WORM uważa się za odporne na uszkodzenia i niezawodne, ponieważ warstwa metalu jest bezpiecznie obudowana dwiema ochronnymi płytami ze szkła lub plastiku, a pola magnetyczne nie mogą uszkodzić zapisu.

Dyski przeznaczone tylko do czytania (ang. *read-only disks*), takie jak CD-ROM i DVD, są zapisywane już w wytwórni. Technologia ich produkcji jest podobna do technologii dysków WORM, dzięki czemu są one również trwałe.

14.1.2 Taśmy

Taśma magnetyczna jest innego rodzaju wymiennym nośnikiem informacji. Ogólnie biorąc, możemy powiedzieć, że taśma jest tańsza niż dysk optyczny lub magnetyczny i mieści więcej danych. Szybkość przesyłania przewijaków taśm i napędów dysków są podobne. Jednakże dostęp losowy do taśmy jest o wiele wolniejszy niż szukanie na dysku, ponieważ wymaga operacji szyb-

kiego przewijania w przód lub wstecz, co zabiera dziesiątki sekund lub nawet minuty.

Chociaż typowy przewijak taśmy jest droższy niż typowy napęd dysku, koszt jednej kasety z taśmą jest znacznie mniejszy niż koszt dysku magnetycznego o równoważnej pojemności. Taśma jest więc ekonomicznym nośnikiem w zastosowaniach, które nie wymagają szybkiego dostępu swobodnego. Taśm używa się powszechnie do przechowywania zapasowych kopii danych dyskowych. Korzysta się z nich także w dużych centrach superkomputerowych do przechowywania olbrzymich ilości danych wykorzystywanych w badaniach naukowych oraz w wielkich przedsięwzięciach handlowych.

Niektoře taśmy mogą pomieścić znacznie więcej danych niż napęd dyskowy; powierzchnia robocza taśmy jest zazwyczaj znacznie większa niż powierzchnia robocza dysku. Pojemność pamięci taśmowych mogłaby się nawet jeszcze zwiększyć, ponieważ obecnie gęstość polowa (liczba bitów przypadających na cał kwadratowy) w technologiach taśmowych jest około 100 razy mniejsza niż dla dysków magnetycznych.

Wielkie instalacje taśmowe są zwykle wyposażone w automatyczne zmieniacze taśm, które przenoszą taśmy między przewijakami taśm a przegrodami na magazynowanie taśm w taśmotece. W ten sposób komputer ma zautomatyzowany dostęp do dużej liczby stosunkowo tanich szpul taśm. Bibliotekę, w której przechowuje się niewielką liczbę taśm, nazywa się czasem *stakerem* (ang. *stacker*)*, a na bibliotece zawierającą tysiące taśm mówi się niekiedy – *silos* (ang. *silo*) z taśmami.

Zautomatyzowana taśmoteka obniża ogólny koszt magazynowania danych. Pozostający na dysku plik, który przez pewien czas nie będzie potrzebny, można zarchiwizować na taśmie, której koszt za megabajt jest znacznie niższy. Jeśli plik stanie się potrzebny w przyszłości, to komputer może go przemieścić (ang. *stage*) z powrotem na dysk w celu czynnego użytkowania. Zautomatyzowaną taśmotekę nazywa się czasami biblioteką *prawie bezpośredni dostępną* (ang. *near-line library*), gdyż plasuje się ona między wysokowydajnymi, bezpośrednio dostępnymi (ang. *on-line*) dyskami magnetycznymi, a tanimi taśmami zaledającymi półki w magazynie taśm, dostępnymi pośrednio (ang. *off-line*).

14.1.3 Przyszłe technologie

W przyszłości mogą nabrać znaczenia inne technologie. Światło lasera może być na przykład użyte do zapisywania na specjalnych nośnikach fotografii holograficznych. O fotografiach czarno-białych można myśleć jak o dwuwy-

* Czyli stosem taśm. – Przyp. tłum.

miarowych tablicach pikseli. Kazdy piksel reprezentuje jeden bit: 0 dla czerni i 1 dla bieli. Fotografia o dużej rozdzielczości może zawierać miliony bitów danych, przy czym wszystkie piksele hologramu są przesyłane w jednym błysku światła laserowego. Szybkość przesyłania danych jest więc niezwykle duża. Zakres stosowania pamięci holograficznych jest dziś ograniczony, lecz postęp w ich rozwoju może doprowadzić do zastosowań na skalę hardlową.

Niezależnie od tego, czy nośnik pamięci jest dyskiem magnetooptycznym, płytą CD-ROM czy taśmą magnetyczną system operacyjny musi być wyposażony w środki umożliwiające korzystanie z nośników wymiennych do magazynowania danych. Środki te omawiamy w p. 14.2.

14.2 ■ Zadania systemu operacyjnego

Dwoma podstawowymi zadaniami systemu operacyjnego są: zarządzanie urządzeniami fizycznymi oraz tworzenie na użytku aplikacji abstrakcji maszyny wirtualnej. W rozdziale 13 przekonaliśmy się, że w przypadku dysków twardych system operacyjny tworzy dwie abstrakcje. Jedną jest urządzenie surowe, wyglądające po prostu jak tablica bloków danych. Drugą abstrakcję stanowi system plików. Dla systemu plików na dysku magnetycznym system operacyjny ustawia zamówienia w kolejki i planuje ich przeplatane wykonywanie z uwzględnieniem wielu aplikacji. Teraz przyjrzymy się, jak system operacyjny wywiązuje się ze swoich zadań wówczas, gdy nośniki pamięci są wymienne.

14.2.1 Interfejs aplikacji

Większość systemów operacyjnych obsługuje dyski wymienne niemal tak samo jak dyski stałe. Po włożeniu do napędu nowej kasety^{*} (po jej „zamontowaniu”) należy ją sformatować, w wyniku czego na dysku powstaje pusty system plików. Z systemu tego korzysta się tak samo jak z systemu plików na dysku twardym.

Z taśmami często postępuje się inaczej. System operacyjny zazwyczaj przedstawia taśmę jako surowy nośnik pamięci. Program użytkowy nie otwiera na taśmie pliku, natomiast otwiera cały przewijak taśmy jako urządzenie surowe^{**}. Na ogół przewijak taśmy zostaje zarezerwowany na wyłączny użytk danej aplikacji, do chwili aż zakończy ona działanie lub zamknie przewi-

^{*} Nie używamy tu terminu „dyskietka”, gdyż opis dotyczy także innych rozwiązań technicznych, np. dysków wymiennych w sztywnych obudowach metalowych. – Przyp. tłum.

^{**} Istnieją też systemy narzucające informacjom na taśmie pewną strukturę (podobną do plików). – Przyp. tłum.

jak odpowiednią operacją. Ta wyłączność jest uzasadniona, gdyż dostęp do losowo wybieranych miejsc na taśmie może trwać dziesiątki sekund lub nawet minuty, zatem przeplatanie takich dostępów odnoszących się do więcej niż jednej aplikacji powodowałoby prawdopodobnie szamotanie.

Skoro przewijak taśmy jest prezentowany jako urządzenie surowe, system operacyjny nie zapewnia na nim usług plikowych. Program użytkowy musi sam decydować, w jaki sposób korzystać z tablicy bloków. Na przykład program wykonujący na taśmie składowanie zawartości dysku twardego, może zapamiętać wykaz nazw i rozmiarów plików na początku taśmy, a potem kopiować dane z plików na taśmę z zachowaniem tego porządku.

Latwo zauważać trudności, które mogą wynikać z takiego sposobu użytkowania taśmy. Jeśli każda aplikacja stosuje własne reguły organizacji taśmy, to taśma zapelniona danymi może być wykorzystana tylko przez program, który ją utworzył. Na przykład, nawet jeśli wiadomo, że taśma składowania zawiera wykaz nazw plików i ich rozmiarów, po którym po kolei następują dane plików, wciąż możemy napotkać trudności z jej użyciem. Jak dokładnie wygląda pamiętanie nazw na taśmie? Czy rozmiary plików są podane binarnie, czy w kodzie ASCII? Czy pliki są zapamiętyane po jednym w bloku, czy też są połączone w jeden długий łańcuch bajtów? Nie znamy nawet rozmiaru bloku na taśmie, ponieważ z reguły można go wybierać z osobna przy zapisywaniu każdego bloku.

Podstawowymi operacjami napędu dysku są: czytaj, pisz i szukaj (ang. *read, write, seek*). Przewijaki taśm mają natomiast inny zbiór operacji. Zamiast operacji szukaj przewijak wykonuje operację znajdź (ang. *locate*). Taśmowa operacja odnajdywania jest dokładniejsza niż operacja szukania na dysku, ponieważ powoduje ona ustawienie taśmy w miejscu występowania określonego bloku logicznego, a nie odnalezienie całej ścieżki. Umiejscowienie taśmy w pozycji bloku 0 jest równoznaczne z jej zwinięciem.

Wiele rodzajów przewijaków taśmy umożliwia odnalezienie dowolnego bloku zapisanego na taśmie; nie jest jednak możliwe ustawienie taśmy w jakimś miejscu obszaru położonego za ostatnim zapisanym blokiem. Możliwość taka nie istnieje, ponieważ większość przewijaków taśmy zarządza swoją przestrzenią fizyczną inaczej niż pamięci dyskowe. W pamięci dyskowej sektory mają ustalony rozmiar, a zapisywanie danych musi być poprzedzone procesem formatowania, rozlokowującym docelowe, puste sektory. Większość przewijaków taśmy dopuszcza stosowanie bloków o różnych rozmiarach, które są ustalane tuż przed zapisaniem bloku na taśmie. Jeśli podczas pisania zostanie wykryty obszar wadliwy, to pomija się go i zapisuje blok ponownie. To działanie pozwala wyjaśniczyć, dlaczego nie jest możliwe odnalezienie miejsca w pustym obszarze poza zapisanymi danymi – nie są tam jeszcze określone położenia ani liczba bloków logicznych.

Większość przewijaków taśmy ma operację pozycja czytania (ang. *read position*) przekazującą numer bloku logicznego, od którego zaczyna się nagłówek taśmy. Wiele przewijaków taśmy ma także operację odstęp (ang. *space*)*, umożliwiającą zmianę położenia taśmy względem bieżącego miejsca. Tak więc na przykład operacja odstęp-2 spowodowałaby cofnięcie taśmy o dwa bloki logiczne.

W większości przewijaków taśmy zapisywanie bloku wywołuje efekt uboczny w postaci logicznego zatarcia wszystkiego, co występuje poza pozycją pisania. W praktyce ten efekt uboczny oznacza, że większość przewijaków taśmy to urządzenia umożliwiające tylko dopisywanie, ponieważ uaktualnienie bloku w środku taśmy powoduje również usunięcie wszystkiego poza tym blokiem. Przewijak taśmy implementuje to dopisywanie przez umieszczenie znacznika końca taśmy (ang. *end-of-tape* – EOT) po zapisaniu bloku. Przewijak odmawia przechodzenia poza znacznik EOT, umożliwia natomiast jego odnalezienie i rozpoczęcie po nim pisania. Ta czynność powoduje usunięcie starego znacznika końca taśmy i zapisanie nowego znacznika tuż za właśnie zapisanym blokiem.

W zasadzie byłoby możliwe zrealizowanie na taśmie systemu plików. Jednak wiele struktur danych i algorytmów systemu plików musiałoby różnić się od stosowanych na dyskach ze względu na to, że na taśmie informacje mogą być tylko dopisywane.

14.2.2 Nazywanie plików

Innym pytaniem, na które w systemie operacyjnym trzeba znaleźć odpowiedź, jest sposób nazywania plików na wymiennych nośnikach. Nazywanie na dysku stałym nie jest rzeczą trudną. W komputerze PC nazwa pliku składa się z litery określającej jednostkę pamięci dyskowej oraz z występującej dalej nazwy ścieżki. W systemie UNIX nazwa pliku nie zawiera litery określającej napęd dyskowy, za to tablica montowania umożliwia systemowi operacyjnemu zorientowanie się, która jednostka pamięci zawiera dany plik. Jeśli jednak dysk jest wymienny, to wiedza o tym, który napęd mieścił zawierający go zasobnik w przeszłości nie daje podstawy do tego, żeby wiedzieć, jak odnaleźć plik. Gdyby wszystkie wymienne kasety z nośnikami informacji na świecie miały niepowtarzalne numery seryjne, to nazwa pliku na wymiennym urządzeniu pamięci mogłaby być poprzedzana numerem seryjnym. Jednak zapewnienie, aby zadne dwa numery seryjne nigdy się nie powtórzyły, wymagałoby, żeby każdy z nich miał około 12 cyfr. Kto zdolałby zapamiętać nazwy plików, z których każda zaczynałaby się od 12-cyfrowego numeru seryjnego?

* Często nazywaną „przewijaniem”. Przyp. tłum.

Problem staje się jeszcze poważniejszy, jeżeli chcemy zapisywać dane na kasetowym nośniku informacji w jednym komputerze, po czym korzystać z tej kasety na innym komputerze. Jeśli obie maszyny są tego samego typu i rozporządzamy tym samym rodzajem wymiennego napędu, to jedyną trudnością jest znajomość zawartości i układu danych na takiej kasetce. Jeśli jednak maszyny lub jednostki pamięci są różne, to może powstać wiele dodatkowych trudności. Nawet w przypadku zgody jednostek pamięci, może się okazać, że różne komputery pamiętają bajty w różnym porządku i stosują różne sposoby kodowania liczb dwójkowych lub nawet znaków (np. kod ASCII na komputerach PC i kod EBCDIC na dużych maszynach stacjonarnych).

W odniesieniu do nośników wymiennych dzisiejsze systemy operacyjne pozostawiają na ogół problem przestrzeni nazewniczej nie rozwiązany. Spособ dostępu do danych na tych nośnikach oraz ich interpretacja należy do aplikacji i użytkowników. Na szczęście kilka rodzajów wymiennych nośników informacji jest tak dobrze ustandaryzowanych, że wszystkie komputery używają ich w ten sam sposób. Przykładem jest dysk kompaktowy (CD). Muzyczne płyty kompaktowe są stosowane jako uniwersalny format zrozumiały dla dowolnego napędu CD. Dyski kompaktowe z danymi mają tylko kilka różnych formatów, więc zazwyczaj w ich przypadku moduł obsługi napędu dysków kompaktowych w systemie operacyjnym jest zaprogramowany tak, że potrafi obsługiwać wszystkie popularne formaty.

14.2.3 Zarządzanie pamięcią hierarchiczną

Robot kasetowy (ang. *robotic jukebox*) umożliwia komputerowi zmianę wybranej kasety z taśmą lub dyskiem bez udziału człowieka. Dwa ważne zastosowania takiej techniki to: wykonywanie składowań i hierarchiczne systemy pamięci. Zastosowanie robota kasetowego do składowania jest proste – kiedy jedna kasa zostaje zapeliona, wtedy komputer poleca robotowi przełączyć się na następną kasę.

Hierarchiczny system pamięci jest rozwinięciem hierarchii pamięci poza pamięć podstawową i pomocniczą (tj. dysk magnetyczny), tak aby objąć zasięgiem pamięć trzeciorzędną. Pamięć trzeciorzędna jest zazwyczaj implementowana za pomocą robota kasetowego z taśmami lub dyskami wymiennymi. Ten poziom pamięci jest obszerniejszy i tańszy, lecz najczęściej także – powolniejszy.

Chociaż system pamięci wirtualnej można by rozszerzyć w bezpośredni sposób na pamięć trzeciorzędną, rozszerzenie takie rzadko kiedy wdraża się w praktyce. Przyczyna leży w tym, że odzyskiwanie informacji przechowywanych przez taśmowego robota kasetowego może trwać dziesiątki sekund lub nawet minuty. Tak długie opóźnienie jest nie do przyjęcia w stronicowaniu na żądanie ani w innych formach zastosowania pamięci wirtualnej.

Zazwyczaj pamięć trzeciorzędną wdraża się jako rozszerzenie systemu plików. Małe i często używane pliki pozostają na dysku magnetycznym, podczas gdy wielkie, stare pliki, które nie są aktualnie w użyciu, podlegają archiwizowaniu przez roboty kasetowe. W niektórych systemach archiwizacji plików pozostawia się wpis katalogowy pliku, lecz zawartość pliku przestaje zajmować miejsce w pamięci pomocniczej. Jeśli aplikacja usiłuje otworzyć plik, to systemowe wywołanie `open()` zostaje zawieszone do chwili, w której zawartość pliku zostanie sprowadzona na dysk z pamięci trzeciorzędej. Z chwilą gdy zawartość pliku staje się znów dostępna na dysku magnetycznym, operacja `open()` przekazuje sterowanie do aplikacji, która kontynuuje działanie, korzystając z dyskowej kopii danych. Hierarchiczne zarządzanie pamięcią (ang. *hierarchical storage management* – HSM) wdrożono w zwykłych systemach z podziałem czasu, takich jak TOPS-20, które działały na minikomputerach firmy Digital Equipment Corporation w latach siedemdziesiątych. W dzisiejszych czasach organizacja HSM pamięci jest zazwyczaj spotykana w centrach superkomputerowych i innych wielkich instalacjach, które rozporządzają niezwykle wielką liczbą tomów danych.

14.3 ■ Zagadnienia dotyczące wydajności

W tym punkcie omawiamy trzy aspekty wydajności pamięci trzeciorzędej: szybkość, niezawodność i koszt.

14.3.1 Szybkość

Istnieją dwa aspekty szybkości pamięci trzeciorzędej: szerokość pasma (czyli przepustowość) i opóźnienie. Szerokość pasma mierzy się w bajtach na sekundę. Przez *przepustowość stałą* (ang. *sustained bandwidth*) rozumie się średnią szybkość przesyłania danych obserwowaną w trakcie długich przesłań; oblicza się ją dzieląc liczbę przesłanych bajtów przez czas przesyłania. *Przepustowość efektywna* (ang. *effective bandwidth*) jest średnią z całego czasu trwania operacji wejścia-wyjścia, łącznie z czasem szukania lub odnajdywania danych oraz czasami zużywanymi na wszelkie przełączenia kaset przez roboty. Z natury rzeczy przepustowość stała określa szybkość danych podczas przepływu ich strumienia, a przepustowość efektywna jest ogólną szybkością przesyłania danych uzyskiwaną w danym urządzeniu. Mówiąc „przepustowość napędu” (ang. *bandwidth of a drive*), z zasady rozumie się przepustowość stałą.

Szerokości pasma dysków wymiennych wahają się w przedziale od 0,25 MB/s (najwolniejsze) do kilkunastu MB/s (najszybsze). Przewijaki taśm

charakteryzują się nawet większą szerokością pasma, od mniej niż 0.25 MB/s do ponad 30 MB/s. Najszybsze przewijaki taśmy mają znacznie większą przepustowość niż dyski wymienne lub stałe. Średnio możemy przyjmować, że przepustowość napędów taśmowych jest mniej więcej taka sama jak przepustowość napędów dyskowych.

Drugim aspektem mającym wpływ na szybkość jest *opóźnienie dostępu* (ang. *access latency*). Według tej miary wydajności dyski okazują się znacznie szybsze od taśm. Pamięć dyskowa ma w istocie dwa stopnie swobody – wszystkie bity są w zasięgu w chwili otwarcia. Podczas dostępu do dysku jego ramię po prostu przesuwa się nad wybrany cylinder i czeka przez czas opóźnienia obrotowego, co może zajmować mniej niż 35 ms. Natomiast pamięć taśmowa ma trzy stopnie swobody. W każdej chwili pod głowicą jest dostępna mała część taśmy, natomiast większość bitów kryją setki lub tysiące warstw taśmy nawiniętej na szpulę. Dostęp losowy do taśmy wymaga przewijania taśmy ze szpuli na szpulę, aż wybrany blok znajdzie się pod głowicą, co może zabierać dziesiątki lub setki sekund. Możemy więc powiedzieć, że na ogólny dostęp losowy do taśmy jest setki lub tysiące razy wolniejszy niż dostęp swobodny do dysku^{*}.

W przypadku robota kasetowego opóźnienie dostępu może znacznie wzrosnąć. Aby wymienić dysk, urządzenie napędowe musi się zatrzymać, automatyczny podajnik musi wybrać kasetę z dyskiem, po czym wybrany nośnik w nowej kasecie musi zostać wprowadzony w ruch wirowy. Operacja ta zajmuje kilkanaście sekund – około stu razy dłużej niż dostęp swobodny w ramach jednego dysku. Przelaczanie dysku w robocie powoduje więc względnie duży uszczerbek wydajności.

W przypadku taśm czas pracy robota jest mniej więcej taki sam jak dla dysków. Jednak przełączanie taśm wymaga z reguły zwinięcia starej taśmy, zanim będzie można ją wyjąć, a ta czynność może zajść do 4 minut. Poza tym, po złożeniu nowej taśmy przewijak może potrzebować wielu sekund na kalibrację i przygotowanie do wykonywania operacji wejścia-wyjścia. Choć wolno działający robot kasetowy przełącza taśmy w czasie wynoszącym od 1 do 2 minut, czas ten nie jest zwykle zbytnio większy od losowego dostępu w obrębie jednej taśmy.

Ogólnie biorąc, powiemy, że dostęp losowy w robotach obsługujących kasety z dyskami ma opóźnienie rzędu dziesiątek sekund, natomiast opóźnienie dostępu losowego w robotach obsługujących kasety z taśmami wynosi setki sekund. Przelaczanie dysków jest kosztowne, za to przełączanie taśm jest tanie. Nie należy przesadzać z uogólnianiem. Niektóre drogie „grające

* Dlatego nie chcemy nawet używać określenia „dostęp swobodny” w przypadku taśmy.
– Przyp. tłum.

szafy" z taśmami potrafią zwinąć, wyrzucić, załadować nową taśmę i przewinąć ją do dowolnie wybranego miejsca w czasie krótszym niż 30 s.

Jeśli skupiamy uwagę wyłącznie na wydajności napędów w robotach kasetowych, to szerokość pasma i opóźnienie wydają się możliwe do przyjęcia. Jeśli jednak przyjrzymy się samym kasetom, to okaże się, że kryją one potworne ograniczenie. Rozważmy najpierw szerokość pasma. W porównaniu z dyskiem stałym zautomatyzowana biblioteka ma znacznie gorszy stosunek szerokości pasma do pojemności pamięci. Przeczytanie wszystkich danych z wielkiej biblioteki mogłoby zająć lata. Niemal tak samo zła jest sytuacja w odniesieniu do opóźnienia dostępu. Aby to zilustrować, zauważmy, że jeśli 100 zamówień czeka w kolejce do napędu dysku, to średni czas czekania będzie wynosić około 1 s. Gdyby 100 zamówień czekało w kolejce do biblioteki taśm, to średni czas czekania mógłby wynieść ponad 1 godzinę. Niski koszt pamięci trzeciorzędnej wynika ze wspólnego korzystania z wielu tanich kaset na kilku drogich napędach. Jednak biblioteka wymiennych nośników informacji najlepiej nadaje się do pamiętania rzadko używanych danych, ponieważ jest ona w stanie realizować stosunkowo mało zamówień wejścia-wyjścia w ciągu godziny.

14.3.2 Niezawodność

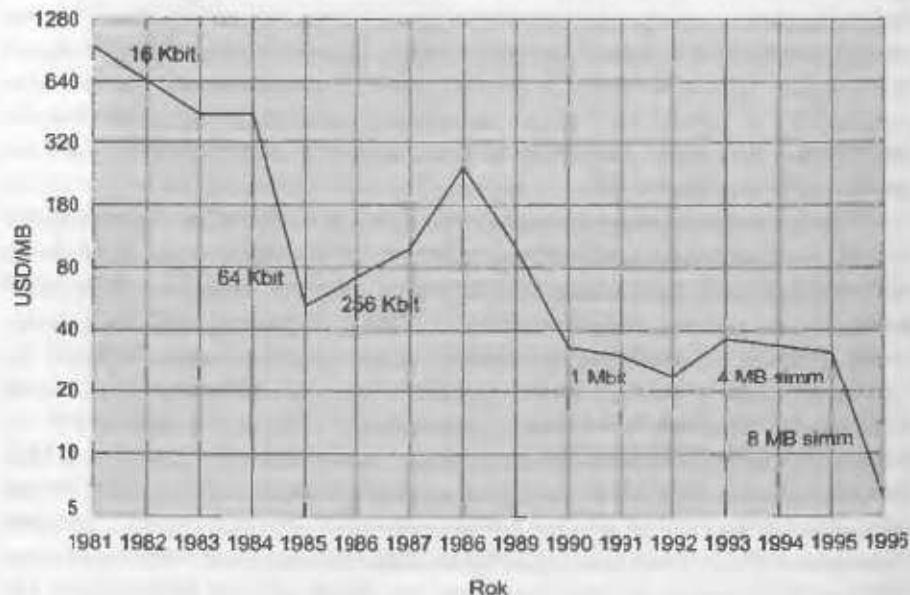
Choć zwykle uważamy, że „dobra wydajność” oznacza „dużą szybkość”, innym aspektem wydajności jest *niezawodność* (ang. *reliability*). Jeżeli chcemy przeczytać jakieś dane i nie możemy tego zrobić z powodu awarii napędu lub nośnika, to we wszystkich praktycznych zastosowaniach czas dostępu wydłuża się w nieskończoność, a szerokość pasa staje się nieskończoność mała. Jest zatem ważne, aby zdawać sobie sprawę z niezawodnością wymienionych nośników informacji.

Wymienne dyski magnetyczne są nieco mniej niezawodne niż stałe dyski twarde, ponieważ kasety są bardziej narażone na szkodliwe warunki środowiska, takie jak kurz, duże zmiany temperatury i wilgotności oraz oddziaływanie mechaniczne, jak na przykład wyginanie. Kasety z dyskami optycznymi uważa się za bardzo niezawodne, ponieważ warstwa przechowująca bity jest obłożona dwiema przezroczystymi, plastиковymi lub szklanymi warstwami ochronnymi. Niezawodność taśmy magnetycznej zmienia się znacznie w zależności od rodzaju napędu. Niektóre tanie napędy ścierają taśmę już po kilkudziesięciu przebiegach. Inne typy napędów obchodzą się z nośnikami tak delikatnie, że pozwalają używać ich miliony razy. W porównaniu z dyskiem magnetycznym głowica przewijaka taśmy magnetycznej jest czulym punktem. Głowica dysku unosi się nad nośnikiem, natomiast głowica taśmy jest w bliskim kontakcie z taśmą. Szorstka taśma może zetrzeć głowicę po kilku lub kilkudziesięciu tysiącach godzin.

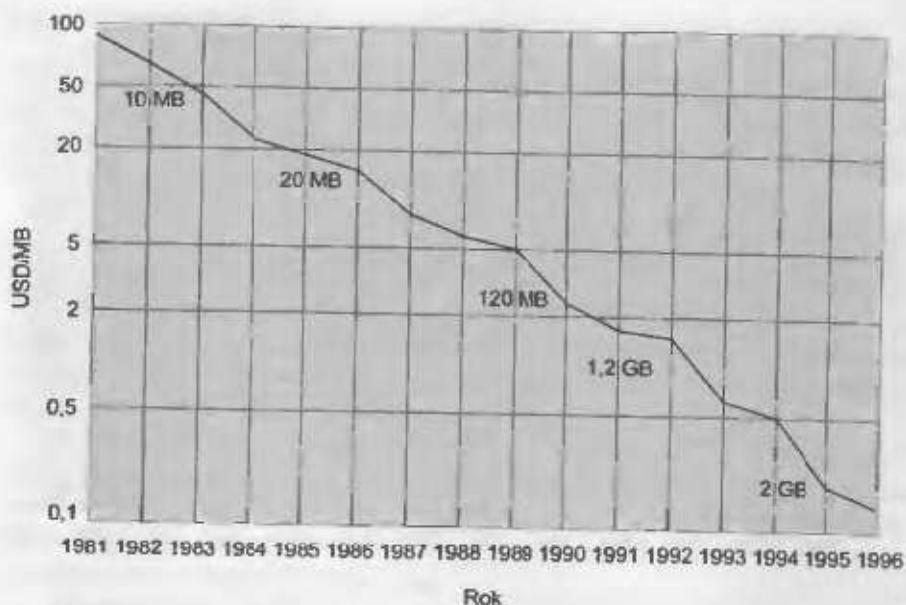
Podsumowując, możemy powiedzieć, że napęd dysku stałego działa zwykle bardziej niezawodnie niż napęd dysku wymiennego lub taśmy, a dysk optyczny jest bardziej niezawodny niż dysk magnetyczny lub taśma. Jednak stare dyski magnetyczne mają jedną słabość. Awaria głowicy dysku twardego niszczy dane zupełnie, podczas gdy po awarii napędu taśmy lub dysku optycznego dane często pozostają w stanie nienaruszonym.

14.3.3 Koszt

Innym, ważnym czynnikiem jest koszt pamięci. Podamy tu konkretny przykład obniżania przez nośniki wymienne kosztu całej pamięci. Przyjmijmy, że dysk twardy o pojemności X gigabajtów kosztuje 200 USD. W tej cenie 190 USD to koszt obudowy, silnika i sterownika, a cena płyt magnetycznych wynosi 10 USD. Koszt pamięci na takim dysku wynosi zatem $200 \text{ USD}/X$ za gigabajt. Założymy teraz, że potrafimy wytwarzać płyty w wymiennych kasetach. Za jeden napęd i 10 kaset cena łączna wyniesie $190 \text{ USD} + 100 \text{ USD}$, a uzyskana pojemność będzie równa $10X \text{ GB}$. Koszt pamięci wyniesie więc teraz $29 \text{ USD}/X$ za gigabajt. Nawet jeśli wyprodukowanie wymiennej kasety będzie nieco drozsze, koszt za gigabajt w przypadku pamięci na nośnikach wymiennych będzie zapewne niższy niż koszt za gigabajt na dysku twardym, ponieważ wysoka cena jednego napędu zostaje usredniona przez niską cenę wielu wymiennych kaset.



Rys. 14.1 Cena za kilobit (Kbit), kilobajt (KB) lub megabajt (MB) pamięci DRAM na przestrzeni lat 1981-1996 (na podstawie magazynu BY/E)

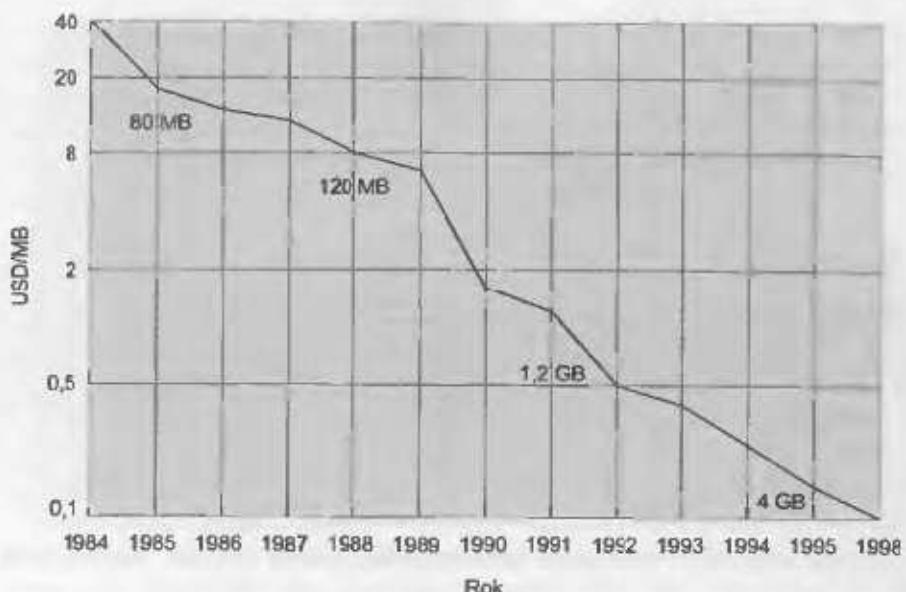


Rys. 14.2 Cena za megabajt (MB) lub gigabajt (GB) na magnetycznym dysku twardym na przestrzeni lat 1981-1996 (na podstawie magazynu *BYTE*)

Na rysunkach 14.1, 14.2 i 14.3 są pokazane tendencje zmian ceny 1 MB pamięci DRAM^{*}, twardych dysków magnetycznych oraz uzyskiwanej na przewijakach taśmy. Ceny na diagramie są cenami minimalnymi z ogłoszeń w magazynie *BYTE*, publikowanymi na końcu każdego roku. Zauważmy, że ceny te odzwierciedlają rynek małych komputerów czytelników magazynu *BYTE*, na którym to rynku ceny są niskie w porównaniu z rynkami dużych komputerów stacjonarnych i minikomputerów. W przypadku taśmy cena dotyczy przewijaka jednej taśmy. Ogólny koszt pamięci tasmowej znacznie maleje w miarę zakupu coraz to większej liczby taśm, gdyż cena taśmy zazwyczaj wynosi 1% ceny przewijaka.

Koszty pamięci DRAM wahają się znacznie. W okresie między 1981 r. a 1996 r. można zaobserwować trzykrotne załamanie cen (około 1981, 1989 i 1996), kiedy to nadmierna produkcja spowodowała nasycenie rynku. Widzimy również dwa okresy (około 1987 i 1993), w których braki na rynku spowodowały spory wzrost cen. W przypadku dysków twardych spadek cen był znacznie bardziej równomierny, choć wydaje się, że od 1992 r. nastąpiło jego przyspieszenie. Ceny przewijaków taśmy maleją również równomiernie, zwłaszcza

* Pamięci dynamicznej o dostępie bezpośrednim (ang. *dynamic random access memory*).
– Przyp. tłum.



Rys. 14.3 Cena za megabajt (MB) lub gigabajt (GB) na przewijaku taśmy na przestrzeni lat 1984-1996 (na podstawie magazynu *BYTE*)

od 1990 r. Zauważmy, że nie pokazano cen przewijaków sprzed 1984 r., gdyż magazyn *BYTE* jest ukierunkowany na rynek małych komputerów, a przewijaki taśmy nie były szeroko używane w małych komputerach do 1984 r.

Z przytoczonych wykresów można wyciągnąć kilka wniosków. Po pierwsze, pamięć operacyjna jest o wiele droższa niż pamięć dyskowa. Po drugie, koszt za megabajt pamięci dysków twardych zmalał gwałtownie i jest konkurencyjny wobec cen taśm magnetycznych, jeśli brać pod uwagę użytkowanie jednej taśmy na przewijaku. Po trzecie, najtańsze napędy taśmowe i najtańsze napędy dysków osiągnęły po latach niemal te same pojemności. Był może wynik ten nie jest zaskakujący, skoro podstawowe zastosowanie napędów taśmowych w małych systemach polega na wykonywaniu na taśmach kopii zapasowych zawartości dysków twardych.

Chociaż nie pokazano tego na rysunkach 14.1, 14.2 i 14.3, zapisywane dyski optyczne mogą być droższymi nośnikami pamięci niż dyski twarde. Pamięć trzeciorzędna umożliwia poczynienie oszczędności, jeśli liczba kaset jest istotnie większa od liczby napędów. W rzeczywistości dopóki nie nastąpi przełamanie rynku, na przykład przez pamięci holograficzne, dopóty ceny pamięci trzeciorzędnej będą równie wysokie jak ceny pamięci pomocniczej, z wyjątkiem zastosowań takich, jak składowania wykonywane na stosunkowo tanich taśmach lub wielkie biblioteki taśmowe, w których koszt sprzętu jest rozłożony na wielką liczbę taśn.

14.4 ■ Podsumowanie

W tym rozdziale dowiedzieliśmy się, że pamięć trzeciorzędną tworzą stacje dysków i taśm, w których używa się wymiennych nośników informacji. Nadaje się do tego wiele różnych rozwiązań technicznych, w tym taśmy magnetyczne, wymienne dyski magnetyczne i magnetoptyczne oraz dyski optyczne.

W odniesieniu do dysków wymiennych system operacyjny z reguły zapewnia pełny zakres usług dostępnych przez interfejs systemu plików, łącznie z zarządzaniem przestrzenią i planowaniem obsługi kolejki zamówień. W wielu systemach operacyjnych nazwa pliku w wymiennej kasetie jest kombinacją nazwy jednostki pamięci i nazwy pliku w obrębie tej jednostki. Ta konwencja jest prostsza, lecz potencjalnie łatwiej może powodowaćomyłki niż nazwy identyfikujące poszczególne kasety.

W przypadku taśm system operacyjny na ogół dostarcza surowego interfejsu. Wiele systemów operacyjnych nie ma też wbudowanej obsługi robotów kasetowych. Ich obsługą może się zajmować moduł sterujący lub uprzywilejowana aplikacja zaprojektowana w celu wykonywania składowań lub zarządzania pamięcią hierarchiczną.

Przepustowość, opóźnienie oraz niezawodność są trzema istotnymi aspektami wydajności. Dyski i taśmy cechują dużą różnorodność przepustowości (tzw. szerokości pasma), jednakże opóźnienie dostępu losowego do taśm jest na ogół znacznie większe niż do dysków. Zmiany kaset w urządzeniach automatycznych są także stosunkowo długotrwałe. Ponieważ w robotach kasetowych liczba napędów przypadająca na liczbę kaset jest mała, więc czytanie dużej ilości danych w tych urządzeniach może zajmować dużo czasu. Nośniki optyczne, w których czuła warstwa jest chroniona przez przezroczyste okładziny, są z reguły bardziej niezawodne niż nośniki magnetyczne, w których materiał magnetyczny jest wystawiony na większe niebezpieczeństwo uszkodzeń fizycznych.

■ Ćwiczenia

- 14.1** System operacyjny zazwyczaj traktuje dyski wymienne jak dzielone systemy plików, natomiast przewijak taśmy przydziela w danym czasie tylko jednej aplikacji. Podaj trzy przyczyny, które mogłyby uzasadnić tę różnicę w traktowaniu dysków i taśm. Omów dodatkowe cechy, które byłyby pożądane w systemie operacyjnym, aby mógł on umożliwić dzielony dostęp do systemu plików w tasmowym robocie kasetowym. Czy aplikacje korzystające wspólnie z taśmowego robota powinny mieć jakieś specjalne właściwości, czy też mogłyby używać plików tak jak wtedy, kiedy pozostają one na dysku? Uzasadnij swoją odpowiedź.

- 14.2 Jaki skutek wywołałoby otwarcie w robocic kasztowym większej liczby plików niż liczba zamontowanych w nim napędów dysków?
- 14.3 Rozważ hierarchiczne zarządzanie pamięcią, archiwizowanie danych oraz wykonywanie kopii zapasowych. Określ, w jakich przypadkach byłoby najlepiej zrealizować pamięć trzeciorzędną w formie funkcji systemu operacyjnego, a w jakich można by ją potraktować po prostu jako pozytyczną aplikację.
- 14.4 Jaki wpływ na koszt i wydajność miałoby uzyskanie przez pamięć taśmową takiej gęstości powierzchniowej, jaką charakteryzuje się dyski?
- 14.5 Gdyby cena 1 GB pamięci na dysku twardym była równa cenie 1 GB pamięci na taśmie, to czy taśmy wyszłby z użycia, czy nadal byłyby potrzebne? Odpowiedź uzasadnij.
- 14.6 Dokonajmy prostego oszacowania, aby porównać koszt i wydajność terabajtowego systemu pamięci utworzonego w całości z dysków oraz systemu, w którym w tym celu wdrożono pamięć trzeciorzędną. Przyjmijmy, że każdy dysk magnetyczny mieści 10 GB, kosztuje 1000 USD, przesyła 5 MB/s, a jego średnie opóźnienie dostępu wynosi 15 ms. Założmy, że koszt 1 GB w bibliotece taśmowej wynosi 10 USD, szybkość przesyłania wynosi w niej 10 MB/s, a średnie opóźnienie dostępu – 20 s. Oblicz ogólny koszt, maksymalną szybkość przesyłania danych oraz średni czas oczekiwania w czystym systemie dyskowym. (Czy należy poczynić jakieś założenia dotyczące obciążen? Jeśli tak, to jakie?) Przypuśćmy teraz, że 5% danych jest w częstym użyciu, więc powinny one przebywać na dysku, natomiast pozostałe 95% zarchiwizowano w bibliotece taśmowej. Przypuśćmy też, że 95% zamówień jest obsługiwanych przez system dyskowy, a pozostałe 5% przez bibliotekę. Ile wyniesie ogólny koszt, maksymalna szybkość przesyłania danych oraz średni czas oczekiwania w takim systemie pamięci hierarchicznej?
- 14.7 Powiada się niekiedy, że taśma jest nośnikiem o dostępie sekwencyjnym, natomiast dysk jest nośnikiem o dostępie swobodnym. W rzeczywistości o swobodnym dostępie do urządzenia pamięci decydują osiągane przez nie rozmiary transmisji. Termin *strumieniowa szybkość przesyłania* (ang. *streaming transfer rate*) określa szybkość przesyłania danych w trakcie tego procesu, tj. z pominięciem opóźnień związanych z dostępem. Z kolei *efektywna szybkość przesyłania* (ang. *effective transfer rate*) wyraża się całkowitą liczbą bajtów przesłanych w ciągu sekundy – łącznie z czasem opóźnień dostępu itp.

Załóżmy, że w pcwnym komputerze pamięć podręczna drugiego poziomu ma opóźnienie dostępu równe 8 ns, a jej strumieniowa szybkość przesyłania wynosi 800 MB/s, opóźnienie dostępu do pamięci głównej wynosi 60 ns, natomiast strumieniowa szybkość przesyłania tej pamięci równa się 80 MB/s, dysk magnetyczny ma opóźnienie dostępu wielkości 15 ms i strumieniową szybkość przesyłania wynoszącą 5 MB/s, a dla napędu taśmy parametry te wynoszą: opóźnienie dostępu – 60 s, strumieniowa szybkość przesyłania – 2 MB/s.

- (a) Swobodny dostęp powoduje zmniejszenie efektywnej szybkości przesyłania urządzenia, ponieważ podczas uzyskiwania dostępu nie są przesyłane żadne dane. Jaka będzie efektywna szybkość przesyłania w przypadku dysku o podanych parametrach, jeśli średnio po jednym dostępie następuje przesłanie strumienia: 512 B, 8 KB, 1 MB i 16 MB?
 - (b) Miarą wykorzystania urządzenia jest współczynnik określony stosunkiem efektywnej szybkości przesyłania i strumieniowej szybkości przesyłania. Oblicz współczynnik wykorzystania jednostki pamięci dyskowej przy dostępie swobodnym, w wyniku którego następuje przesyłanie porcji danych każdego z rozmiarów podanych w części (a).
 - (c) Załóżmy, że wykorzystanie urządzenia wynoszące 25% (lub większe) uznajemy za zadowalające. Na podstawie przedstawionych parametrów wydajności oblicz najmniejszy rozmiar przesyłanej porcji danych, przy którym stopień wykorzystania dysku mieści się w akceptowalnym przedziale.
 - (d) Uzupełnij następujące zdanie: dysk jest urządzeniem o dostępie swobodnym dla przesyłań większych niż _____ bajtów, natomiast dla przesyłań mniejszych jest urządzeniem o dostępie sekwencyjnym.
 - (e) Oblicz minimalne rozmiary przesyłania, przy których wykorzystanie pamięci podręcznej, pamięci głównej oraz taśmy jest jeszcze zadowalające.
 - (f) W jakich warunkach taśmę można uznać za urządzenie o dostępie swobodnym, a w jakich za urządzenie o dostępie sekwencyjnym?
- 14.8** Co by się stało, gdyby wynaleziono urządzenie do korzystania z pamięci holograficznej? Załóżmy, że cena takiego urządzenia wyniosłaby 10 000 USD, a jego średni czas dostępu – 40 ms. Załóżmy, że w urządzeniu tym strumieniowa szybkość przesyłania wynosi 100 MB/s, a opóźnienie dostępu do pamięci wynosi 10 ns. Oblicz minimalny rozmiar przesyłania, przy którym stopień wykorzystania tego nowego typu pamięci mieści się w akceptowalnym przedziale.

dzeniu pamięci holograficznej znalazły zastosowanie kaseta wielkości płyty CD, w cenie 100 USD. Kaseta taka mogłaby przechować 40 000 obrazów, z których każdy miałby postać kwadratowego, czarno-białego zdjęcia o rozdzielczości 6000×6000 pikseli (każdy piksel odpowiadałby 1 bitowi). Przypuśćmy, że napęd ów potrafiłby przeczytać lub zapisać 1 obraz w ciągu 1 ms. Rozwaź odpowiedzi na następujące pytania:

- Do czego można by z pożytkiem zastosować takie urządzenie?
- Jaki byłby wpływ tego urządzenia na wydajność wejścia-wyjścia w systemie komputerowym?
- Jakie inne rodzaje urządzeń pamięci – jeśli w ogóle jakiekolwiek – stałyby się zbędne w wyniku upowszechnienia tego urządzenia?

14.9 Założymy, że gęstość powierzchniowa kasety z jednostronnym, 5,25-calowym dyskiem optycznym wynosi 1 (Gbit) na cal kwadratowy. Przymijmy, że gęstość powierzchniowa taśmy magnetycznej wynosi 20 (Mbit) na cal kwadratowy oraz że taśma ta ma pół cala szerokości i 1800 stóp długości*. Oblicz przybliżoną pojemność pamięci obu tych kaset. Przypuśćmy, że istnieje taśma optyczna o takich samych rozmiarach fizycznych jak opisana tu taśma magnetyczna, lecz o gęstości powierzchniowej dysku optycznego. Ile danych mogłaby ona pomieścić? Jaką uczciwą cenę można by było ustalić dla taśmy optycznej, jeśli taśma magnetyczna kosztuje 25 USD?

14.10 Założymy, że przez 1 kilobajt (KB) rozumiemy 1024 B, 1 megabajt (MB) to 1024^2 B, a 1 gigabajt (GB) równa się 1024^3 B. Ten szereg można kontynuować, ujmując w nim terabajty, petabajty i eksabajty (1024^6). Naukowcy oceniają, że w kilku wyjątkowo rozległych, geologicznych i astronomicznych projektach badawczych potrzeba będzie zapisać i przechować kilka eksabajtów danych w ciągu następnego dziesięciolecia. Udzielenie odpowiedzi na poniższe pytania, będzie wymagać przyjęcia kilku rozsądnych założeń. Przedstaw poczynione przez Ciebie założenia.

- Ile jednostek pamięci dyskowej należałoby przygotować do zapamiętania 4 eksabajtów danych?
- Ile taśm magnetycznych potrzeba do zapamiętania 4 eksabajtów danych?

* 1 cal = 2,540 cm, 1 stopa = 12 cali = 30,48 cm. – Przyp. tłum.

- (c) Na ilu dyskach optycznych można by pomieścić 4 eksabajty danych (ćw. 14.9)?
- (d) Ile kaset pamięci holograficznej należałoby przygotować, aby pomieścić 4 eksabajty danych (ćw. 14.8)?
- (e) Jak duża przestrzeń (w metrach sześciennych) byłaby potrzebna w celu zgromadzenia ilości pamięci określonej w każdym z wymienionych przypadków?

14.11 Omów, w jaki sposób system operacyjny mógłby utrzymywać wykaz wolnych obszarów w taśmowym systemie plików. Załóż, że technika taśmowa umożliwia tylko dopisywanie oraz że można w niej korzystać ze znacznika końca tasmy (EOT) i operacji znajdź, odstęp oraz pozycja czytania, opisanych w p. 14.2.1.

Uwagi bibliograficzne

Dyski optyczne omówili Kenville [206], Fujitani [143], O'Leary i Kitts [313], Gait [144] oraz Olsen i Kenley [316]. Ammon i in. [9] opisali szybki system dysków optycznych obsługiwanych przez robota.

Cannon [61] przedstawił technikę taśm magnetycznych. Freese [142] podał przejrzysty opis działania dysku magnetoptycznego. Quinlan [341] omówił sposób implementowania systemu plików w pamięci WORM z podręczną pamięcią dyskową. Richards [351] odnosi do pamięci trzeciorzędnej koncepcję systemu plików. Maher i in. w referacie [269] dokonali przeglądu integrowania rozproszonych systemów plików i pamięci trzeciorzędnej.

Pojęcie hierarchii pamięci przeanalizowano przed ponad czwierćwieczem. Na przykład w 1970 r. Mattson i in. omówili w artykule [273] matematyczną metodę określania wydajności hierarchii pamięci. Alt [8] przedstawił zastosowanie pamięci wymiennej w handlowym systemie operacyjnym, a Miller i Katz [294] podali charakterystykę dostępu do pamięci trzeciorzędnej w środowisku superkomputera. Benjamin w artykule [30] zaważył przegląd wymagań dotyczących pamięci masowej w projekcie EOSDIS opracowywanym w NASA.

Technologia pamięci holograficznej jest przedmiotem artykułu Psaltisa i Moka [336]. Zbiór artykułów dotyczących pamięci holograficznej, datujących się od 1963 r., zebrali Sincerbox i Thompson [392]. Asthana i Finkelstein w artykule [17] omówili kilka pojawiających się technologii pamięci, w tym takich, jak pamięć holograficzna, taśma optyczna i metoda wychwytywania elektronów.



Część 5

SYSTEMY ROZPROSZONE

System rozproszony jest zbiorem procesorów, które nie dzielą pamięci ani zegara. Każdy procesor ma własną, lokalną pamięć, a komunikacja między procesorami odbywa się za pomocą rozmaitych linii komunikacyjnych. Procesory w systemie rozproszonym różnią się rozmiarami i funkcjami. Mogą znajdować się wśród nich małe mikroprocesory, stacje robocze, minikomputery i wielkie, uniwersalne systemy komputerowe.

System rozproszony umożliwia użytkownikowi dostęp do różnych zasobów, nad którymi sprawuje opiekę. Dostęp do wspólnego zasobu pozwala na przyspieszanie obliczeń oraz zwiększa dostępność i niezawodność danych.

Rozproszony system plików jest systemem usług plikowych, którego użytkownicy, serwery i urządzenia magazynowania informacji znajdują się w różnych instalacjach systemu rozprozonego. Wskutek tego usługi muszą być wykonywane za pośrednictwem sieci; zamiast jednego, skoncentrowanego magazynu danych istnieje wiele niezależnych urządzeń pamięci.

System rozproszony musi zawierać różnorodne mechanizmy synchronizacji procesów i komunikacji, aby radzić sobie z problemem zakleszczeń oraz ze skutkami rozmaitych awarii, których nie spotyka się w systemie skoncentrowanym.

Rozdział 15

STRUKTURY SIECI

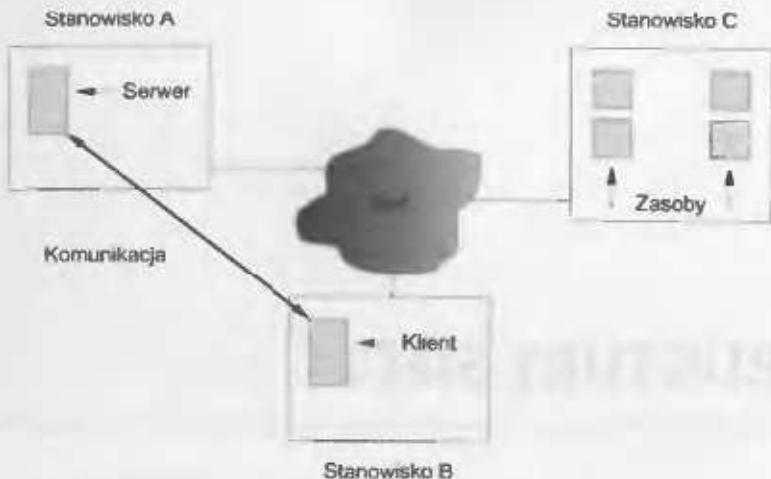
W systemach komputerowych widać od niedawna tendencję do rozdzielania obliczeń między wiele fizycznych procesorów. Istnieją dwa podstawowe schematy budowy takich systemów. W *systemie wieloprocesorowym* (ang. *multiprocessor system*), czyli ścisłe powiązanym, procesory dzielą pamięć i zegar, toteż komunikacja odbywa się w nim zazwyczaj poprzez pamięć dzieloną. W *systemie rozproszonym* (ang. *distributed system*), inaczej – luźno powiązanym, procesory nie dzielą pamięci ani zegara. Zamiast tego każdy procesor ma własną pamięć lokalną. Procesory komunikują się ze sobą za pomocą różnych sieci komunikacyjnych, takich jak szyny szybkiego przesyłania danych lub linie telefoniczne. W tym rozdziale omawiamy ogólną strukturę systemów rozproszonych oraz łączących je sieci. Szczegółowe omówienie jest zawarte w rozdz. 16–18.

15.1 ■ Podstawy

System rozproszony jest zbiorem luźno powiązanych ze sobą procesorów połączonych siecią komunikacyjną. Dla konkretnego procesora systemu rozproszonego pozostałe procesory i ich zasoby są *zdalne** (ang. *remote*), podczas gdy jego własne zasoby są *lokalne* (ang. *local*).

Procesory w systemie rozproszonym mogą różnić się mocą obliczeniową i funkcjami. Mogą znajdować się wśród nich małe mikroprocesory, stacje robocze, minikomputery i wielkie systemy komputerowe ogólnego przeznaczenia.

* Inaczej: *odległe*. – Przyp. tłum.



Rys. 15.1 System rozproszony

czenia. Procesory te określa się za pomocą kilku różnych nazw, takich jak *stanowiska* (ang. *sites*), *węzły* (ang. *nodes*), *komputery*, *maszyny*, *komputery sieciowe* lub *macierzyste* (ang. *hosts*) itp. – zależnie od kontekstu, w którym występują. Kiedy będziemy chcieli zwrócić uwagę na położenie maszyn, wtedy będziemy najczęściej używać terminu *stanowisko*, gdy zaś będziemy odnosić się do konkretnego systemu w danym miejscu, użyjemy (po prostu) nazwy *komputer* (sieciowy). Ogólnie biorąc, jest tak, że pewien proces na jakimś stanowisku – nazywany *serwerem* (ang. *server*) – dysponuje zasobem, którego potrzebuje inny proces na innym stanowisku – *klient* (ang. *client*). Zadaniem systemu rozproszonego jest tworzenie wydajnego i wygodnego środowiska umożliwiającego ten sposób dzielenia zasobów. System rozproszony jest pokazany na rys. 15.1.

Rozproszony system operacyjny umożliwia użytkownikom dostęp do różnorodnych zasobów, nad którymi sprawuje nadzór. Mianem *zasób* (ang. *resource*) określamy zarówno urządzenia sprzętowe, na przykład drukarki i przewijaki taśmy, jak i oprogramowanie – na przykład pliki i programy. Dostęp do tych zasobów jest nadzorowany przez system operacyjny. Istnieją dwa zasadnicze, uzupełniające się schematy dostarczania takich usług:

- **Sieciowe systemy operacyjne:** Użytkownicy są świadomi wielości maszyn i w celu dostępu do zasobów muszą rejestrować się na zdalnych maszynach lub przesyłać dane z odległych maszyn do swoich.
- **Rozproszone systemy operacyjne:** Użytkownicy nie muszą być świadomi wielości maszyn. Dostęp do zasobów zdalnych uzyskują oni tak samo jak do zasobów lokalnych.

Zanim omówimy te dwa rodzaje systemów operacyjnych, zwrócićmy uwagę na ich użyteczność oraz na strukturę sieci komputerowej tworzącej ich podstawy. W rozdziale 16 przedstawimy szczegółowe omówienie budowy tych dwóch rodzajów systemów operacyjnych.

15.2 ■ Motwy

Cztery główne powody uzasadniają budowę systemów rozproszonych: dzielenie zasobów, przyspieszanie obliczeń, niezawodność i komunikacja. W tym punkcie omówimy pokrótko każdy z nich.

15.2.1 Dzielenie zasobów

Jeśli pewna liczba różnych stanowisk (z różnymi możliwościami) jest ze sobą połączona, to użytkownik jednego stanowiska może korzystać z zasobów dostępnych na innym stanowisku. Użytkownik stanowiska A może na przykład korzystać z drukarki laserowej dostępnej tylko na stanowisku B. W tym samym czasie użytkownik stanowiska B może sięgać po plik rezydujący w A. Ogólnie można powiedzieć, że *dzielenie zasobów* (ang. *resource sharing*) w systemie rozproszonym jest mechanizmem umożliwiającym wspólne korzystanie z plików na zdalnych stanowiskach, przetwarzanie informacji w rozproszonych bazach danych, drukowanie plików na zdalnych stanowiskach, używanie zdalnych wyspecjalizowanych urządzeń (takich jak szybki procesor macierzowy) oraz wykonywanie innych operacji.

15.2.2 Przyspieszanie obliczeń

Jeśli konkretne obliczenie dałoby się podzielić na pewną liczbę obliczeń częściowych, które mogłyby być wykonywane współbieżnie, to system rozproszony może umożliwić rozdzielenie obliczeń między różne stanowiska w celu ich współbieżnego wykonywania. Na dodatek, jeśli jakieś stanowisko jest w danej chwili przeładowane zadaniami, to niektóre z tych zadań można przesunąć do innych, mniej obciążonych stanowisk. To przemieszczanie zadań nosi nazwę *dzielenia obciążenia* (ang. *load sharing*). Automatyczne dzielenie obciążenia, w którym rozproszony system operacyjny sam przemieszcza zadania, wciąż jest rzadkością w systemach dostępnych w handlu. Pozostaje on jednak przedmiotem intensywnych badań.

15.2.3 Niezawodność

Jeśli jedno stanowisko w systemie rozproszonym ulega awarii, to pozostałe stanowiska mają nadal szansę kontynuować działanie. Gdy system składa się z pewnej liczby dużych, autonomicznych instalacji (tzn. komputerów ogólnego przeznaczenia), wówczas awaria jednego z nich nie ma wpływu na resztę. Jeśli natomiast system składa się z pewnej liczby małych maszyn, z których każda odpowiada za jakąś ważną funkcję systemu (np. za operacje znakowego wejścia-wyjścia lub za system plików), to pojedyncze uszkodzenie może spowodować zatrzymanie całego systemu. Ujmując rzecz ogólnie, możemy powiedzieć, że jeśli w systemie istnieje wystarczająca nadmiarowość (zarówno w odniesieniu do sprzętu, jak i do danych), to system może wykonywać swoje zadania pomimo uszkodzenia jednego lub kilku z jego stanowisk.

Awaria stanowiska powinna być wykrywana przez system, przy czym może się okazać konieczne podjęcie odpowiednich działań zmierzających do usunięcia jej skutków. System powinien wstrzymać korzystanie z usług uszkodzonego stanowiska. Ponadto, jeżeli zadanie uszkodzonego stanowiska może przejąć inne stanowisko, to system powinien zapewnić, że będzie to wykonane poprawnie. Kiedy wreszcie uszkodzone stanowisko zostanie przywrócone do działania, wtedy będzie potrzebny mechanizm gładkiego włączenia go z powrotem do systemu. Jak się przekonamy w następnych rozdziałach, działania te stanowią trudne zagadnienia, mające wiele możliwych rozwiązań.

15.2.4 Komunikacja

Kiedy wiele stanowisk jest połączonych ze sobą za pomocą sieci komunikacyjnej, wtedy użytkownicy różnych stanowisk mają możliwość wymieniania informacji. Na niskim poziomie systemy przekazują między sobą *komunikaty* (ang. *messages*), co przypomina przekazywanie komunikatów w jednym komputerze, omówione w p. 4.6. Dzięki możliwości przekazywania komunikatów można rozszerzyć na system rozproszony wszystkie wyżej zorganizowane działania systemu autonomicznego. Do działań tych zaliczymy przekazywanie plików, rozpoczęcie sesji, obsługę poczty i wywoływanie procedur zdalnych (RPC).

Zaletą systemu rozprozonego jest to, że działania mogą być wykonywane na wielkie odległości. Nad jednym projektem mogą pracować dwie osoby na geograficznie oddalonych stanowiskach. Użytkownicy takiego systemu mogą zmniejszać ograniczenia wynikające z dużej odległości dzięki przesyłaniu plików projektu, rejestrowaniu się w zdalnych systemach w celu wykonywania programów i wymianie poczty umożliwiającej koordynowanie pracy. Niestety książka została w istocie napisana w ten sposób.

Zalety systemów rozproszonych, razem wzięte, spowodowały na skalę przemysłową dążenie do zastępowania dużych komputerów przez rozproszone złożone systemy sieciowe (ang. *downsizing*). Wiele firm zastępuje swoje wielkie centralne instalacje komputerowe sieciami stacji roboczych lub komputerów osobistych. Korzyści przy tym płynące, to przede wszystkim uzyskiwanie większej funkcjonalności w stosunku do poniesionych nakładów, elastyczność w rozmieszczaniu zasobów i rozszerzanie możliwości, lepsze interfejsy użytkowników i łatwiejsze pielęgnowanie systemu.

Jest oczywiste, że system operacyjny, w którym przewidziano zbiór procesów porozumiewających za pomocą systemu komunikatów (jak np. system Accent), daje się znacznie łatwiej rozszerzyć do systemu rozproszonego niż system nie obsługujący przekazywanego komunikatów. Na przykład system MS-DOS nie jest łatwy do zintegrowania w sieć, ponieważ jego jądro działa na zasadzie przerwań i nie ma w nim żadnych możliwości przesyłania komunikatów.

15.3 ■ Topologia

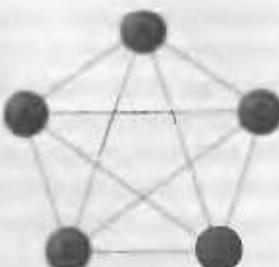
Stanowiska w systemie mogą być fizycznie połączone na różne sposoby. Każda konfiguracja ma zalety i wady. Omówimy pokrótko najczęściej realizowane konfiguracje i porównamy je według następujących kryteriów:

- **Koszt podstawowy:** Jak drogie jest połączenie różnych stanowisk w jeden system?
- **Koszt komunikacji:** Ile trwa przesłanie komunikatu ze stanowiska A do stanowiska B?
- **Niezawodność:** Jeśli łączy lub stanowisko w systemie ulegnie awarii, to czy pozostałe stanowiska pozostaną nadal ze sobą w kontakcie?

Różnorodne topologie są zilustrowane w postaci grafów, których węzły odpowiadają stanowiskom. Krawędź łącząca węzły A i B odpowiada bezpośredniemu połączeniu między tymi dwoma stanowiskami.

15.3.1 Sieci całkowicie połączone

W sieci *całkowicie połączonej* (ang. *fully connected network*) każde stanowisko jest bezpośrednio połączone ze wszystkimi pozostałymi stanowiskami w systemie (rys. 15.2). Koszt podstawowy takiej konfiguracji jest wysoki, ponieważ między każdymi dwoma stanowiskami musi istnieć bezpośrednie połączenie. Koszt podstawowy rośnie wprost proporcjonalnie do kwadratu

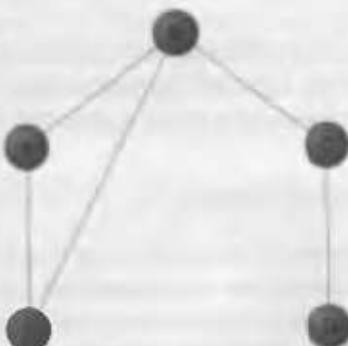


Rys. 15.2 Sieć w pełni połączona

liczby stanowisk. Jednak w tym środowisku komunikaty między stanowiskami mogą być przesyłane szybko; komunikat potrzebuje tylko jednego łącza, aby przebyć drogę między dwoma stanowiskami. Ponadto systemy takie są niezawodne, gdyż wiele połączeń musiałyby ulec awarii, zanim doszłoby do podziału systemu. System jest podzielony (ang. *partitioned*), jeśli został rozbity na dwa (lub więcej) podsystemów, między którymi brakuje połączeń.

15.3.2 Sieci częściowo połączone

W sieci częściowo połączonej (ang. *partially connected network*) bezpośrednie łącza istnieją między niektórymi, lecz nie wszystkimi parami stanowisk (rys. 15.3). Koszt podstawowy takiej konfiguracji jest więc niższy od kosztu sieci całkowicie połączonej. Jednak wysłanie komunikatu z jednego stanowiska do drugiego może wymagać przesyłania go przez kilka pośrednich stanowisk, co opóźnia komunikację. Na przykład w systemie naszkicowanym na rys. 15.3 komunikat ze stanowiska A do stanowiska D musi być przesyłany przez stanowiska B i C.



Rys. 15.3 Sieć częściowo połączona

Ponadto system częściowo połączony nie jest tak niezawodny jak sieć całkowicie połączona. Awaria jednego łącza może podzielić sieć. Jeśli na przykład awarii ulegnie przedstawione na rys. 15.3 połączenie między stanowiskami B i C, to sieć zostanie podzielona na dwa podsystemy. Jeden podsystem będzie zawierał stanowiska A, B i E, a drugi podsystem będzie się składał ze stanowisk C i D. Stanowiska w jednej części sieci nie będą mogły komunikować się ze stanowiskami w drugiej. Aby zmniejszyć możliwość wystąpienia takiej sytuacji, zazwyczaj stanowisko jest połączone przynajmniej z dwoma innymi stanowiskami. Jeśli na przykład dodalibyśmy połączenie między węzłami A i D, to awaria pojedynczego łącza nie spowodowałaby podziału sieci.

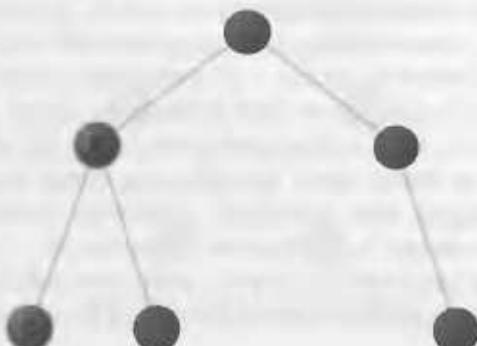
15.3.3 Sieci hierarchiczne

W sieci hierarchicznej (ang. *hierarchical network*) stanowiska tworzą strukturę drzewistą (rys. 15.4). Jest to organizacja powszechnie stosowana w sieciach obejmujących instytucje. Indywidualne biura są połączone z lokalnym biurem głównym. Biura główne są połączone z biurami regionalnymi. Biura regionalne są połączone z głównymi centrami korporacji.

Każde stanowisko (z wyjątkiem korzenia) ma jednoznacznie określone stanowisko zwierzchnie^{*} i pewną liczbę (być może zero) stanowisk podległych. Koszt podstawowy tej konfiguracji jest na ogół mniejszy od kosztu wynikającego ze schematu połączeń częściowych. W tym środowisku stanowisko zwierzchnie i stanowiska podległe komunikują się bezpośrednio. Stanowiska równorzędne mogą komunikować się ze sobą tylko za pomocą ich wspólnego zwierzchnika. Komunikat między dwoma stanowiskami równorzędnymi musi być wysyłany w górę do ich zwierzchnika, a następnie w dół do drugiego stanowiska równorzędowego. Podobnie, głębiej umiejscowione stanowiska podległe mogą się komunikować ze sobą tylko za pośrednictwem kilku szczebli tej hierarchii. Taka konfiguracja dobrze odpowiada ogólnemu spostrzeżeniu, że systemy położone blisko siebie komunikują się częściej niż systemy odległe. Na przykład szansa na to, że systemy będą sobie przekazywać dane jest większa, jeśli znajdują się one w jednym budynku niż wówczas, gdy są w oddzielnnych instalacjach.

Jeśli stanowisko zwierzchnie ulegnie awarii, to podległe mu stanowiska nie będą w stanie komunikować się ze sobą ani z innymi procesorami. Awaria dowolnego węzła (z wyjątkiem liścia) powoduje na ogół podział sieci na kilka rozłącznych poddrzew.

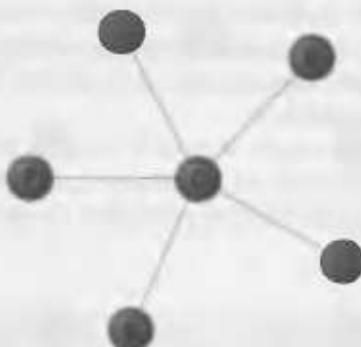
^{*} W oryginale użyto w tym akapicie terminologii „genealogicznej” (rodzic, dziecko, rođenstwo, kuzyni, dziadek). Mówi się też w podobnych sytuacjach o „przodkach” i „potomkach”. – Przyp. tłum.



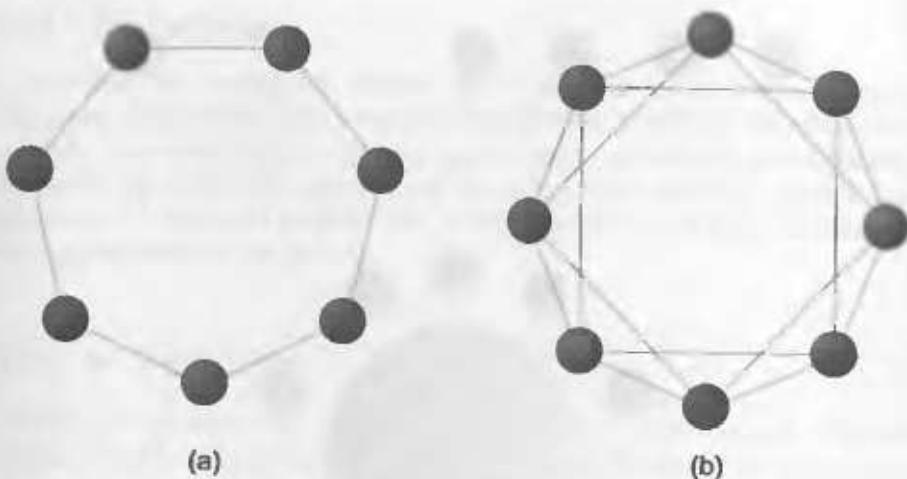
Rys. 15.4 Sieć o strukturze drzewiastej

15.3.4 Sieci gwiaździste

W sieci w kształcie gwiazdy (ang. *star network*) jedno ze stanowisk systemu jest połączone ze wszystkimi innymi stanowiskami (rys. 15.5). Żadne z pozostałych stanowisk nie ma połączenia z żadnym innym. Koszt podstawowy tego systemu jest funkcją liniową liczby stanowisk. Koszt komunikacji również jest niski, ponieważ wysłanie komunikatu od procesu A do procesu B oznacza wykonanie co najwyżej dwu przesyłań (od stanowiska A do stanowiska centralnego, a następnie od stanowiska centralnego do stanowiska B). W tym prostym schemacie stanowisko centralne może się okazać wąskim gardłem. Tak więc, chociaż liczba przesyłanych komunikatów jest mala, czas potrzebny na ich przesłanie może być długi. W związku z tym w wielu systemach gwiaździstych stanowisko centralne służy wyłącznie do przelatczania komunikatów.



Rys. 15.5 Sieć w kształcie gwiazdy



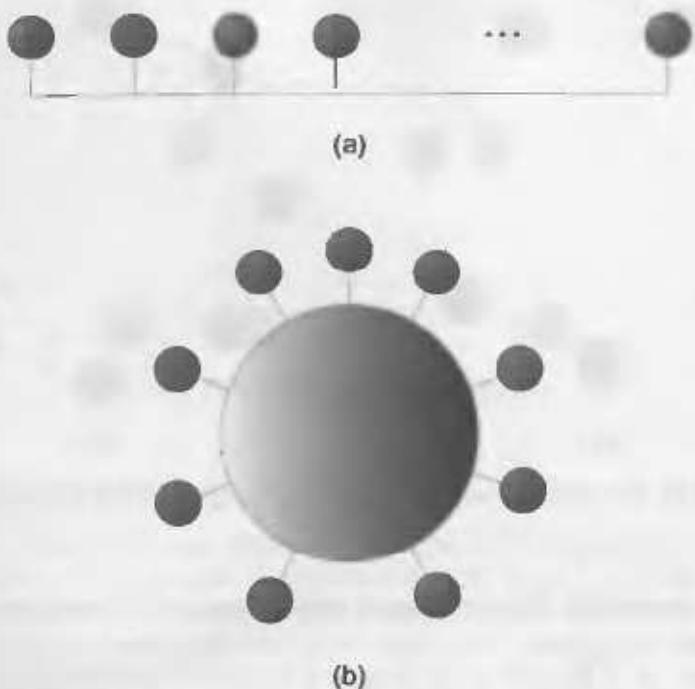
Rys. 15.6 Sieci pierścieniowe: (a) z pojedynczymi połączeniami, (b) z podwójnymi połączeniami

Jeśli stanowisko centralne ulegnie uszkodzeniu, to sieć będzie całkowicie podzielona.

15.3.5 Sieci pierścieniowe

W sieci pierścieniowej (ang. *ring network*) każde stanowisko jest fizycznie połączone z dokładnie dwoma innymi stanowiskami (rys. 15.6a). Pierścień może być jednokierunkowy lub dwukierunkowy. W pierścieniu jednokierunkowym stanowisko może przesyłać informację tylko do jednego sąsiada. Wszystkie stanowiska muszą wysyłać informacje w tym samym kierunku. W pierścieniu dwukierunkowym stanowisko może przesyłać informację do obu sąsiadów. Koszt podstawowy pierścienia jest funkcją liniową liczby stanowisk. Koszt komunikacji może być jednak wysoki. Komunikat z jednego stanowiska do drugiego podróżuje naokoło pierścienia, zanim dotrze do miejsca przeznaczenia. W pierścieniu jednokierunkowym może to wymagać wykonania $n - 1$ przesłan. W dwukierunkowym pierścieniu potrzeba co najwyżej $n/2$ przesłan.

W pierścieniu dwukierunkowym podział sieci powstaje dopiero w wyniku awarii dwu łącz. W jednokierunkowym pierścieniu awaria pojedynczego stanowiska (lub łącza) spowoduje podzielenie sieci. Jednym ze środków zapobiegawczych jest rozszerzenie sieci o podwojone łącz, jak na rys. 15.6b. Sieć pierścieniowa z żetonem (ang. *token ring*) firmy IBM jest siecią pierścieniową.



Rys. 15.7 Sieć szynowa: (a) szyna liniowa, (b) szyna pierścieniowa

15.3.6 Sieci z szynami wielodostępnymi

W sieci z szyną wielodostępną (ang. *multiaccess bus networks*) istnieje jedno łącze dzielone (szyna – ang. *bus*)*. Wszystkie stanowiska systemu są bezpośrednio do niego dołączone. Szyna może biec wzdłuż linii prostej (rys. 15.7a) lub tworzyć pierścień (rys. 15.7b). Stanowiska komunikują się ze sobą przez to łącze. Koszt podstawowy sieci jest funkcją liniową liczby stanowisk. Koszt komunikacji jest dość niski, chyba że łącze stanie się wąskim gardłem. Zauważmy, że ukształtowanie tej sieci jest podobne do sieci w kształcie gwiazdy z wyspecjalizowanym stanowiskiem centralnym. Awaria jednego stanowiska nie zakłóca komunikacji między pozostałymi stanowiskami. Jednak gdy uszkodzeniu ulegnie łącze, wówczas sieć będzie całkowicie podzielona. We wszechobecnej sieci Ethernet, tak pospolitej w wielu instytucjach na całym świecie, zastosowano szynę wielodostępną.

* Inaczej: magistrala. – Przyp. tłum.

15.3.7 Sieci mieszane

Często łączy się ze sobą sieci różnych typów. Na przykład w obrębie stanowiska może być używana sieć z szyną wielodostępną, w rodzaju sieci Ethernet, a między stanowiskami może być zastosowane połączenie hierarchiczne. Komunikacja w takich środowiskach może być dość zawiła z powodu konieczności tłumaczenia między sobą wielu protokołów; bardziej złożone jest też wyznaczanie tras dla danych.

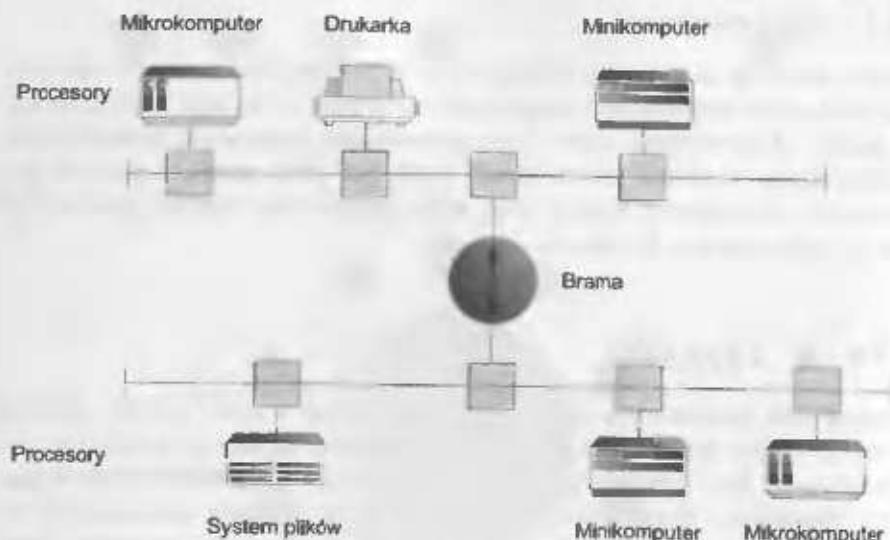
15.4 ■ Typy sieci

Istnieją dwa podstawowe typy sieci: *sieci lokalne* i *sieci rozległe*. Główna różnica między nimi polega na tym, w jaki sposób są one geograficznie rozmieszczone. Sieci lokalne są złożone z procesorów rozmieszczonych w małych obszarach – w pojedynczych budynkach lub w grupie sąsiadujących ze sobą budynków. Sieci rozległe są układami autonomicznych procesorów rozprzestrzenionych na dużych obszarach geograficznych (np. takich jak Stany Zjednoczone). Te różnice implikują dużą rozmaistość szybkości i niezawodności sieci komunikacyjnych i rzutują na projekt rozproszonego systemu operacyjnego.

15.4.1 Sieci lokalne

Sieci lokalne (ang. *local-area networks* – LANs) pojawiły się we wczesnych latach siedemdziesiątych jako substytuty wielkich, skoncentrowanych systemów komputerowych (ang. *mainframe*). Okazało się, że w wielu przedsięwzięciach pewna liczba małych komputerów przeznaczonych do odrębnych zastosowań jest ekonomiczniejsza niż jeden wielki system. Ponieważ każdy mały komputer potrzebował na ogół pełnego zestawu urządzeń peryferyjnych (takich jak dyski i drukarki), w poszczególnych zaś przedsięwzięciach występuowało swoiste zapotrzebowanie na korzystanie ze wspólnych danych, naturalnym krokiem było połączenie tych małych systemów w sieć.

Sieci lokalne projektuje się zazwyczaj z myślą o pokryciu niewielkiego terenu (jak osobny budynek lub obszar kilku przyległych do siebie budynków). Są one z zasady używane w biurach. Ponieważ wszystkie stanowiska w takim systemie znajdują się niedaleko od siebie, dąży się do tego, aby łączności komunikacyjne były szybkie i miały niskie wskaźniki błędów w porównaniu z ich odpowiednikami w sieciach rozległych. W celu osiągnięcia większych szybkości i niezawodności są potrzebne wysokiej jakości (a więc drogie) kable. Zdarza się, że kable takie są używane wyłącznie do przesyłania danych



Rys. 15.8 Sieć lokalna

sieciowych. W przypadkach dużych odległości koszt stosowania wysokiej jakości kabli staje się znaczny, co eliminuje możliwość stosowania kabli na zasadzie wyłączności.

Najpopularniejszymi łączami w sieci lokalnej są: kabel spleciony z dwóch żył, czyli tzw. *skrętka* (ang. *twisted pair*), oraz *kabel światłowodowy* (ang. *fiber optic*). Najpowszechniej spotykane konfiguracje to szyna wielodostępna oraz sieci pierścieniowe i gwiazdiste. Szybkość komunikacji waha się w granicach od 1 Mbit/s w sieciach takich, jak Appletalk lub działających w paśmie podczerwieni, do 1 Gbit/s w gigabitowej sieci Ethernet. Najczęściej stosowaną szybkością jest 10 Mbit/s. Taką szybkością przesyłania ma sieć Ethernet typu *10BaseT*. Sieć Ethernet typu *100BaseT* wymaga kabla wyższej jakości, za to działa z szybkością 100 Mbit/s i coraz bardziej się upowszechnia. Rośnie udział na rynku sieci FDDI* wyposażonych w światłowody. Działają one na zasadzie przekazywania żetonu, a ich szybkości przekraczają 100 Mbit/s.

Typowa sieć LAN może składać się z pewnej liczby różnych komputerów – od dużych komputerów stacjonarnych, aż po komputery podręczne (ang. *laptops*) lub komputerowe notatniki (ang. *personal digital assistant – PDA*) – rozmaitych dzielonych urządzeń zewnętrznych (jak drukarki laserowe

* Światłowodowy, rozproszony interfejs danych (ang. *fiber distributed data interface*).
– Przyp. tłum.

lub jednostki taśmy magnetycznej) oraz jednej lub więcej *bram** (ang. *gateways*), czyli specjalizowanych procesorów, umożliwiających dostęp do innych sieci (rys. 15.8). Do budowy sieci lokalnych powszechnie stosuje się schemat Ethernet. W sieci Ethernet nie występuje centralny sterownik, ponieważ sieć ta tworzy szynę wielodostępną, co ułatwia dołączanie do niej nowych komputerów.

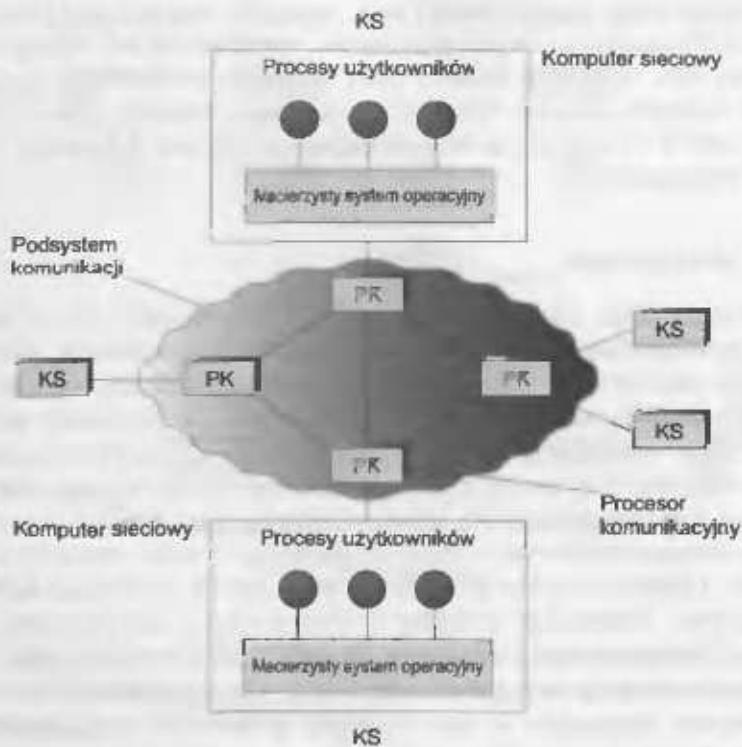
15.4.2 Sieci rozległe

Sieci rozległe (ang. *wide-area networks* – WAN) pojawiły się w późnych latach sześćdziesiątych, głównie jako akademickie konstrukcje badawcze, mające na celu umożliwienie wydajnej komunikacji między stanowiskami, tak by szeroki krąg użytkowników mógł wygodnie i ekonomicznie korzystać ze wspólnego sprzętu i oprogramowania. Pierwszą siecią, którą zaprojektowano i wykonano, była sieć *Arpanet*. Prace nad siecią Arpanet rozpoczęto w 1968 r. Czterostanowiskowa, eksperymentalna sieć Arpanet rozrosła się w sieć o zasięgu światowym – Internet, skupiający tysiące systemów komputerowych. Ostatnio na rynku pojawiła się także pewna liczba sieci komercyjnych. System Telenet jest dostępny w obrębie części kontynentalnej USA, z systemu Datapac można korzystać w Kanadzie. Sieci te umożliwiają swoim użytkownikom dostęp do bogatych zasobów sprzętowych i programowych.

Ponieważ stanowiska w sieci rozległej są fizycznie rozmieszczone na wielkich obszarach geograficznych, ich łącza komunikacyjne są z założenia względnie powolne i zawodne. Typowymi łączami są linie telefoniczne, łącza mikrofalowe i kanały satelitarne. Łącza komunikacyjne są nadzorowane przez specjalne procesory komunikacyjne (rys. 15.9), które odpowiadają za określanie interfejsu służącego do komunikowania się stanowisk w sieci oraz za przekazywanie informacji między różnorodnymi stanowiskami.

Jako przykład rozważmy sieć rozległą Internet. System ten umożliwia komunikowanie się ze sobą komputerów znajdujących się w geograficznie odseparowanych miejscach. Komputery te różnią się na ogół między sobą pod względem typu, szybkości, długości słowa, systemu operacyjnego itd. Komputery takie znajdują się z reguły w sieciach lokalnych, które są połączone do sieci Internet za pomocą sieci regionalnych. Sieci regionalne, takie jak NSFnet w północno-wschodniej części USA, są połączone za pomocą tzw. *ruterów* (omawiamy je w p. 15.5.2), tworząc sieć o zasięgu światowym. Łączność w sieciach często opiera się na systemie usług telefonicznych, zwanym T1, który umożliwia przesyłanie danych z szybkością 1,544 Mbit/s przez

* Inne spotykane określenia tego elementu sieci to: *wrótka*, *furtka*, *przepust*, *przejście*.
– Przyp. tłum.



Rys. 15.9 Procesory komunikacyjne w sieci rozległej

linię dzierżawioną. Stanowiska wymagające szybszego dostępu do sieci Internet łączy się za pomocą zwielenkrotnionych łącz T1, które – działając równolegle – mają większą przepustowość. Na przykład łącz T3 składa się z 28 łącz T1, a jego szybkość przesyłania wynosi 45 MB/s. Rutery sterują trasami komunikatów przesyłanych przez sieć. Wyznaczanie tras może odbywać się dynamicznie, aby zwiększać wydajność komunikacji, lub statycznie – w celu zmniejszenia ryzyka utraty bezpieczeństwa lub umożliwienia naliczania opłat komunikacyjnych.

W innych sieciach rozległych jako podstawowego środka komunikacji używa się standardowych linii telefonicznych. Urządzenia, które przyjmują dane cyfrowe od komputera i zamieniają je na sygnały analogowe przesyłane systemem telefonicznym, nazywają się *modemami*^{*}. Modem w stanowisku odbiorczym zamienia sygnały analogowe z powrotem na sygnały cyfrowe, umożliwiając przyjęcie danych w miejscu docelowym. Uniksowa sieć nowin

* Nazwa pochodzi od złożenia słów: modulator-demodulator. – Przyp. tłum.

UUCP^{*} pozwala systemom łączyć się ze sobą o określonych porach za pośrednictwem modemów w celu wymiany komunikatów. Komunikaty są wtedy kierowane do innych, sąsiednich systemów i w ten sposób rozpowszechniane we wszystkich komputerach w sieci (komunikaty publiczne) lub przekazywane do miejsc przeznaczenia (komunikaty prywatne). Sieci rozległe są z reguły wolniejsze niż sieci lokalne; ich szybkości przesyłania wahają się w przedziale od 1200 bit/s do ponad 1 Mbit/s.

Metoda UUCP jest gwałtownie zastępowana przez protokół PPP (ang. *point-to-point protocol*) komunikacji od punktu do punktu. Protokół PPP jest odmianą protokołu IP (zob. dalej). Działa on na połączeniach modemowych, umożliwiając pełne połączenie do sieci Internet komputerów domowych.

15.5 ■ Komunikacja

Skoro omówiliśmy już fizyczne aspekty sieci, przejdziemy do wewnętrznego ich działania. Projektant sieci komunikacyjnej musi wziąć pod uwagę następujących pięć kwestii:

- **Nazewnictwo i tłumaczenie nazw:** W jaki sposób dwa procesy odnajdują się w celu komunikacji?
- **Strategie wyznaczania tras:** Którędy komunikaty są przesyłane przez sieć?
- **Strategie postępowania z pakietami:** Czy pakiety są wysyłane pojedynczo, czy grupowo?
- **Strategie połączeń:** Jak dwa procesy wysyłają ciągi komunikatów?
- **Współzawodnictwo:** W jaki sposób rozwiązywać konfliktowe żądania wobec sieci, która jest przecież zasobem dzielonym?

Zagadnienia te omawiamy po kolej w punktach od 15.5.1 do 15.5.5.

15.5.1 Nazewnictwo i tłumaczenie nazw

Pierwszym elementem komunikacji sieciowej jest sposób nazywania systemów występujących w sieci. Aby procesy na stanowiskach A i B mogły wymieniać między sobą informacje, muszą się jakoś identyfikować. W obrębie systemu komputerowego każdy proces ma swój identyfikator, który może

^{*}Z ang. *UNIX-to-UNIX copy*: kopiowanie między systemami UNIX – Przyp. tłum.

służyć jako adres komunikatu. Systemy w sieci nie korzystają ze wspólnej pamięci, więc na początku nie mają żadnych informacji o komputerze, w którym znajduje się proces docelowy, a nawet nie są zorientowane, czy proces taki istnieje.

Aby rozwiązać ten problem, procesy w zdalnych systemach z reguły identyfikuje się za pomocą pary *<nazwa komputera, identyfikator>*, przy czym „nazwa komputera” jest w sieci jednoznaczna, a „identyfikator” może być identyfikatorem procesu lub inną niepowtarzalną liczbą w obrębie danego komputera. Nazwa komputera jest zazwyczaj identyfikatorem alfanumerycznym, a nie liczbą, aby użytkownicy mogli łatwiej się nią posługiwać. Na przykład stanowisko A mogłoby mieć komputery o nazwach: „iza”, „ewa”, „zuza” i „jowita”. Nazwę „zuza” zapamiętuje się z pewnością łatwiej niż „128|483|1100”.

Nazwy są wygodne do używania przez ludzi, jednak dla komputerów szybsze i prostsze są liczby. Z tej przyczyny musi istnieć *mechanizm tłumaczenia* (ang. *resolution*) nazwy komputera w sieci na jego identyfikator, na podstawie którego sprzęt sieciowy określi położenie systemu docelowego. Mechanizm ten jest podobny do wiązania nazwy z adresem, które występuje podczas komplikowania programu, jego konsolidowania, ładowania i wykonywania (zob. rozdz. 8). W przypadku nazw komputerów sieciowych istnieją dwie możliwości. Po pierwsze, każdy komputer może mieć plik danych zawierający nazwy i adresy wszystkich innych komputerów osiągalnych przez sieć (przypomina to wiązanie nazw podczas komplikacji). Niedogodnością tego modelu jest konieczność aktualniania plików danych we wszystkich komputerach sieci w przypadku dodania do niej (lub usunięcia) jakiegokolwiek komputera. Druga możliwość polega na upowszechnianiu informacji między systemami w sieci. W sieci takiej trzeba zatem stosować protokół upowszechniania i odzyskiwania informacji. Ta metoda jest podobna do wiązania adresów podczas wykonywania programu. Pierwszej metody użyto początkowo w sieci Internet, jednak w miarę rozrastania się Internetu nie dało się jej obronić, wobec czego zastosowano drugą metodę – domenowe usługi nazewnicze (ang. *domain name service* – DNS), z której korzysta się do dzisiaj.

Uslugi nazewnicze DNS określają strukturę nazw komputerów sieciowych, jak również definiują tłumaczenie nazw na adresy. Komputery w sieci Internet są poadresowane logicznie za pomocą wielozłonowych nazw. Elementy w nazwie występują w kolejności od najbardziej szczegółowej części adresu do części najbardziej ogólnej. Poszczególne części nazwy są pooddzielane kropkami. Na przykład nazwa „bob.cs.brown.edu” odnosi się do komputera „bob” w Instytucie Informatyki uczelni Brown University. Ogólnie biorąc, można

* Czyli rozbioru (gramatycznego) nazw. – Przyp. tłum.

powiedzieć, że system wydobywa adresy, analizując części nazwy komputera w odwrotnym porządku. Każdy fragment nazwy odnosi się do *serwera nazw* (ang. *name server*), czyli po prostu do procesu w systemie usług nazewniczych, który przyjmuje nazwę domeny i zwraca adres odpowiedzialnego za nią serwera nazw. W ostatnim kroku następuje kontakt z serwerem zawierającym nazwę poszukiwanego komputera, w wyniku czego następuje przekazanie identyfikatora tego komputera w sieci. W naszym przykładzie proces zgłoszający zapotrzebowanie na komunikację z komputerem „bob.cs.brown.edu”, spowoduje wykonanie przez system następujących czynności:

1. Jądro systemu A zwraca się do serwera nazw domeny „edu” z prośbą o adres serwera nazw domeny „brown.edu”. Serwer nazw domeny „edu” musi występować pod znanym adresem, aby można go było odpytywać. (Inne domeny szczytowych poziomów to m.in. „com” – dla stanowisk komercyjnych, „org” – dla instytucji. Na każdy kraj przylączony do sieci przypada również po jednej domenie przeznaczonej w tym przypadku dla systemów krajowych, a nie instytucji).
2. Serwer nazw „edu” zwraca adres komputera, w którym rezyduje serwer nazw domeny „brown.edu”.
3. Jądro systemu A zapytuje wówczas serwer nazw pod tym adresem o adres serwera nazw domeny „cs.brown.edu”.
4. Kiedy nowy adres zostanie przekazany, wtedy wysłanie zamówienia z pytaniem o adres komputera „bob.cs.brown.edu” kończy się zwróceniem adresu internetowego poszukiwanego komputera, czyli jego identyfikatora (np. 128.148.31.100).

Ten protokół może się wydawać mało wydajny, jednak każdy serwer nazw utrzymuje na ogół pamięć podręczną, aby proces ten przyspieszyć. Na przykład serwer nazw „edu” mógłby mieć adres domeny „brown.edu” w pamięci podręcznej, mógłby więc poinformować system A, że może przetłumaczyć dwie części nazwy i zwrócić wskaźnik do serwera nazw domeny „cs.brown.edu”. Rzecz jasna zawartość takich pamięci podręcznych musi być odświeżana z upływem czasu, gdy serwer nazw jest przenoszony lub gdy zmienia się jego adres. Usługa ta jest w istocie tak ważna, że realizujący ją protokół zawiera wiele optymalizacji i wiele zabezpieczeń. Rozważmy, co by się stało, gdyby podstawowy serwer nazw domeny „edu” uległ awarii. Mogliby to uniemożliwić przetłumaczenie nazw wszystkich komputerów w domenie „edu” i spowodować ich niedostępność! Aby tego uniknąć, stosuje się pomocnicze, zapasowe serwery nazw, które dublują zawartości serwerów podstawowych.

Przed wprowadzeniem domenowych usług nazewniczych wszystkie komputery w sieci Internet musiały mieć kopie pliku, który zawierał nazwy i adresy wszystkich komputerów w sieci. Każde zmiana w tym pliku musiała być zarejestrowana w pewnym stanowisku (komputer SRI-NIC), a wszystkie inne komputery musiały okresowo kopiować aktualny plik z tego stanowiska, aby móc się kontaktować z nowymi systemami lub uwzględniać zmiany w adresach komputerów sieciowych. W systemie domenowych usług nazewniczych stanowisko każdego serwera nazw odpowiada za aktualnianie informacji o komputerach w danej domenie. Na przykład odnotowanie każdej zmiany komputera w Brown University należy do obowiązków serwera nazw domeny „brown.edu” i nie trzeba o niej komunikować gdziekolwiek indziej. Poszukiwania wykonywane przez system DNS automatycznie doprowadzą do odzyskania aktualionej informacji dzięki bezpośredniemu kontaktowi z domeną „brown.edu”. Domeny mogą zawierać autonomiczne poddomeny, co umożliwia dalszy podział odpowiedzialności za nazwy komputerów sieciowych i zmiany ich identyfikatorów.

Mówiąc ogólnie, do obowiązków systemu operacyjnego należy przyjmowanie od procesów komunikatów przeznaczonych dla komputera określonego przez parę *<nazwa komputera, identyfikator>* i przekazywanie ich do procesu nazwanego za pomocą tej pary. Ta wymiana w żadnym wypadku nie jest zadaniem prostym – omawiamy ją w p. 15.5.4.

15.5.2 Strategie wyboru trasy

Jeśli proces z węzła A chce skomunikować się z procesem w węźle B, to jaką drogą będzie przesłany komunikat? Jeśli istnieje tylko jedna fizyczna droga od A do B (np. w sieci hierarchicznej lub gwiązdistej), to komunikat musi zostać przesłany tą drogą. Jednak gdy istnieje wiele fizycznych dróg od A do B, wówczas mamy do czynienia z kilkoma możliwymi trasami. Każde stanowisko ma *tablicę tras* (ang. *routing table*) wskazującą alternatywne drogi, których można użyć przy przesyłaniu komunikatu do innych stanowisk. Tabela może zawierać informacje o szybkości i koszcie różnych połączeń oraz może być aktualizowana stosownie do potrzeb – ręcznie lub za pomocą programów, które wymieniają informacje o trasach. Trzy najpopularniejsze schematy określania tras to: trwałe wyznaczenie trasy, metoda obwodu wirtualnego oraz dynamiczne wyznaczanie trasy.

- **Trwałe wyznaczenie trasy** (ang. *fixed routing*): Droga z A do B jest określona z góry i nie zmienia się, chyba że korzystanie z niej uniemożliwi awaria sprzętu. Wybiera się zwykle najkrótszą drogę, aby koszty komunikacji były jak najmniejsze.

- **Metoda obwodu wirtualnego** (ang. *virtual circuit*): Droga z A do B jest ustalana na czas trwania jednej sesji. Komunikaty wysyłane z A do B podczas różnych sesji mogą podróżować różnymi drogami. Sesja może spowodować się do przesłania pliku lub trwać tyle co zdalna sesja konwersacyjna.
- **Dynamiczne wyznaczanie trasy** (ang. *dynamic routing*): Drogę przesyłania komunikatu ze stanowiska A do stanowiska B obiera się tuż przed wysłaniem komunikatu. Ponieważ decyzja ta jest podejmowana dynamicznie, poszczególnym komunikatom mogą zostać przypisane różne drogi. Stanowisko A decyduje, by przesyłać komunikat do stanowiska C; C z kolei posyła go do D itd. W końcu jakieś stanowisko dostarczy komunikat do B. Na ogół stanowisko posyła komunikat do innego stanowiska tym lączem, które w danym czasie jest najmniej używane.

Miedzy tymi trzema schematami istnieje swoista równowaga. Trasy stałej nie można przystosować do możliwości wystąpienia awarii lub zmian obciążenia. Innymi słowy, jeśli ustali się drogę między A i B, to komunikaty muszą być nią kierowane nawet wtedy, gdy droga ta jest niesprawna lub intensywnie używana, podczas gdy inna możliwa droga jest mało obciążona. Można tej niedogodności częściowo zaradzić, stosując obwody wirtualne, a dynamiczne wyznaczanie trasy usuwa ją w zupełności. Trasy stałe i obwody wirtualne gwarantują, że komunikaty z A do B będą nadchodziły w porządku, w którym zostały wysłane. Komunikaty przesyłane trasami wyznaczanymi dynamicznie mogą nadchodzić nie po kolej. Można temu zaradzić przez dodawanie numeru porządkowego do każdego komunikatu.

Zauważmy, że dynamiczne wyznaczanie tras jest najbardziej skomplikowane w przygotowaniu i działaniu. Jednakże w złożonych środowiskach okazuje się ono najlepszym rozwiązaniem. W systemie UNIX jest możliwe zarówno ustalanie tras na stałe do użytku między komputerami wewnątrz prostej sieci, jak i dynamiczne wyznaczanie tras w przypadku skomplikowanych środowisk sieciowych. Można także połączyć obie metody. W ramach jednego stanowiska wystarczy, aby komputery знаły drogę do systemu, który łączy sieć lokalną z innymi sieciami (np. z sieciami obejmującymi swym zasięgiem przedsiębiorstwa lub z siecią Internet). Komputer dokonujący takich połączeń jest nazywany *bramą* (ang. *gateway*). Komputery sieci lokalnej mogą zatem mieć wyznaczoną stałą trasę do komputera-bramy. Bramy mogą wyznaczać trasy dynamicznie, aby umożliwiać dotarcie do dowolnego komputera w pozostałej części sieci.

Ruterem (ang. *router*)^{*} określa się jednostkę systemu komputerowego odpowiedzialną za wyznaczanie tras komunikatów. Funkcje rутera moze pełnić

* Czyli „traserem” – tym co wyznacza trasę. – Przyp. tłum.

dowolny komputer sieciowy zaopatrzony w oprogramowanie do wyboru tras lub wyspecjalizowane urządzenie. W każdym przypadku ruter musi mieć co najmniej dwa połączenia do sieci, w przeciwnym bowiem razie nie miałby dokąd kierować komunikatów. Ruter decyduje, czy komunikat, który otrzymuje z jednej sieci, należy przekazać do jakiejś innej sieci, z którą ma połączenie. Decyzję tę podejmuje na podstawie analizy internetowego adresu przeznaczenia komunikatu. Ruter przegląda własną tablicę, aby określić położenie docelowego komputera lub przynajmniej znajdującą się na drodze do niego sieć i móc tam skierować komunikat. W przypadku tras statycznych tablicę tę uaktualnia się tylko ręcznie (do rutera wprowadza się nowy plik). Jeśli trasy są wybierane dynamicznie, to routery wykonują protokół *wyznaczania tras* (ang. *routing protocol*), informując się wzajemnie o zmianach występujących w sieci i umożliwiając sobie automatyczne aktualizowanie tablic tras.

15.5.3 Postępowanie z pakietami

Komunikaty mają na ogół zmienną długość. Aby uprościć projekt systemu, komunikację powszechnie implementuje się przy użyciu komunikatów o stałej długości nazywanych *pakietami*, *ramkami* lub *datagramami* (ang. *packets*, *frames*, *datagrams*). Informacja mieszcząca się w jednym pakiecie może być wysłana do miejsca przeznaczenia jako *komunikat bezpołączeniowy* (ang. *connectionless*). Komunikat bezpołączeniowy może być zawodny (ang. *unreliable*), co oznacza, że nadawcy nie gwarantuje się, ani nie oznajmia, czy pakiet dotarł do celu. Pakiety mogą być też *niezawodne* (ang. *reliable*), co zwykle osiąga się przez wysłanie z miejsca docelowego pakietu zawiadomującego o dotarciu pakietu źródłowego do celu*. (Oczywiście pakiet powrotny może zaginąć po drodze). Jeśli komunikat jest za długi, aby pomieścić się w jednym pakiecie lub jeśli pakiety muszą przechodzić tam i z powrotem między komunikującymi się stronami, to w celu niezawodnej wymiany wielu pakietów należy ustanowić *połączenie* (ang. *connection*).

15.5.4 Strategie połączeń

Skoro do miejsc przeznaczenia można już dostarczać komunikaty, w procesach można tworzyć „sesje” komunikacji w celu wymiany informacji. Istnieje kilka różnych sposobów na połączenie par procesów, które mają się komunikować przez sieć. Oto trzy najpopularniejsze schematy: komutowanie łączy, komutowanie komunikatów oraz komutowanie pakietów.

* Zauważmy, że cecha niezawodności bezpośrednio dotyczy protokołu dostarczania pakietów, a nie ich samych. – Przyp. tłum.

- **Komutowanie łączy:** W celu umożliwienia dwóm procesom komunikacji ustala się między nimi stałe fizyczne połączenie. Łączy to zostaje przydzielone na czas trwania komunikacji i żaden inny proces nie może go użyć w tym okresie (nawet jeśli dane dwa procesy przez chwilę nie wymieniają komunikatów). Schemat ten jest podobny do używanego w telefonii. Z chwilą otwarcia linii komunikacyjnej między dwoma aparatami (tzn. abonent A dzwoni do abonenta B), nikt inny nie może użyć tej linii dopóki, dopóki łączność nie zostanie jawnie zakończona (np. w jednym z aparatów odwieszono słuchawkę).
- **Komutowanie komunikatów:** Jeśli dwa procesy chcą się komunikować, to ustala się czasowe łącze na czas przesyłania jednego komunikatu. Łącza fizyczne są przydzielane dynamicznie korespondentom – stosownie do potrzeb – i tylko na krótki czas. Każdy komunikat jest blokiem danych wyposażonym w pewne informacje systemowe (takie jak miejsce nadania, adres przeznaczenia i kody korygujące), które umożliwiają sieci komunikacyjnej poprawne dostarczenie go do punktu docelowego. Ten schemat jest podobny do systemu pocztowego. Każdy list jest komunikatem zawierającym zarówno adres docelowy, jak i adres nadawcy (zwrotny). Zwrócmy uwagę, że tym samym łączem można wysyłać wiele komunikatów (od różnych użytkowników).
- **Komutowanie pakietów:** Komunikat logiczny można podzielić na pewną liczbę pakietów. Każdy pakiet może być osobno wysyłany do miejsca przeznaczenia, musi więc zawierać razem z danymi adres miejsca nadania i adres docelowy. Każdy pakiet może przechodzić przez sieć inną drogą. Po przybyciu do celu pakiety muszą zostać z powrotem połączone w komunikat.

Miedzy tymi trzema schematami istnieje oczywista równowaga. Komutowanie łączy wymaga czasu na przygotowanie do pracy, ale wysyłanie każdego komunikatu jest w tej metodzie tańsze; może jednak dochodzić do marnowania przepustowości sieci. Komutowanie komunikatów i pakietów wymaga z kolei mniej czasu na rozpoczęcie działania, ale zwiększa koszt każdego komunikatu. W przypadku komutowania pakietów każdy komunikat musi być ponadto podzielony na pakiety, a potem zestawiony z nich na nowo. W sieciach danych stosuje się najczęściej komutowanie pakietów, gdyż zapewnia ono najlepsze wykorzystanie przepustowości sieci, a danym nie przeszkadza to, że są dzielone na pakiety, byc moze przesyłane różnymi drogami i składane z powrotem u celu. W przypadku sygnału dźwiękowego (powiedzmy – w telefonii) takie postępowanie mogłoby wprowadzić duże zakłócenia przy braku staranności.

15.5.5 Rywalizacja

W zależności od kształtu sieci jedno łącze może łączyć więcej niż dwa stanowiska sieciowe, jest więc możliwe, że kilka stanowisk będzie chciało jednocześnie przesyłać przez nie informacje. Trudność ta pojawia się głównie w sieciach pierścieniowych lub szynach wielodostępnego. W takim przypadku przesyłane informacje mogłyby ulec zniekształceniu i trzeba by je usunąć. Stanowisko musi być powiadamiane o takim zdarzeniu, aby mogło powtórnie wysłać informację. Jeśli nie poczyni się żadnych kroków, to sytuacja taka może się powtarzać, powodując pogorszenie działania. Opracowano kilka metod unikania powtarzających się kolizji, a wśród nich takie jak: wykrywanie kolizji, przekazywanie żetonu oraz stosowanie przegródek na komunikaty.

- **CSMA/CD:** Przed wysłaniem komunikatu przez łącze stanowisko musi nasłuchiwać, czy przez łącze nie jest właśnie przesyłany inny komunikat. Ta metoda nosi nazwę *wielodostępu do łącza z badaniem jego stanu* (ang. *carrier-sense with multiple access* – CSMA). Jeśli łącze jest wolne, to stanowisko może rozpoczęć przesyłanie komunikatu. W przeciwnym razie musi zaczekać (kontynuując nasłuch), aż łącze się zwolni. Jeśli dwa lub więcej stanowisk rozpoczęte przesyłanie komunikatów w dokładnie tym samym czasie (każde przekonane, że żadne inne stanowisko nie używa łącza), to zarejestrują one *wykrycie kolizji* (ang. *collision detection* – CD) i zaprzestaną przesyłania. Po losowo wybranym przedziale czasu każde stanowisko spróbuje od nowa. Zauważmy, że jeśli stanowisko A rozpoczęta przesyłanie komunikatu przez łącze, to musi nieustannie nasłuchiwać w celu wykrywania kolizji z komunikatami z innych stanowisk. Główną wadą tego podejścia jest to, że gdy system jest bardzo zjęty, może powstawać wiele kolizji, co pogorszy jego wydajność. Niemniej jednak metoda wielodostępu do łącza z badaniem jego stanu i wykrywaniem kolizji (CSMA/CD) została z powodzeniem zastosowana w systemie Ethernet – najpopularniejszym systemie sieciowym. (Protokół Ethernet jest zdefiniowany przez standard IEEE 802.3). Aby ograniczyć liczbę kolizji, należy ograniczać liczbę komputerów w jednej sieci Ethernet. Powiększanie liczby komputerów w zagęszczonej sieci może pogorszać przepustowość sieci. Coraz to szybsze systemy potrafią wysyłać coraz więcej pakietów w jednostce czasu. Wskutek tego zmniejsza się liczbę systemów przypadających na segment sieci Ethernet, co ma umożliwić utrzymanie zadowalającej wydajności.
- **Przekazywanie żetonu:** Komunikat specjalnego typu – zwany *żetonem* (ang. *token*) – krąży bez przerwy w systemie (zwykle o strukturze pierścienia). Stanowisko, które chce przesyłać dane, musi czekać na nadanie że-

tonu. Wówczas usuwa żeton z pierścienia i rozpoczyna wysyłanie swoich komunikatów. Kiedy stanowisko zakończy swą rundę przekazywania komunikatów, ponownie przesyła żeton. To działanie pozwala z kolejnym stanowisku odebrać i usunąć żeton, i rozpoczęć wysyłanie swojego komunikatu. Gdy żeton zginie, system musi to wykryć i wygenerować nowy żeton. W tym celu zazwyczaj ogłasza się *elekcję* (ang. *election*), aby wybrać dokładnie jedno stanowisko, w którym będzie wytworzony nowy żeton. Algorytm elekcji prezentujemy w p. 18.6. Schemat przekazywania żetona został zaadaptowany przez systemy IBM i HP/Apollo. Zaletą sieci z przekazywaniem żetona jest stała wydajność. Podłączenie nowego systemu do sieci może wydłużyć czas oczekiwania na żeton, lecz nie powoduje znacznego spadku wydajności, co może się zdarzyć w sieci Ethernet. Jednak w sieciach słabo obciążonych technika Ethernet jest wydajniejsza, ponieważ systemy mogą wysyłać komunikaty w dowolnej chwili.

- **Przegródki na komunikaty:** Pewna liczba stałej długości przegródek na komunikaty krąży bez przerwy w systemie (zazwyczaj o strukturze pierścienia). W każdej przegródce może znaleźć się komunikat ustalonej długości oraz informacje sterujące (dotyczące jego źródła i miejsca przeznaczenia oraz stanu przegródki: czy jest pusta czy pełna). Stanowisko gotowe do wysyłania komunikatu musi czekać, aż nadjejdzie pusta przegródka. Wówczas wkłada do niej komunikat, określając odpowiednio informacje sterujące. Przegródka z komunikatem przemieszcza się potem w sieci. Gdy dociera do stanowiska, wówczas stanowisko to sprawdza informacje sterujące, aby określić, czy przegródka zawiera komunikat przeznaczony dla niego. Jeśli nie, to posyła przegródkę z komunikatem dalej. W przeciwnym razie stanowisko usuwa komunikat (z sieci), tak określając informacje sterujące, aby wskazywały, że przegródka jest pusta. Stanowisko może następnie użyć tej przegródki do wysłania własnego komunikatu lub ją zwolnić. Ponieważ przegródka może zawierać tylko komunikaty o stałej długości, pojedynczy logiczny komunikat trzeba będzie podzielić na pewną liczbę mniejszych pakietów, z których każdy będzie wysyłany w osobnej przegródce. Rozwiążanie takie zastosowano w eksperymentalnym systemie Cambridge Digital Communication Ring.

15.6 ■ Strategie projektowe

Przy projektowaniu sieci komunikacyjnej należy radzić sobie ze swoistą złożonością koordynowania asynchronicznych operacji komunikujących się w potencjalnie powolnym i podatnym na błędy środowisku. Jest również

istotne, aby podłączone do sieci systemy uzgodniły protokół lub zbiór protokołów określania nazw komputerów sieciowych, odnajdywania komputerów w sieci, ustanawiania połączeń itp. Problem projektowania (i związaną z nim implementację) można uprościć przez podzielenie go na kilka warstw. Każda warstwa w jednym systemie komunikuje się z równoważną jej warstwą w innych systemach. Kazda warstwa może mieć określone własne protokoły lub wyrażać podział logiczny. Protokoły mogą mieć realizację sprzętową lub programową. Na przykład trzy najniższe warstwy logiczne komunikacji między dwoma komputerami mogą być zaimplementowane sprzętowo. Zgodnie z zaleceniami Międzynarodowej Organizacji Normalizacyjnej (ang. *International Standards Organization – ISO*) rozważamy następujące warstwy:

- 1. Warstwa fizyczna:** Warstwa fizyczna odpowiada za obsługę zarówno mechanicznych, jak i elektrycznych szczegółów fizycznej transmisji strumieni bitów. Komunikujące się systemy muszą w warstwie fizycznej uzgodnić elektryczną reprezentację cyfr dwójkowych 0 i 1, aby podczas przesyłania danych w postaci strumienia sygnałów elektrycznych odbiorca był w stanie interpretować je poprawnie jako dane binarne. Ta warstwa jest implementowana przez sprzęt sieciowy.
- 2. Warstwa łącza danych:** Warstwa łącza danych odpowiada za obsługę ramek, czyli pakietów o stałej długości; do jej zadań należy m.in. wykrywanie wszelkich błędów, które wystąpiły w warstwie fizycznej oraz ich usuwanie.
- 3. Warstwa sieciowa:** Warstwa sieciowa odpowiada za organizację połączeń i za określanie tras pakietów w sieci komunikacyjnej, w tym za obsługę adresów wychodzących pakietów, dekodowanie adresów nadchodzących pakietów i utrzymywanie informacji o trasach w celu odpowiedniego reagowania na zmieniające się obciążenia sieci. W tej warstwie działają routery.
- 4. Warstwa transportu:** Warstwa transportu odpowiada za dostęp do sieci na niskim poziomie i za przesyłanie komunikatów między klientami, w tym za dzielenie komunikatów na pakiety, dopilnowywanie porządku pakietów, kontrolowanie przepływu i generowanie adresów fizycznych.
- 5. Warstwa sesji:** Warstwa sesji odpowiada za implementację sesji, czyli protokołów komunikacyjnych między procesami. Na ogół są to protokoły określające bieżące reguły rejestrowania się na zdalnych komputerach oraz zasady przesyłania plików i poczty.
- 6. Warstwa prezentacji:** Warstwa prezentacji odpowiada za likwidowanie różnic w formatach między różnymi stanowiskami w sieci, w tym za

konwersję znaków oraz pracę w trybach połduplexowym i pełnoduplexowym (z echem znaków).

1. **Warstwa zastosowań:** Warstwa zastosowań odpowiada za bezpośrednią interakcję z użytkownikami: przesyła pliki, obsługuje protokoły zdalnych rejestracji i pocztę elektroniczną, jak również schematy rozpraszonych baz danych.

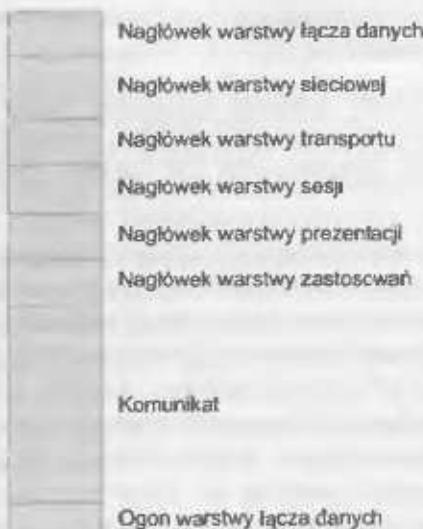
Zbiór powiązanych ze sobą protokołów nazywa się *stosem protokołów* (ang. *protocol stack*). Jednym ze standardowych zestawów protokołów jest stos protokołów ISO. Jak już wspomniano, każda warstwa stosu protokołów komunikuje się z logicznie równoważną jej warstwą w innych systemach. Jednak rozpatrując rzecz z fizycznego punktu widzenia, komunikat rozpoczyna się w warstwie zastosowań (lub powyżej tej warstwy), po czym następuje jego przekazywanie na coraz to niższe poziomy. Każda z pośredniczących warstw może zmienić komunikat i dołączyć do niego dane nagłówkowe dla równoważnej warstwy po stronie odbiorczej. Na koniec komunikat zostaje przetworzony przez warstwę danych sieci i przesłany w postaci jednego pakietu lub większej liczby pakietów (rys. 15.10). Te dane są przyjmowane przez warstwę łączą danych w systemie docelowym i komunikat wędruje w górę stosu protokołów, będąc w trakcie tej drogi analizowany, modyfikowany i pozbywany nagłówków.

W modelu ISO sformalizowano pewne wcześniejsze prace dotyczące protokołów sieciowych, a on sam pochodzi z końca lat siedemdziesiątych i nie jest szeroko używany. Część podstaw modelu ISO jest jeszcze starszym i powszechnie stosowanym stosem protokołów opracowanych pod systemem UNIX do użytku w sieci Arpanet (która przerodziła się w Internet).

Większość stanowisk internetowych wciąż komunikuje się za pomocą protokołu internetowego (ang. *Internet Protocol*), popularnie nazywanego IP. Usługi są implementowane powyżej protokołu IP za pośrednictwem bezpołączeniowego, protokołu datagramów użytkownika (ang. *User Datagram Protocol – UDP*)^{*} i połączniowego protokołu sterowania przesyaniem wraz z protokołem internetowym (ang. *Transmission Control Protocol/Internet Protocol*), powszechnie znanych pod nazwą TCP/IP^{**}. Stos protokołów TCP/IP ma mniej warstw niż model ISO. Teoretycznie powinien on być trudniejszy do implementacji, ponieważ łączy po kilka funkcji w każdej warstwie, ale zapewnia większą wydajność pracy w sieci, niż przy zastosowaniu protokołu ISO.

* Skrót UDP hywa też rozwijany jako *Universal Datagram Protocol* (uniwersalny protokół datagramowy). – Przyp. tłum.

** Czytelników zainteresowanych szczegółami tego szeroko stosowanego protokołu można odesłać do książki D. E. Comera. *Sieci komputerowe TCP/IP. Tom I. Zasady, protokoły i architektura*. Wydawnictwa Naukowo-Techniczne, Warszawa 1998. – Przyp. tłum.



Rys. 15.10 Komunikat sieci ISO

Protokół IP odpowiada za przesyłanie datagramów IP – podstawowej jednostki informacji – w sieci Internet sterowanej protokołami TCP/IP. W protokole TCP korzysta się z protokołu IP do transportowania niezawodnego strumienia danych między dwoma procesami. Zestaw UDP/IP tworzy protokół bezpołączeniowy, w którym nie gwarantuje się niezawodności. Protokół IP znajduje w nim zastosowanie do przesyłania pakietów, przy czym zaopatruje je w korekcję błędów i adres portu protokołu, aby określić proces w odległym systemie, dla którego pakiet jest przeznaczony.

15.7 ■ Przykład działania sieci

Powrócimy teraz do zagadnienia tłumaczenia nazw, poruszonego w p. 15.5.1, aby przyjrzeć się mu w kontekście stosu protokołów TCP/IP używanych w sieci Internet. Rozważmy czynności niezbędne do przesłania pakietu między komputerami znajdującymi się w różnych sieciach Ethernet.

Każdy komputer w sieci TCP/IP ma nazwę i związany z nią 32-bitowy numer internetowy (identyfikator komputera). Obie te dane muszą być jednoznaczne oraz posegmentowane. Ten drugi wymóg ma umożliwić zarządzanie przestrzenią nazw. Nazwa jest hierarchiczna (co wyjaśniliśmy w p. 15.5.1) – zawiera nazwę komputera oraz określa organizacje, w których posiadaniu znajduje się komputer. Identyfikator komputera jest podzielony na numer sieci i numer komputera. Proporcja tego podziału jest zmieniona, zależnie od

rozmnaru sieci. Z chwilą nadania sieci numeru przez administratorów Internetu przypisywanie identyfikatorów komputerom w obrębie tak oznaczonego miejsca może odbywać się dowolnie.

System nadawczy sprawdza swoje tablice tras, aby odnaleźć ruter, który skieruje pakiet na właściwą drogę. Aby przesłać pakiet z sieci źródłowej do sieci docelowej, routery korzystają z sieciowej części identyfikatora komputera. System docelowy odbiera wówczas pakiet. Pakiet może być kompletnym komunikatem lub tylko jego częścią, a wtedy trzeba będzie zebrać więcej pakietów, zanim nastąpi wtórne zestawienie z nich komunikatu i przekazanie go do warstwy TCP/UDP w celu przesłania do procesu-adresata.

Wiemy już, w jaki sposób pakiet przemieszcza się z sieci źródłowej do miejsca swojego przeznaczenia. A jak przebiega droga pakietu między nadawcą (zwykłym komputerem sieciowym lub routерem) a odbiorcą w obrębie sieci? Każde urządzenie Ethernet^{*} ma jednoznaczny, bajtowy numer, służący do jego adresowania. Dwa urządzenia komunikują się ze sobą tylko za pomocą tego numeru. Jądro generuje okresowo pakiet UDP zawierający identyfikator komputera oraz eternetowy numer systemu. Pakiet ten jest *rozgłoszany* do wszystkich innych systemów w danej sieci Ethernet. Komunikat rozgłoszany ma specjalny adres sieciowy (z reguły adres maksymalny), aby sygnalizować, że zawierający go pakiet powinien być przyjęty i przetworzony przez każdy komputer w sieci. Nie jest on retransmitowany przez bramy, więc odbierają je tylko systemy w sieci lokalnej. Po odebraniu tego komunikatu (pakietu UDP) każdy komputer wydobywa z niego parę identyfikatorów i zapamiętuje ją podręcznie w wewnętrznej tablicy. Ciąg tych działań określa się jako *protokół tłumaczenia adresu* (ang. *Address Resolution Protocol – ARP*). Wpisy w pamięci podręcznej są *postarzane*, tak aby po pewnym czasie następowało ich usuwanie z pamięci, jeśli nie nadchodzi odnawiające je komunikaty rozgłoszane. Dzięki temu po jakimś czasie komputery usunięte z sieci zostają „zapomniane”.

Z chwilą gdy urządzenie Ethernet oznajmi swój identyfikator komputera i adres komunikacji może się rozpoczęć. Proces może określić nazwę komputera, z którym chce nawiązać łączność. Jądro pobiera tę nazwę i za pomocą przeszukiwania bazy DNS ustala internetowy numer adresata. Komunikat zostaje przekazany z warstwy zastosowan do warstw programowych i warstwy sprzętowej. Dochodząc do warstwy sprzętowej, pakiet (lub pakiety) jest zaopatrzony w nagłówek z adresem eternetowym oraz w zakończenie wskażające jego koniec i zawierające *sumę kontrolną* (ang. *checksum*) służącą do wykrywania jego uszkodzeń (rys. 15.11). Pakiet ten jest umieszczany w sieci

^{*} Układ ten, jak wiele innych podzespołów komputerowych, jest też nazywany „kartą” (tu: Ethernet). – Przyp. tłum.

Bajty			
7	Preambuła - początek pakietu	Każdy bajt w postaci 10101010	
1	Początek ogranicznika ramki	Układ bitów 101D1011	
2 lub 6	Adres przeznaczenia	Adres eternetowy lub adres rozgłoszania	
2 lub 6	Adres źródłowy	Adres eternetowy	
2	Długość sekcji danych	Długość w bajtach	
0-1500	Dane	Dane komunikatu	
0-46	Wyrównanie (niekonieczne)	Komunikat musi być dłuższy niż 63 bajty	
4	Suma kontrolna ramki	Służy do wykrywania błędów	

Rys. 15.11 Pakiet eternetowy

przez urządzenie Ethernet (kartę sieciową). Zauważmy, że część pakietu przeznaczona na dane może zawierać fragment danych komunikatu lub nawet cały komunikat w jego pierwotnej postaci, lecz również może zawierać wchodzące w skład komunikatu elementy nagłówków górnych warstw protokołu. Innymi słowy, wszystkie części oryginalnego komunikatu muszą być przesłane od źródła do miejsca przeznaczenia, a wszystkie nagłówki z poziomów wyższych niż warstwa 802.3 (warstwa łącza danych) wchodzą do pakietów eternetowych jako dane.

Jeśli miejsce przeznaczenia znajduje się w tej samej sieci co miejsce nadania, to system może go poszukać w pamięci podręcznej ARP i po znalezieniu eternetowego adresu komputera może przesłać pakiet siecią lokalną. Docelowe urządzenie Ethernet zauważa wówczas swój adres w pakiecie, czyta ten pakiet i przekazuje w gorę stosu protokołów.

Jeśli system docelowy jest w innej sieci niż źródło komunikatu, to system źródłowy odnajduje odpowiedni ruter w swojej sieci i posyla pakiet do tego routera. Rutery przekazują pakiet przez sieć rozległą, aż dotrze on do sieci docelowej. Ruter, który jest połączony z siecią docelową, odszukuje w swojej podręcznej pamięci ARP numer eternetowy miejsca przeznaczenia i wysyła pakiet do właściwego komputera. Podczas całego tego łańcucha przesyłan wraz z użyciem adresu sieciowego kolejnego routera może się zmieniać nagłówek warstwy łącza danych. Inne nagłówki pakietu pozostają natomiast takie same do chwili jego przyjęcia, kiedy to następuje ich przetworzenie przez stos protokołów i wreszcie przekazanie przez jądro do procesu odbiorczego.

15.8 ■ Podsumowanie

System rozproszony jest zbiorem procesorów, które nie dzielą pamięci ani zegara. Każdy procesor ma natomiast własną lokalną pamięć, a procesory komunikują się ze sobą za pośrednictwem różnych linii komunikacyjnych, takich jak szybkie szyny lub linie telefoniczne. Procesory w systemie rozproszonym różnią się rozmiarami i funkcjami. Mogą znajdować się wśród nich małe mikroprocesory, stacje robocze, minikomputery i wielkie, uniwersalne systemy komputerowe.

Procesory w systemie są połączone za pomocą sieci komunikacyjnej, która może być skonfigurowana na kilka różnych sposobów. Sieć może być połączona całkowicie lub częściowo. Może ona mieć kształt drzewa, gwiazdy, pierścienia lub przyjmować postać szyny wielodostępnej. W projekcie sieci komunikacyjnej należy uwzględnić wybieranie tras oraz strategie połączeń, należy też wziąć pod uwagę problemy rywalizacji i bezpieczeństwa.

Zasadniczo istnieją dwa typy systemów rozproszonych: sieci lokalne (LAN) oraz sieci rozległe (WAN)¹. Podstawową różnicą między nimi jest ich rozmieszczenie geograficzne. Sieci lokalne są złożone z procesorów rozmieszczonych na niewielkim terenie, jak pojedynczy budynek lub kilka sąsiednich budynków. Sieci WAN są zbudowane z autonomicznych procesorów rozproszonych na wielkim obszarze geograficznym (np. takim jak USA).

Stosy protokołów, takie jak określone w modelach warstw sieci, obejmują komunikat, zaopatrując go w informacje zapewniające, że dotrze on do miejsca przeznaczenia. Do tłumaczenia nazw komputerów na ich adresy sieciowe trzeba stosować system nazewnictwa, taki jak DNS, a do tłumaczenia numeru sieci na adres urządzenia sieciowego (np. adres standardu Ethernet) może się okazać przydatny inny protokół (taki jak ARP). Jeśli systemy są w osobnych sieciach, to do przekazywania pakietów z sieci źródłowej do sieci docelowej stosuje się rutery.

■ Ćwiczenia

- 15.1 Porównaj różne topologie sieci pod względem niezawodności.
- 15.2 Dlaczego w większości sieci rozległych stosuje się tylko topologię połączeń częściowych?
- 15.3 Jakie są główne różnice między sieciami rozległymi a lokalnymi?

¹ Istnieją też inne klasyfikacje typów systemów rozproszonych. – Przyp. tłum.

- 15.4** Jaka konfiguracja sieci najlepiej pasowałaby do następujących środowisk?
- (a) piętro w domu akademickim;
 - (b) ośrodek uniwersytecki;
 - (c) województwo;
 - (d) państwo.
- 15.5** Chociaż w modelu ISO określono siedem warstw funkcjonalnych, w większości systemów komputerowych stosuje się mniej warstw do implementowania sieci. Co jest powodem takiego postępowania? Jakie trudności może powodować użycie mniejszej liczby warstw?
- 15.6** Wyjaśnij, dlaczego podwojenie szybkości systemów podłączonych do segmentu sieci Ethernet może spowodować pogorszenie wydajności sieci. Jakie zmiany można wprowadzić, aby rozwiązać ten problem?
- 15.7** W jakich warunkach sieć pierścieniowa z żetonem jest efektywniejsza od sieci Ethernet?
- 15.8** Dlaczego pomyślu, aby bramy przekazywały do innych sieci pakiety rozmawiane, nie można uznać za dobry? Jaka byłaby tego zaleta?
- 15.9** Pod jakimi względami użycie serwera nazw jest lepsze niż użycie tablic statycznych w komputerach sieciowych? Jakie trudności i komplikacje dotyczą serwerów nazw? Jaki metody można by zastosować, aby zmniejszyć ruch w sieci wytwarzany przez serwery nazw obsługujące zamówienia związane z tłumaczeniami?
- 15.10** Oryginalny protokoł HTTP^{*} korzysta z protokołów TCP/IP jako wewnętrznych protokołów sieci. Przy każdym kontakcie ze stroną, obrazkiem lub aplitem jest organizowana oddzielnna sesja TCP, korzysta się z nawiązanej łączności, po czym ją przerysuje. Wskutek kosztów wynikających ze stawienia i likwidowania połączeń według protokolu TCP/IP w metodzie tej wystąpiły kłopoty z wydajnością. Czy zastosowanie protokołu UDP zamiast TCP byłoby dobrą propozycją zastępczą? Jakie inne zmiany można by poczynić w celu poprawienia wydajności protokołu HTTP?
- 15.11** Do czego przydaje się protokoł tłumaczenia adresów? Dlaczego posługiwanie się takim protokołem jest lepsze niż powodowanie, aby każdy

* Protokoł przesyłania hiperekstu (ang. *HyperText Transfer Protocol*). Przyp. tłum.

komputer czytał każdy pakiet w celu określenia, dla kogo jest on przeznaczony? Czy w sieci pierścieniowej z zetonem protokół taki jest potrzebny?

Uwagi bibliograficzne

Książki Tanenbauma [414] i Halsalla [163] zawierają ogólny przegląd sieci komputerowych. Sprzęt i oprogramowanie sieciowe szczegółowo przedstawia Fortier w książce [139].

Sieć Internet i kilka innych zostały omówione przez Quartermana i Hoskinsa w artykule [340]. Sieć Internet oraz jej protokoły opisano w książkach Comera [81] oraz Comera i Stevensa [82], [83]. Programowanie sieciowe w systemie UNIX zostało wszechstronnie opisane przez Stevensa w książce [407]. Ogólny przegląd sieci podał Quarterman [338].

Feng w artykule [133] dokonał przeglądu różnych topologii sieciowych. Boorstyn i Frank [46] oraz Gerla i Kleinrock [149] omówili problemy projektowania topologii sieci. Day i Zimmerman w materiałach [94] przedstawili model OSI.

Specjalne wydanie miesięcznika *Computer Networks*, z grudnia 1979, zawiera dziewięć artykułów poświęconych sieciom LAN, obejmujących takie tematy, jak: sprzęt, oprogramowanie, symulacja i przykłady. Taksonomię oraz duży wykaz sieci LAN zaprezentowali Thurber i Freeman [427]. Stallings w artykule [400] omówił różne odmiany pierścieniowych sieci lokalnych.

Organizowanie niezawodnej komunikacji pomimo występowania awarii omówili Birman i Joseph [38]. Zaopatrywanie wielkiego systemu rozproszonego w środki bezpieczeństwa przedstawił Satyanarayanan w artykule [374].

the first time in the history of the world, the people of the United States have been called upon to determine whether they will submit to the law of force, or the law of the Constitution.

THE BATTLE OF BIRDS

It is a remarkable fact that the most important battle ever fought in the history of the world was fought by birds.

The battle took place in the year 1865, between the United States and the Confederacy.

The United States had a large army, and the Confederacy had a smaller army.

The United States army was led by General Grant, and the Confederacy army was led by General Lee.

The battle was fought at the Battle of Birds, which is located in the state of Virginia.

The battle lasted for several hours, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

The battle was fought by birds, and the result was a victory for the United States.

Rozdział 16

STRUKTURY SYSTEMÓW ROZPROSZONYCH

W tym rozdziale omawiamy ogólną strukturę systemów rozproszonych. Pokazujemy główne różnice w projektach systemów operacyjnych występujące między tym rodzajem systemów a systemami scentralizowanymi, którymi zajmowaliśmy się poprzednio.

16.1 ■ Sieciowe systemy operacyjne

Sieciowy system operacyjny (ang. *network operating system*) tworzy środowisko, w którym użytkownicy – świadomi wielkości maszyn – mają dostęp do zdalnych zasobów, rejestrując się na odpowiednich zdalnych maszynach lub przesyłając do własnych maszyn dane z maszyn zdalnych.

16.1.1 Zdalna rejestracja

Istotne zadanie sieciowego systemu operacyjnego polega na umożliwianiu użytkownikom zdalnego rejestrowania się na innych komputerach. Sieć Internet oferuje do tego celu usługę *telnet*. Aby zilustrować to udogodnienie, założmy, że użytkownik w Brown University chce wykonać obliczenia na komputerze „cs.utexas.edu” zlokalizowanym w University of Texas. W tym celu użytkownik musi mieć na tamtej maszynie ważne konto. Aby zarejestrować się na odległość, użytkownik wydaje polecenie

telnet cs.utexas.edu

Polecenie to powoduje utworzenie połączenia między lokalną maszyną w Brown University a komputerem „cs.utexas.edu”. Po nawiązaniu połącze-

nia oprogramowanie sieciowe tworzy przezroczyste, dwukierunkowe łącze, dzięki któremu wszystkie znaki wprowadzane przez użytkownika są wysyłane do procesu w komputerze „cs.utexas.edu”, a wszystkie dane wyjściowe tego procesu są posyłane z powrotem do użytkownika. Proces w odległym komputerze prosi użytkownika o podanie nazwy rejestracyjnej i hasła. Po otrzymaniu poprawnych informacji, proces ten działa jako przedstawiciel użytkownika, który może na zdalnej maszynie wykonywać obliczenia zupełnie tak samo, jak to robi każdy użytkownik lokalny.

16.1.2 Przesyłanie odległych plików

Inną ważną funkcją sieciowego systemu operacyjnego jest dostarczenie mechanizmu przesyłania plików między jedną maszyną a drugą. W takim środowisku każdy komputer utrzymuje własny, lokalny system plików. Jeśli użytkownik w jakimś miejscu (powiedzmy „cs.brown.edu”) chce uzyskać dostęp do pliku znajdującego się w innym komputerze (powiedzmy „cs.utexas.edu”), to plik musi być jawnie przekopiowany z komputera w Teksasie do komputera w Brown.

W sieci Internet mechanizmem umożliwiającym tego rodzaju przesyłanie jest program **FTP** (ang. *file transfer protocol*). Przypuśćmy, że użytkownik komputera „cs.brown.edu” chce skopiować plik *publikacja.tex*, przechowywany w komputerze „cs.utexas.edu”, do lokalnego pliku *moja-publikacja.tex*. Użytkownik musi najpierw wywołać program FTP, pisząc

```
ftp cs.utexas.edu
```

Uruchomiony program poprosi wtedy użytkownika o podanie nazwy rejestracyjnej i hasła. Po otrzymaniu poprawnych informacji użytkownik musi przejść do podkatalogu, w którym jest plik *publikacja.tex*, po czym skopiować ten plik za pomocą polecenia

```
get publikacja.tex moja-publikacja.tex
```

W tym schemacie położenie pliku nie jest przezroczyste dla użytkownika, który musi dokładnie wiedzieć, gdzie dany plik się znajduje. Co więcej, nie ma tu rzeczywistego dzielenia plików, ponieważ użytkownik może tylko skopiować plik z jednego stanowiska na inne. Może więc istnieć wiele kopii tego samego pliku, co powoduje marnowanie miejsca. Ponadto w przypadku wykonywania zmian w tych kopiach staną się one niespójne.

Zauważmy, że w naszym przykładzie użytkownik z Brown University, musi mieć pozwolenie rejestrowania się na stanowisku „cs.utexas.edu”. Program FTP oferuje także możliwość zdalnego kopирования plików przez użytkowników, którzy nie mają kont na komputerze w Teksasie. Takie zdalne kopowanie odbywa się metodą tzw. „anonimowego przesyłania plików” (ang. *ano-*

ymous ftp), która działa następująco. Plik, który ma być przekopiowany (tzn. *publikacja.tex*), musi być umieszczony w specjalnym podkatalogu (powiedzmy – *ftp*) z zezwoleniem na czytanie przez ogół. Użytkownik chcący skopiować ten plik wydaje polecenie *ftp*, tak jak poprzednio. Gdy użytkownik zostanie poproszony o podanie nazwy rejestracyjnej, poda on nazwę „anonymous” i dowolne hasło.

Jeśli system dopuszcza rejestracje w formie anonimowej, to musi zadbać, aby użytkownik z częściowymi uprawnieniami nie uzyskał dostępu do nie właściwych plików. Na ogół użytkownikowi pozwala się na dostęp tylko do tych plików, które są wpisane w drzewie katalogowym użytkownika „anonymous”. Wszystkie pliki umieszczone w tym drzewie są dostępne dla każdego anonimowego użytkownika zgodnie z obowiązującymi w danej maszynie regulami ochrony plików. Użyciu użytkownikom anonimowym nie pozwala się natomiast na dostęp do plików występujących poza tym drzewem katalogowym.

Mechanizm FTP jest zrealizowany w sposób podobny do usługi *telnet*. Na zdalnym komputerze działa demon obserwujący zamówienia nadchodzące do systemowego portu FTP. Następuje uwierzytelnienie rejestrującego się użytkownika, po czym pozwala mu się na zdalne wykonywanie poleceń. W odróżnieniu od demona usługi *telnet*, który wykonuje dla użytkownika dowolne polecenie, demon FTP reaguje tylko na z góry zdefiniowany zbiór poleceń dotyczących plików. Należą do nich operacje:

- **get** – przesyłanie pliku z maszyny zdalnej do maszyny lokalnej;
- **put** – przesyłanie pliku z maszyny lokalnej do maszyny zdalnej;
- **ls lub dir** – wyprowadzenie wykazu plików w bieżącym katalogu maszyny zdalnej;
- **cd** – zmiana bieżącego katalogu na zdalnej maszynie.

Istnieją także różne polecenia służące do zmiany trybu przesyłania (dla plików binarnych lub ASCII) oraz do określania stanu połączenia.

Istotną cechą usług *telnet* i FTP jest to, że wymagają one od użytkownika zmiany wzorca zachowania. Program FTP wymaga od użytkownika znajomości poleceń zupełnie różnych od zwykłych poleceń systemu operacyjnego. Usługa *telnet* wymaga nieco mniejszego przedstawienia – użytkownik musi znać odpowiednie polecenia odległego systemu. Jeśli na przykład użytkownik maszyny z systemem UNIX rejestruje się zdalnie na maszynie z systemem VMS, to na czas trwania zdalnej sesji musi się przerzucić na polecenia systemu VMS. Wygodniejsze dla użytkowników są takie rozwiązania, które nie wymagają posługiwania się różnymi zbiorami poleceń. Celem projektowania rozproszonych systemów operacyjnych jest łagodzenie tego problemu.

16.2 ■ Rozproszone systemy operacyjne

W rozproszonym systemie operacyjnym użytkownicy uzyskują dostęp do zasobów zdalnych w taki sam sposób jak do zasobów lokalnych. Wędrowka danych i procesów z jednego stanowiska do innego odbywa się pod nadzorem systemu operacyjnego.

16.2.1 Wędrowka danych

Załóżmy, że użytkownik na stanowisku A chce mieć dostęp do danych (np. do pliku) znajdujących się na stanowisku B. Są dwie podstawowe metody, które system może zastosować w celu przesłania danych. Jedna z nich polega na przesłaniu całego pliku na stanowisko A; po tej operacji wszystkie dostępy do pliku są lokalne. Gdy użytkownik rezygnuje już z dostępu do danego pliku, kopię pliku (jeśli został zmieniony) posyła się z powrotem na stanowisko B. Oczywiście, nawet jeśli dokonano drobnych zmian w wielkim pliku, to trzeba będzie przesyłać wszystkie dane. Na mechanizm taki można spoglądać jak na zautomatyzowany system FTP. Metodę tę zastosowano w systemie plików Andrew, co omówimy w rozdz. 17, jednak okazała się ona zbyt niewydajna.

Druga metoda polega na przesłaniu na stanowisko A tylko tej porcji pliku, która jest *niezbędna* do natychmiastowego działania. Jeśli później będzie potrzebna inna porcja pliku, to nastąpi kolejne przesłanie. Gdy użytkownik nie chce już korzystać z pliku, każda część pliku, która została zmieniona, musi być z powrotem wysłana do stanowiska B. (Zauważmy podobieństwo do stronicowania na żądanie). Metodę tę zastosowano w protokole NFS (Network File System) firmy Sun Microsystems (zob. rozdz. 17); użyto jej także w nowszych wersjach systemu Andrew. Protokół SMB firmy Microsoft (działający powyżej protokołów TCP/IP lub protokołu Microsoft NETBUI) umożliwia również dzielenie pliku przez sieć. Więcej informacji na temat protokołu SMB znajduje się w p. 23.6.1.

Jest więc oczywiste, że jeśli dostęp jest potrzebny tylko do małej części wielkiego pliku, to lepiej posłużyć się drugą metodą. Jeśli zaś chcemy mieć dostęp do sporej części pliku, to bardziej opłacalne jest skopiowanie całego pliku.

Zauważmy, że nie wystarcza tylko skopiować dane z jednego stanowiska na drugie. Jeśli oba stanowiska nie są całkowicie zgodne (np. jeśli używa się w nich różnych reprezentacji kodów znaków lub liczb całkowitych o różnej liczbie lub porządku bitów), to system musi także wykonać różnorodne tłumaczenia danych.

16.2.2 Wędrówka obliczeń

W pewnych warunkach może się okazać wydajniejsze przesłanie w systemie obliczeń zamiast danych. Rozważmy na przykład zadanie polegające na wykonaniu pewnego zestawienia informacji zawartych w różnych wielkich plikach rezydujących na innym stanowisku. Wydajniej byłby skorzystać z tych plików w miejscu, w którym są one przechowywane, a następnie przekazać wyniki do stanowiska, na którym rozpoczęto obliczenia. Ogólnie biorąc, można powiedzieć, że jeżeli czas przesłania danych jest dłuższy niż czas wykonania polecenia zdalnego, to powinno się użyć polecenia zdalnego.

Takie obliczenia można wykonać na kilka różnych sposobów. Założymy, że proces P powinien mieć dostęp do pliku na stanowisku A. Dostęp do pliku realizuje się na stanowisku A, ale można go zainicjować za pomocą *wywołania procedury zdalnej* (ang. *remote procedure call* – RPC). W realizacji wywołania RPC używa się protokołu datagramowego (w sieci Internet jest to protokół UDP) w celu wykonania podprogramu w zdalnym systemie (zob. p. 16.3.1). Proces P wywołuje zdefiniowaną wcześniej procedurę na stanowisku A. Procedura ta, po odpowiednim wykonaniu, przekazuje procesowi P wyniki.

Inny sposób polega na wysłaniu przez proces P komunikatu do stanowiska A. System operacyjny na stanowisku A tworzy wtedy nowy proces Q, który ma wykonać wyznaczone zadanie i po zakończeniu działania wysłać oczekiwane wyniki do procesu P za pomocą systemu komunikatów. Zauważmy, że w tym układzie proces P może działać współbieżnie z procesem Q, a w rzeczywistości można w ten sposób mieć wiele procesów wykonywanych współbieżnie na wielu stanowiskach.

Obu schematów dałoby się używać w celu dostępu do wielu plików rezydujących na różnych stanowiskach. Wywołanie jednej zdalnej procedury mogłoby spowodować rozpoczęcie innej zdalnej procedury albo też przekazanie komunikatu do innego stanowiska. Podobnie proces Q mógłby podczas swojego działania wysłać do innego stanowiska komunikat, który spowodowałby utworzenie innego procesu. Nowy proces wysłałby komunikat z powrotem do Q albo powtórzyłby cykl tworzenia kolejnych procesów.

16.2.3 Wędrówka procesów

Logicznym rozszerzeniem wędrówki obliczeń jest wędrówka procesów. Przedłożony do wykonania proces nie zawsze jest wykonywany na stanowisku, na którym go rozpoczęto. Może okazać się korzystne wykonanie całego procesu lub jego części na innych stanowiskach. Schemat ten znajduje następujące uzasadnienia:

- **Równoważenie obciążzeń:** Procesy (lub podprocesy) można rozmieszczać w sieci w celu wyrównywania obciążen poszczególnych stanowisk.
- **Przyspieszanie obliczeń:** Jeśli pojedynczy proces daje się podzielić na pewną liczbę podprocesów, to mogą one być wykonane równolegle na różnych stanowiskach, dzięki czemu łączny czas wykonania całego procesu ulega skróceniu.
- **Preferencje sprzętowe:** Proces może odznaczać się pewnymi cechami, które predysponują go do wykonania na wyspecjalizowanym procesorze (jest tak np. w przypadku odwracania macierzy, które łatwiej wykonać na procesorze macierzowym niż na mikroprocesorze).
- **Preferencje programowe:** Do wykonania procesu może być potrzebne oprogramowanie, które jest osiągalne tylko na określonym stanowisku, z którego nie wolno go przenosić lub lepiej opłaca się przenieść do niego proces.
- **Dostęp do danych:** Tak jak w przypadku wędrowki obliczeń, gdy w obliczeniach używa się wielu danych, zdalne wykonanie procesu może okazać się bardziej opłacalne niż przenoszenie wszystkich danych do mierzystego komputera procesu.

Zasadniczo istnieją dwie uzupełniające się techniki, które można zastosować do przenoszenia procesów w sieciach komputerowych. W pierwszej z nich system może próbować ukrywać fakt, że proces „wyemigrował” od klienta. Zaletą tego schematu jest to, że użytkownik nie musi w swoim programie jawnie kodować przemieszczania procesu. Metody tej używa się zwykle do równoważenia obciążen i przyspieszania obliczeń w systemach jednorodnych, gdyż nie wymagają one od użytkownika żadnych wskazówek co do zdalnego wykonywania programów.

Druga metoda polega na umożliwieniu (lub wymaganiu), by użytkownik jawnie określił, jak proces ma być przemieszczony. Ta metoda jest zwykle przydatna, gdy proces musi zostać przemieszczony ze względu na preferencje sprzętowe lub programowe.

16.3 ■ Usługi zdalne

Rozważmy sytuację użytkownika, który potrzebuje dostępu do danych umieszczonych na innym stanowisku. Powiedzmy, że użytkownik chce policzyć całkowitą liczbę wierszy, słów i znaków w pliku przechowywanym na drugim stanowisku. Zamówienie na wykonanie tej pracy będzie obsłużone

przez zdalny serwer na stanowisku A, który uzyska dostęp do pliku, obliczy pożądane wyniki, aby na koniec przesłać je do użytkownika.

Jednym ze sposobów organizacji tego przesyłania jest metoda *usługi zdalnej* (ang. *remote service*). Zamówienia na dostęp są dostarczane do serwera. Kontakt z danymi następuje w maszynie serwera, a otrzymane wskutek tych działań wyniki przekazuje się z powrotem do użytkownika. Między dostępami a ruchem na drodze do i od serwera istnieje bezpośrednia odpowiedniość. Zamówienia dostępu są tłumaczone na komunikaty dla serwerów, a odpowiedzi serwerów są pakowane do komunikatów i odsyłane z powrotem do użytkowników. Każdy dostęp jest obsługiwany przez serwer i powoduje ruch w sieci. Czytanie powoduje na przykład wysłanie do serwera zamówienia czytania oraz wysłanie do użytkownika odpowiedzi zawierającej potrzebne dane.

16.3.1 Zdalne wywołania procedur

Jedną z najpopularniejszych form usługi zdalnej jest wzorzec postępowania określany jako RPC, czyli zdalne wywołanie procedury^{*}, które omówiliśmy pokrótko w p. 4.6.3. Wywołanie RPC zaprojektowano jako sposób uogólnienia mechanizmu wywołania procedury na użytek systemów połączonych siecią. Jest ono pod wieloma względami podobne do mechanizmu komunikacji międzyprocesowej (IPC) – opisanego w p. 4.6 – i zazwyczaj jest budowane powyżej takiego systemu. Ponieważ zajmujemy się środowiskiem, w którym procesy są wykonywane w oddzielnych systemach, więc usługi zdalne muszą być udostępniane na zasadzie wymiany komunikatów. W przeciwnieństwie do komunikacji międzyprocesowej komunikaty wymieniane w trybie RPC mają scisłe określoną budowę i nie są już tylko zwykłymi pakietami danych. Są one adresowane do demona RPC prowadzącego na słuch portu RPC w odległym systemie i zawierają identyfikator funkcji do wykonania oraz parametry, które należy jej przekazać. Funkcja taka zostanie następnie wykonana zgodnie z życzeniem, a wszystkie wyniki będą odeslane do zamawiającego w oddzielnym komunikacie.

Port jest po prostu numerem umieszczonym na początku pakietu z komunikatem. Chociaż system ma zazwyczaj tylko jeden adres sieciowy, może pod tym adresem udostępniać wiele portów, aby rozróżnić swoje liczne usługi sieciowe. Jeśli proces zdalny potrzebuje obsługi, to adresuje komunikat do właściwego portu. Jeżeli na przykład jakiś system miał umożliwiać innym systemom wyprowadzanie bieżącego wykazu jego użytkowników, to mógłby

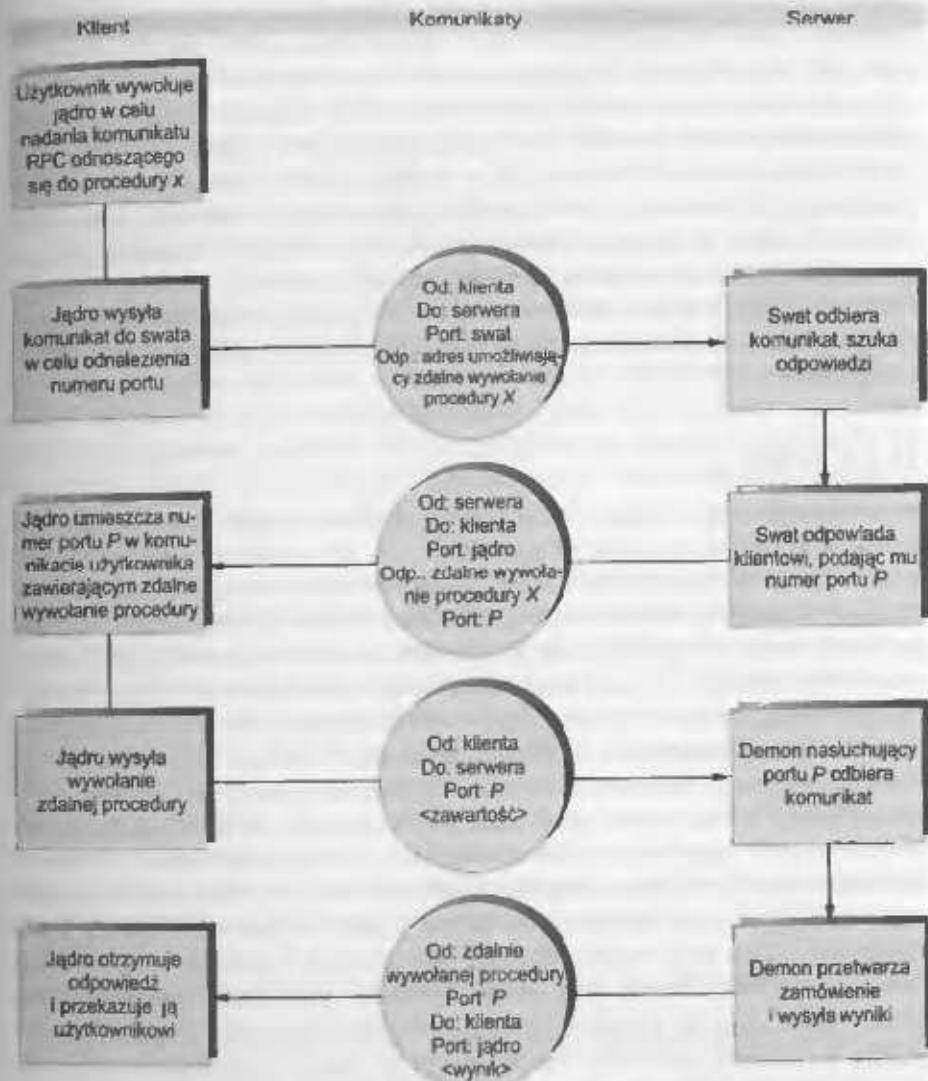
* Oba terminy: „zdalne wywołanie procedury” i „wywołanie procedury zdalnej” są równoważne. – Przyp. tłum.

działać w nim demon przypisany do portu, powiedzmy – portu 3027, obsługujący tego rodzaju wywołania RPC. Dowolny system zdalny mógłby uzyskać potrzebne informacje (tj. spis bieżących użytkowników), wysyłając komunikat RPC do portu 3027 w danym serwerze; wyniki otrzymałby w komunikacie wysłanym w odpowiedzi.

Mechanizm RPC jest powszechnie stosowany w systemach sieciowych, toteż warto omówić kilka związań z nim zagadnień. Jedną z istotnych spraw jest semantyka wywołania. Procedura lokalna ulega awarii jedynie w wyjątkowych sytuacjach, natomiast wywołania RPC mogą zawodzić lub być podwajane i wykonywane więcej niż jednokrotnie z powodu typowych niedomagań sieci. Ponieważ mamy do czynienia z przesyaniem komunikatów przez zawodne łącza komunikacyjne, więc znacznie łatwiej jest zapewniać w systemie operacyjnym co najwyżej jednokrotne wystąpienie danego komunikatu niż gwarantować, że komunikat pojawi się dokładnie jeden raz. Z uwagi na to, że procedury lokalne działają zgodnie z drugim założeniem, w wielu systemach dochodzi do prób podwajania ich działań. W systemach takich każdy komunikat zostaje zaopatrzony w znacznik czasu. Serwer musi utrzymywać rejestr znaczników czasu wszystkich obsłużonych komunikatów lub przynajmniej tylu, aby można było wykrywać ich powtórzenia. Komunikaty, których znaczniki czasu już występują w takim rejestrze, będą po nadaniu pomijane. Wytwarzanie znaczników czasu omawiamy w p. 18.1.

Inne ważne zagadnienie dotyczy komunikacji między serwerem a klientem. W standardowych wywołaniach procedur dochodzi do swoistych wiązań podczas konsolidacji, ładowania lub wykonywania (zob. rozdz. 8), wskutek których nazwa wywołania procedury jest zastępowana przez adres wywołania danej procedury w pamięci. Schemat wywołania procedury zdalnej (RPC) wymaga podobnego powiązania klienta i portu serwera, skąd jednak klient ma znać numery portów serwera? Żaden system nie ma pełnej informacji o drugim, gdyż nie dzielą one pamięci. W powszechnym użyciu są dwa podejścia. W pierwszym z nich informacje wiążące mogą być określone z góry w postaci ustalonych adresów portów. Wywołaniu RPC można wówczas przydzielić stały numer portu podczas komplikacji. Po skompilowaniu programu serwer nie może zmienić numeru portu zamawianej usługi. W drugim podejściu wiązanie jest wykonywane dynamicznie, za pomocą mechanizmu *rendez-vous*. Zazwyczaj system operacyjny dysponuje demonem spotkań (zwany również swatem*) w stałym porcie RPC. Klient wysyła wtedy komunikat do demona spotkań, prosząc o adres portu RPC, z którym chce pracować. W odpowiedzi otrzymuje numer portu, do którego może dopóty przesyłać wywołania proce-

* Z ang. *matchmaker*; w użyciu jest również termin „łącznik” (ang. *binder*). – Przyp. tłum.



Rys. 16.1 Wykonanie zdalnego wywołania procedury (RPC)

dur zdalnych, dopóki nie zakończy działania (lub nie zepsuje się serwer). Ta metoda wymaga dodatkowych zabiegów na początku zamówienia, lecz jest bardziej elastyczna niż pierwszy sposób. Przykład interakcji jest pokazany na rys. 16.1.

Schemat zdalnych wywołań procedur przydaje się do realizowania rozproszonego systemu plików (rozdz. 17). System taki można zaimplemento-

wać w postaci zbioru demonów RPC i klientów. Komunikaty są adresowane do portu rozproszonego systemu plików w serwerze, w którym ma być wykonana operacja plikowa. Komunikat zawiera operację dyskową, którą należy wykonać. Do zbioru operacji dyskowych może należeć czytanie, pisanie, przemianowywanie, usuwanie i badanie stanu – odpowiednio do zwyczajnych wywołań systemowych związanych z obsługą plików. Komunikat powrotny zawiera wszelkie dane przekazywane przez wywołanie, którego wykonanie bierze na siebie w imieniu klienta demon rozproszonego systemu plików. Komunikat może na przykład zawierać żądanie przesłania do klienta całego pliku, choć może to być również zamówienie pojedynczego bloku. W ostatnim przypadku do przesłania całego pliku może być potrzebnych kilka takich zamówień.

16.3.2 Wątki

W systemach rozproszonych wątki znajdują również częste zastosowania jak zdalne wywołania procedur. Wątki nadają się do wysyłania i przyjmowania komunikatów w sposób umożliwiający asynchroniczne wykonywanie innych operacji w zadaniu. Możliwe jest na przykład skonstruowanie demona, którego liczne wątki mogą czekać na komunikaty z zamówieniami i pobierać po jednym komunikacie w celu przetworzenia go wspólnie z innymi wątkami, wykonującymi te same czynności. Trwają badania nad znalezieniem sposobu odciążenia za pomocą wątków mechanizmu zdalnego wywołania procedury. W jednym z ulepszeń w stosunku do konwencjonalnych wywołań RPC skorzystano z faktu, że kontekst wątku serwera, który blokuje się w oczekiwaniu na nowe zamówienie, zawiera niewiele istotnych informacji. W metodzie nazywanej niekiedy *odbiorkiem niejawnym* (ang. *implicit receive*) wątek znika po zakończeniu zadania. Aby obsłużyć nadchodzące zamówienie, jądro tworzy wówczas nowy wątek. Wpisuje ono również komunikat w przestrzeń adresową serwera i tworzy stos, aby nowy wątek miał dostęp do komunikatu. Wątek utworzony na zasadzie „według potrzeb” jako odpowiedź na nowe zdalne wywołanie procedury można nazywać wątkiem *wyskakującym* (ang. *pop-up thread*). Taki sposób działania umożliwia poprawienie wydajności, ponieważ rozpoczęcie nowego wątku jest tańsze niż regenerowanie wątku istniejącego. Zważywszy że zaden wątek nie blokuje się z powodu oczekiwania na pracę, nie trzeba przechowywać ani odtwarzać żadnego kontekstu. Dodatkowy czas oszczędza się na tym, że nadchodzących wywołań RPC nie trzeba kopiować do bufora w wątku serwera.

Udowodniono, że znaczna liczba zdalnych wywołań procedur w systemach rozproszonych odnosi się do procesów na tej samej maszynie co proces wywołujący. Zdalne wywołania procedur można odciążyć dzięki zastosowa-

niu, w odniesieniu do wątków różnych procesów wykonywanych na tej samej maszynie, bardzo wydajnej komunikacji za pośrednictwem pamięci dzielonej. Na początku wątek serwera – *WS* – udostępnia swój interfejs jądra. W interfejsie są zdefiniowane procedury, które mogą być wywoływanie, wraz z ich parametrami i innymi niezbędnymi charakterystykami. Wątek klienta *WK*, który importuje ten interfejs, otrzyma jednoznaczny identyfikator, którym będzie się posługiwać później przy wykonywaniu wywołań. Aby spowodować wywołanie wątku *WS*, wątek *WK* odkłada argumenty na *stosie argumentów*, który jest strukturą danych użytkowaną wspólnie przez wątki *WS* i *WK*. W ramach tego wywołania wątek *WK* umieszcza w rejestrze jednoznaczny identyfikator. Po tym właśnie jądro rozpoznaje, że wywołanie nie jest zdalne, lecz lokalne. Jądro umieszcza wówczas klienta w przestrzeni adresowej serwera i rozpoczyna wykonywanie procedury wątku *WS* na rzecz wątku *WK*. Ponieważ argumenty znajdują się już na właściwym miejscu i nie trzeba ich kopować, lokalne wywołania RPC wykonywane w przedstawiony sposób są wydajniejsze pod względem czasowym od konwencjonalnych wywołań procedur zdalnych.

Zilustrujemy zastosowanie wątków w środowisku rozproszonym, analizując konkretny pakiet wątków, udostępniony przez konsorcjum Open Software Foundation w rozproszonym środowisku obliczeniowym DCE (ang. *Distributed Computing Environment*). Środowisko DCE ma duże znaczenie, gdyż jest ono plonem porozumienia między niemal wszystkimi głównymi dostawcami systemu UNIX co do wyposażenia ich implementacji unikowych w działania i protokoły sieciowe potraktowane jako element standardizujący, umożliwiający lepszą współpracę między systemami. Środowisko to będzie też dostępne w systemie Windows NT firmy Microsoft. Pakiet DCE zawiera dla wygody użytkownika wiele różnych wywołań. W tym rozdziale rozważymy tylko najistotniejsze spośród nich, grupując je w pięć klas:

1. Wywołania zarządzania wątkami: **create, exit, join, detach**.
2. Wywołania synchronizujące: **mutex_init, mutex_destroy, mutex_try_lock, mutex_lock, mutex_unlock**.
3. Wywołania obsługujące zmienne warunkowe: **cond_init, cond_destroy, cond_wait, cond_signal, cond_broadcast**.
4. Wywołania dotyczące planowania przydziału procesora: **setscheduler, getscheduler, setprio, getprio**.
5. Wywołania kończenia wątków: **cancel, setcancel**.

Kazde z tych wywołań omówimy teraz pokróćce.

Cztery wywołania zarządzania wątkami służą do tworzenia nowych wątków oraz umożliwiają ich kończenie po wykonaniu zamówienia. Wywołanie **join**, przypominające uniksowe wywołanie systemowe **wait**, umożliwia wątkowi macierzystemu czekanie na wątek potomny. Jeśli wątek macierzysty nie musi czekać na potomka, to może o tym oświadczyć za pomocą polecenia **detach**.

Dostęp do danych dzielonych można synchronizować za pomocą **zamka**, czyli odmiany semafora binarnego (zob. rozdz. 6). W środowisku DCE zamki mogą być tworzone (inicjowane) i usuwane dynamicznie. Zamek znajduje się w jednym z dwóch stanów: zablokowania albo odblokowania. Do działań na zamkach służą trzy operacje: **mutex_lock**, **mutex_trylock** oraz **mutex_unlock**. Próba zablokowania zamka kończy się sukcesem tylko w odniesieniu do zamka odblokowanego, z chwilą zablokowania zamka działanie na nim nabiera cechy niepodzielności. Zamek można odblokować za pomocą odpowiedniej operacji **mutex_unlock**. Jeśli kilka wątków czeka na zamek, to uwalniony zostanie tylko jeden z nich. Wywołanie **mutex_trylock** służy do próby zablokowania zamka. Jeśli dany zamek jest już zablokowany, to wywołanie to zwraca kod stanu wskazujący, że zablokowanie się nie powiodło, a wątek, który je spowodował nie jest blokowany. Jeśli zamek jest odblokowany, to wywołanie zwraca kod oznaczający swoje pomysłe wykonanie.

Środowisko DCE umożliwia również operacje na zmiennych warunkowych, stosowanych w większości pakietów wątków. Są one bardzo podobne do działań na zmiennych warunkowych w monitorach (zob. rozdz. 6). Zmienna warunkowa pozostaje w związku z zamkiem, jednak jej zastosowanie jest zupełnie inne. Aby to wyjaśnić, przyjmijmy, że wątek A zablokował zamek z_1 w celu uzyskania prawa wejścia do sekcji krytycznej. Po wykonaniu tej czynności wątek stwierdza, że nie może kontynuować pracy dopóty, dopóki inny wątek B nie zostanie wykonany i nie poczyta zmian danych w sekcji krytycznej. W tym celu wątek A mógłby zablokować drugi zamek – z_2 . Jednak użycie drugiego zamka mogłoby spowodować zakleszczenie, ponieważ wątkowi B mogłoby się wtedy nie udało wejść do sekcji krytycznej w celu zmiany danych, na którą czeka wątek A.

Zamiast blokowania zamka z_2 wątek A mógłby zablokować zmienną warunkową. Zablokowanie zmiennej warunkowej spowodowałoby automatyczne odblokowanie zamka z_1 , gdyby jego zablokowanie przez wątek A powodowało czekanie. Gdy wątek B wykona stosowne zmiany w danych, wówczas za pomocą odpowiedniego wywołania obudzi, czyli uaktywni wątek A.

Podobnie jak zamki, zmienne warunkowe można tworzyć i likwidować dynamicznie. Czekanie pod zmiennej warunkowej powoduje operację **cond_wait**. W środowisku DCE istnieją dwa rodzaje operacji budzenia. Operacja **cond_signal** wznowia działanie tylko jednego, czekającego wątku, natomiast

operacja `cond_broadcast` powoduje podjęcie działań przez wszystkie wątki czekające pod zmienną warunkową.

Inną klasą wywołań, które rozpatrujemy w tym rozdziale, są wywołania dotyczące planowania przydziału procesora. Zbiór wywołań planujących w pakiecie DCE umożliwia określanie i sprawdzanie algorytmów i priorytetów. Można dokonywać wyboru z pewnego zbioru algorytmów wywłaszczających i niewywłaszczających, zawierającego m.in. algorytm rotacyjny i FIFO.

Pakiet wątków zawiera wywołania służące do likwidowania wątków. Systemowe wywołanie `cancel` egzekwuje próbę likwidacji innego wątku. Wątek może się też posłużyć wywołaniem `setcancel`, aby zezwolić lub zabronić swojej ewentualnej likwidacji.

16.4 ■ Odporność

System rozproszony jest podatny na rozmaite rodzaje uszkodzeń. Awarie łączys, stanowisk, utraty komunikatów – to najczęściej spotykane niedomagania. Aby zapewnić odporność systemu*, należy wykrywać każde z tych uszkodzeń, zmieniać konfigurację systemu w sposób umożliwiający jego dalszą pracę oraz przywracać stanowiska do systemu po ich naprawie.

16.4.1 Wykrywanie uszkodzeń

W środowisku bez pamięci dzielonej z reguły nie można odróżnić uszkodzenia łącza od uszkodzenia stanowiska lub utraty komunikatu. Zazwyczaj potrafimy wykryć, że wystąpiła któraś z tych awarii, lecz możemy nie być w stanie określić jej rodzaju. Po wystąpieniu awarii muszą być podjęte odpowiednie działania uzależnione od konkretnej aplikacji.

Aby wykrywać uszkodzenia łączys i stanowiska, stosujemy procedurę *uzgadniania* (ang. *handshaking*). Założymy, że stanowiska A i B mają bezpośredni, fizyczne połaczenie. W stałych odstępach czasu oba stanowiska wysyłają sobie wzajemnie komunikat „jestem-czynne” (ang. *I-am-up*). Jeśli stanowisko A nie otrzyma tego komunikatu w określonym czasie, to może przyjąć, że stanowisko B uległo awarii, ze popsuło się łącze między A i B lub że komunikat z B został zagubiony. W tej sytuacji stanowisko A może postąpić dwójako. Może przeszukać przez kolejny okres na otrzymanie od B komunikatu „jestem-czynne” albo też może wysłać do B komunikat „czy-jesteś-czynne?” (ang. *are-you-up?*).

* Inny termin: *tolerowanie uszkodzeń* (ang. *fault tolerance*) lepiej odzwierciedla cel zarysowany w tym punkcie. – Przyp. tłum.

Jeśli stanowisko A nie otrzyma komunikatu „jestem-czynne” ani odpowiedzi na wyslane przez nie w tej sprawie zapytanie, to procedura może zostać powtórzona. Jedyny wniosek, jaki można bezpiecznie wyciągnąć na stanowisku A to ten, że wystąpiła jakaś awaria.

Stanowisko A może próbować odróżnić awarię łącza od awarii stanowiska przez wysłanie komunikatu „czy-jesteś-czynne?” do stanowiska B inną drogą (jeśli taka istnieje). Jeśli stanowisko B w ogóle otrzyma ten komunikat, to natychmiast odpowie pozytywnie. Odpowiedź pozytywna jest informacją dla stanowiska A, że stanowisko B jest czynne, a więc awaria dotyczy bezpośredniego połączenia, jakie między nimi występowało. Ponieważ nie wiadomo z góry, jak długo komunikat może podróżować od stanowiska A do stanowiska B i z powrotem, więc należy zastosować metodę *odliczania czasu* (ang. *timeout*). W chwili, gdy stanowisko A wysyła do B komunikat „czy-jesteś-czynne?”, określa ono odcinek czasu, przez który ma zamiar czekać na odpowiedź od B. Jeśli stanowisko A otrzyma odpowiedź w zadanym terminie, to może spokojnie uznać, że stanowisko B jest czynne. Jeżeli jednak nie otrzyma ono komunikatu z odpowiedzią w określonym czasie (tzn. wyznaczony czas minie), to w stanowisku można dojść tylko do wniosku, że wystąpiła jedna lub więcej z następujących sytuacji:

1. Stanowisko B nie działa.
2. Bezpośrednie łącze (jeśli takie istnieje) między A i B jest zepsute.
3. Alternatywna droga od A do B jest nieczynna.
4. Komunikat uległ zagubieniu.

Stanowisko A nie może jednakże rozstrzygnąć, które z tych zdarzeń wystąpiło naprawdę.

16.4.2 Rekonfigurowanie

Załóżmy, że stanowisko A wykryło za pomocą mechanizmu opisanego w poprzednim punkcie, że wystąpiła awaria. Musi ono wówczas rozpoczęć procedurę, która umożliwi rekonfigurację systemu i kontynuowanie jego pracy w normalnym trybie.

- Jeśli uszkodzeniu uległo bezpośrednie połączenie między stanowiskami A i B, to należy ogłosić to wszystkim stanowiskom w systemie, aby można było dokonać odpowiednich aktualnień rozmaitych tablic tras.
- Jeśli system przyjmuje, że awarii uległo stanowisko (ponieważ stało się ono nieosiągalne), to wszystkie stanowiska w systemie muszą być o tym

powiadomione, aby nie próbowali dalej korzystać z usług uszkodzonego stanowiska. Awaria stanowiska służącego jako centralny koordynator jakichś działań (np. wykrywanie zakleszczenia) wymaga obrania nowego koordynatora. Podobnie, jeśli uszkodzone stanowisko jest częścią pierścienia logicznego, to trzeba skonstruować nowy pierścień logiczny. Zauważmy, że jeżeli stanowisko nie uległo awarii (tzn. jeśli jest czynne, lecz niesiągalne), to może wystąpić niepożądana sytuacja, w której dwa stanowiska pełnią funkcję koordynatora. W przypadku podziału sieci ta dwójka koordynatorów (każdy w swojej części sieci) może doprowadzić do działań konfliktowych. Jeśli na przykład koordynatorzy odpowiadają za realizację wzajemnego wykluczania, to może dojść do sytuacji, w której dwa procesy mogą działać jednocześnie w swoich sekcjach krytycznych.

16.4.3 Usuwanie skutków awarii

Po naprawieniu uszkodzonego łącza lub stanowiska należy je zgrabnie i gładko zintegrować z systemem.

- Założymy, że uszkodzeniu uległo łącze między stanowiskami A i B. O jego zreperowaniu należy zawiadomić zarówno stanowisko A, jak i B. Można tego dokonać za pomocą stałego powtarzania procedury uzgadniania (warunków współpracy), opisanej w p. 16.4.1.
- Przypuśćmy, że awarii uległo stanowisko B. Kiedy powraca ono do pracy, wtedy musi zawiadomić wszystkie pozostałe stanowiska o swojej ponownej aktywności. Stanowisko B może wówczas otrzymać od innych stanowisk różne informacje, które mają mu posłużyć do uaktualnienia lokalnych tablic. Może ono na przykład potrzebować informacji do tablicy tras, wykazów stanowisk nieczynnych lub listy nie dostarczonych komunikatów i nie doręczonej poczty. Zauważmy, że nawet jeśli stanowisko nie uległo awarii, lecz po prostu nie można się z nim było skontaktować, to informacje tego rodzaju też są potrzebne.

16.5 ■ Zagadnienia projektowe

Wyzwaniem dla projektantów stało się uczyñenie wielości procesorów i urządzeń pamięci przezroczystymi dla użytkowników. W warunkach idealnych system rozproszony powinien sprawiać na użytkownikach wrażenie konwencjonalnego systemu scentralizowanego. W interfejsie użytkownika przezroczystego systemu rozproszonego zasoby lokalne i zdalne powinny być

nieodróżnialne. Należy przez to rozumieć, że dostęp do odległych systemów rozproszonych powinien być dla użytkowników taki sam, jak gdyby były one lokalne, a odnajdywanie zasobów i organizowanie właściwej z nimi współpracy powinno należeć do obowiązków systemu rozproszonego.

Innym aspektem przezroczystości jest mobilność użytkowników. Byłoby wygodnie, aby użytkownicy mogli się rejestrować na dowolnej maszynie i nie byli zmuszani do korzystania z konkretnych maszyn. Przecroczyty system rozproszony ułatwiają mobilność użytkowników przez zabieranie wraz z użytkownikiem jego środowiska (np. prywatnego katalogu) do miejsc, w których użytkownik się rejestruje. Zarówno system plików Andrew, rodem z Carnegie Mellon University (CMU), jak i projekt Athena z instytutu MIT umożliwiają to na wielką skalę. W mniejszym stopniu ten rodzaj przezroczystości zapewnia system NFS.

Pojęcie *tolerowania uszkodzeń* (ang. *fault tolerance*) jest przez nas używane w szerokim sensie. Awarie komunikacji, uszkodzenia maszyn (typu całkowite unieruchomienie) i urządzeń pamięci oraz zużycie nośników pamięci – wszystko to uważa się za uszkodzenia, które powinno się w pewnym stopniu tolerować. W wypadku ich występowania system tolerujący awarie powinien działać nadal, co najwyżej nieco gorzej. Pogorszenie może dotyczyć wydajności, funkcjonalności lub jednego i drugiego. Powinno ono jednak w pewnym sensie być proporcjonalne do awarii, które je powodują. System, który przestaje działać na skutek awarii niewielkiej liczby jego składowych z pewnością nie zasługuje na miano tolerującego awarie. Niestety, tolerowanie awarii jest trudne do osiągnięcia. Większość systemów handlowych toleruje uszkodzenia w ograniczonym zakresie. Na przykład w systemie VAXcluster firmy DEC umożliwia się komputerom wspólne używanie zbioru dysków. Jeśli jeden system ulegnie załamaniu, to użytkownicy mogą nadal korzystać ze swoich danych za pomocą drugiego systemu. Rzecz jasna, jeśli dojdzie do uszkodzenia dysku, to dostęp do niego utracą wszystkie systemy. Jednak na tę okoliczność można zastosować tablicę dysków RAID, aby zapewnić stały dostęp do danych nawet w przypadku awarii (zob. p. 13.5).

Zdolność systemu do dostosowywania się do wzrastającego natężenia usług nazywa się *skalowalnością* (ang. *scalability*). Systemy mają powynajmowane zasoby i mogą pęczać od nadmiernych obciążeń. Na przykład w odniesieniu do systemu plików nasycenie występuje wówczas, gdy procesor serwera jest intensywnie eksploatowany, lub wówczas, gdy dyski są bliskie zapełnienia. Skalowalność jest właściwością względową, ale można ją mierzyć dokładnie. System skalowalny powinien reagować harmonijniej na rosnące obciążenie niż system nieskalowalny. Po pierwsze, jego wydajność powinna maleć łagodniej niż w przypadku systemu nieskalowalnego. Po dru-

gie, w porównaniu z systemem nieskalowalnym jego zasoby powinny osiągać stan nasycenia później. Nawet doskonała konstrukcja nie może dostosować się do stale wzrastającego obciążenia. Dodawanie nowych zasobów może rozwiązywać ten problem, lecz może też powodować nowe, pośrednie obciążenia innych zasobów (np. dołączenie maszyn do systemu rozproszonego może zatyczać sieć i zwiększać popyt na usługi). Co gorsza, rozszerzanie systemu może powodować kosztowne zmiany w projekcie. System skalowalny powinien dać się rozszerzać bez tych trudności. Zdolność harmonijnego wzrostu jest szczególnie ważna w systemach rozproszonych, ponieważ na porządku dziennym dochodzi w nich do rozszerzania sieci przez dołączanie nowych maszyn lub łączenie dwu sieci ze sobą. Mówiąc krótko: konstrukcja skalowalna powinna przetrwać duże obciążenie usługami, dostosowywać się do wzrostu społeczności użytkowników i umożliwiać łatwą integrację dodawanych do niej zasobów.

Tolerowanie uszkodzeń oraz skalowalność są ze sobą powiązane. Mocno obciążona składowa może zostać sparaliżowana i zachowywać się tak, jakby uległa awarii. Z kolei przeniesienie obciążenia z uszkodzonej składowej na jej rezerwę może spowodować przeładowanie tej ostatniej. Ogólnie biorąc, możemy powiedzieć, że zarówno w celu zapewniania niezawodności, jak i w celu pokonywania szczytowych obciążen jest istotne, aby dysponować zapasowymi zasobami. Zaletą systemów rozproszonych jest ich „wrodzona”, potencjalna zdolność do tolerowania awarii i skalowalności, co wynika z mnogości występujących w nich zasobów. Jednakże nieodpowiedni projekt może zniweczyć ten potencjał. Zagadnienia tolerowania uszkodzeń oraz skalowalności wymagają od projektu rozproszenia sterowania i danych.

Bardzo wielkie systemy rozproszone pozostają wciąż w dużym stopniu w sferze teorii. Nie ma magicznych sposobów zapewniania skalowalności systemu. Łatwiej jest wykazać, dlaczego istniejące konstrukcje nie są skalowalne. Przedstawimy kilka projektów, które stwarzają problemy w kontekście skalowalności i zaproponujemy możliwe ich rozwiązania.

Jedną z zasad projektowania bardzo wielkich systemów jest wymaganie, aby żądania obsługi pochodzące od dowolnego jego elementu były ograniczone przez stałą, która jest niezależna od liczby węzłów w systemie. Dowolny mechanizm usługowy, którego wymagania są proporcjonalne do rozmiaru systemu, jest skazany na zablokowanie, gdy rozmiary systemu przekroczą pewną wartość. Dodawanie nowych zasobów nie złagodzi tego problemu. Pojemność takiego mechanizmu po prostu ogranicza rozrost systemu.

Do budowania systemów skalowalnych (i tolerujących uszkodzenia) nie powinno się używać schematów sterowania centralnego oraz skoncentrowanych zasobów. Przykładami jednostek skoncentrowanych są centralne serwery uwierzytelniania, centralne serwery nazw oraz centralne serwery plików.

Centralizacja jest formą funkcjonalnej asymetrii między maszynami tworzącymi system. Idealnym jej przeciwieństwem jest konfiguracja funkcjonalnie symetryczna, w której wszystkie maszyny składowe odgrywają równorzędne role w działaniu systemu, dzięki czemu każda maszyna jest w pewnym stopniu autonomiczna. W praktyce spełnienie tej zasady jest prawie niemożliwe. Na przykład wdrożenie do systemu maszyn bez dyskowych narusza jego symetrię funkcjonalną, gdyż bez dyskowe stacje robocze będą zależeć od dysku centralnego. Niemniej jednak autonomiczność i symetria są ważnymi celami, ku którym warto podążać.

Praktycznym przybliżaniem konfiguracji symetrycznej i autonomicznej jest *tworzenie gron* (grupowanie: ang. *clustering*). System składa się z zestawu połautonomicznych gron. Grono składa się ze zbioru maszyn i przydzielonego do nich serwera grona. Aby odniesienia do zasobów między gronami nie były zbyt częste, zamówienia ze strony maszyn grona powinny być w większości obsługiwane przez serwer grona. Oczywiście schemat ten zależy od zdolności lokalizowania odwołań do zasobów i właściwego rozmieszczenia jednostek składowych. Jeśli grono jest dobrze wyważone, tzn. jeśli przydzielony do grona serwer spełnia wszystkie jego żądania, to może służyć jako modularny blok montażowy przy powiększaniu skali systemu.

Poważnego przemyślenia przy projektowaniu dowolnych usług wymaga struktura procesowa serwera. Serwery powinny działać wydajnie w okresach szczytowych, kiedy setki aktywnych klientów żądają jednocześnie obsługi. Zastosowanie serwera jednoprocesowego z pewnością nie jest dobrym pomysłem, gdyż ilekroć wystąpi zamówienie na dyskową operację wejścia-wyjścia, tylekroć cała obsługa będzie wstrzymywana. Przydzielenie każdemu klientowi osobnego procesu jest lepszym rozwiązaniem. Trzeba jednak rozważyć koszty częstego przełączania kontekstu między procesami. Do tego dochodzi konieczność dzielenia informacji przez wszystkie procesy usługowe.

Wydaje się, że jednym z najlepszych rozwiązań konstrukcji serwera jest zastosowanie procesów lekkich, czyli wątków, które omówiliśmy w p. 4.5. Abstrakcja grupy procesów lekkich odpowiada sytuacji, w której występuje wiele wątków sterowania powiązanych z pewnymi dzielonymi zasobami. Proces lekki nie jest na ogół związany z jednym klientem. Planowanie wątków może się odbywać z wywłaszczeniem lub bez wywłaszczenia. Jeśli wątkom pozwala się pracować do końca (bez wywłaszczenia), to ich wspólne dane nie wymagają jawniej ochrony. W przeciwnym razie musi być użyty jawnym mechanizm blokowania zasobów. Pewne rodzaje schematów procesów lekkich mają niewątpliwie istotne znaczenie w skalowalności serwerów.

16.6 ■ Podsumowanie

System rozproszony umożliwia użytkownikom dostęp do rozmaitych zasobów. Dostęp do zasobów dzielonych może się odbywać na zasadzie wędrówki danych, wędrówki obliczeń lub wędrówki zadań. W rozproszonym systemie plików należy uwzględnić dwa podstawowe zagadnienia: przezroczystość (czy dostęp użytkownika do plików odbywa się zawsze tak samo, niezależnie od ich położenia w sieci?) oraz lokalność (w którym miejscu systemu powinny być przechowywane pliki?).

Rozproszony system plików jest budowany powyżej usług sieciowych niższego poziomu. Najpopularniejszą sieciową usługą niższego poziomu jest zdalne wywołanie procedury (RPC). Wywołanie RPC jest strukturalnym komunikatem adresowanym do demona zdalnych wywołań procedur, który nasłuchuje ich w porcie zdalnego systemu. Wywołanie takie zawiera identyfikator funkcji (procedury), która ma zostać wykonana, oraz parametry, które należy jej przekazać. W wyniku zdalnego wywołania funkcja jest wykonywana zgodnie z zamówieniem, przy czym wszelkie ewentualne wyniki są przekazywane zamawiającemu w osobnym komunikacie. W celu ułatwienia za-programowania i zwiększenia wydajności docelowego procesu można zastosować wątki, co pozwala na obsługiwanie zamówienia od początku do końca w jednym wątku i jednocześnie wykonywanie przez jego sąsiadów tych samych czynności w odniesieniu do innych zamówień.

System rozproszony może być narażony na rozmaite rodzaje awarii. Aby system rozproszony mógł tolerować uszkodzenia, musi umieć je wykrywać i dokonywać własnej rekonfiguracji. Po naprawieniu uszkodzenia system musi być ponownie zrekonfigurowany.

■ Ćwiczenia

- 16.1 Jakie są zalety i wady uczynienia sieci komputerowej przezroczystą dla użytkownika?
- 16.2 Jakie najgorsze trudności muszą pokonać projektanci implementujący system przezroczysty pod względem sieci?
- 16.3 Wędrówka procesów w sieci heterogenicznej jest zazwyczaj niemożliwa z powodu różnic w budowie sprzętu i systemów operacyjnych. Opisz metodę wędrówki procesów między różnymi architekturami sprzętowymi, które działają pod kontrolą:
 - (a) takich samych systemów operacyjnych,
 - (b) różnych systemów operacyjnych.

- 16.4 Aby zbudować odporny system rozproszony, musisz znać możliwe rodzaje jego awarii.
- (a) Wymień możliwe rodzaje uszkodzeń w systemie rozproszonym.
 - (b) Określ, które elementy Twojego wykazu odnoszą się również do systemu scentralizowanego.
- 16.5 Czy jest istotne, aby wiedzieć, czy wysłany przez Ciebie komunikat dotarł bezpiecznie do miejsca przeznaczenia? Jeśli odpowiesz „tak”, to wyjaśnij – dlaczego? Jeśli odpowiesz „nie”, to podaj stosowne przykłady.
- 16.6 Przedstaw algorytm rekonstrukcji pierścienia logicznego po wystąpieniu w nim awarii któregoś z procesów.
- 16.7 Rozważ system rozproszony z dwoma stanowiskami A i B. Zastanów się, czy stanowisko A może rozróżnić następujące zdarzenia:
- (a) Stanowisko B przestaje pracować.
 - (b) Łącze między A i B zostaje przerwane.
 - (c) Stanowisko B jest wyjątkowo przeładowane pracą, wskutek czego jego czas reakcji jest 100 razy dłuższy niż zazwyczaj.
- Jakie są konsekwencje Twoich odpowiedzi w odniesieniu do usuwania skutków awarii w systemach rozproszonych?

Uwagi bibliograficzne

Forsdick i in. [138] oraz Donnelley [115] omówili systemy operacyjne dla sieci komputerowych. Przegląd rozproszonych systemów operacyjnych zaproponowali Tanenbaum i van Renesse w artykule [417].

Rozważania dotyczące struktur rozproszonych systemów operacyjnych przedstawili Popek i Walker [333] (system Locus), Cheriton i Zwaenepoel [70] (jadro V), Ousterhout i in. [320] (sieciowy system operacyjny Sprite), Balkovich i in. [22] (projekt Athena), a także Tanenbaum i in. [419] oraz Mullender i in. [305] (rozproszony system operacyjny Amoeba). Porównania systemów Amoeba i Sprite dokonali Douglis i in. [119]. Mullender w książce [303] wszechstronnie opisał obliczenia rozproszone.

Omówienie zagadnień związanych z równoważeniem obciążenia i ich dzieleniem zaprezentowali: Chow i Abraham [73], Eager i in. [123] oraz Ferguson i in. [132]. Zagadnienia wędrówki procesów omówili w swoich arty-

kułach: Eager i in. [123], Zayas [448], Smith [397], Jul i in. [202], Artsy [16], Douglis i Ousterhout [116 i 118] oraz Eskicioglu [126]. Specjalne wydanie periodyku [15], poświęcone zagadnieniom wędrówki procesów, ukazało się pod redakcją Artsy'ego.

Sposoby wspólnego korzystania z bezczynnej stacji roboczej w rozproszonym, dzielonym środowisku komputerowym zaprezentowali: Nichols [310], Mutka i Livny [306] oraz Litzkow i in. [260].

Organizowanie niezawodnej komunikacji w warunkach występujących awarii omówili Birman i Joseph [38]. Zasadę głoszącą, że obsługa wymagana przez dowolną część systemu powinna być ograniczona przez stałą, która nie zależy od liczby węzłów w systemie, po raz pierwszy sformułowali Barak i Kornatzky w zbiorze publikacji [23].



Rozdział 17

ROZPROSZONE SYSTEMY PLIKÓW

W poprzednim rozdziale omówiliśmy budowę sieci oraz protokoły niskiego poziomu potrzebne do przesyłania komunikatów między systemami. Teraz zajmiemy się jednym z zastosowań tej infrastruktury. *Rozproszony system plików* (ang. *distributed file system* – DFS) jest rozproszoną implementacją klasycznego modelu systemu plików z podziałem czasu, w którym wielu użytkowników dzieli pliki i zasoby pamięci (zob. rozdz. 10). Zadaniem rozproszonego systemu plików jest umożliwienie tego samego rodzaju dzielenia zasobów w warunkach fizycznego rozmieszczenia plików na różnych stanowiskach systemu rozproszonego.

W tym rozdziale omawiamy różne sposoby projektowania oraz realizacji rozproszonego systemu plików. Najpierw przedstawimy powszechnie znane podstawy konstruowania rozproszonych systemów plików. Następnie zilustrujemy je, dokonując przeglądu następujących rozproszonych systemów plików: UNIX United, NFS, Andrew, Sprite i Locus. Taki sposób prezentacji przyjmujemy dlatego, że rozproszone systemy plików są terenem aktywnych badań i wiele z zagadnień projektowych, które tu przedstawiamy, podlega ciągłej weryfikacji. Dokonując przeglądu tych przykładowych systemów, mamy nadzieję przybliżyć sens rozważań dotyczących projektowania systemów operacyjnych, jak również wskazać bieżące obszary ich badań.

17.1 ■ Podstawy

System rozproszony jest zbiorem luźno powiązanych maszyn, połączonych za pomocą sieci komunikacyjnej. Terminu *maszyna* używamy do określenia zarówno komputera głównego, jak i stacji roboczej. W odniesieniu do konkret-

nej maszyny w systemie rozproszonym reszta maszyn i należące do nich zasoby są *zdalne*, podczas gdy własne zasoby danej maszyny są *lokalne*.

Aby wyjaśnić strukturę rozproszonego systemu plików, musimy zdefiniować terminy: usługa, serwer oraz klient. *Usługa* (ang. *service*) jest jednostką oprogramowania działającą na jednej lub wielu maszynach, wypełniającą dla nieznanego uprzednio klienta funkcję konkretnego typu. *Serwer* (ang. *server*) jest to oprogramowanie usługowe wykonywane na jednej maszynie. *Klientem* (ang. *client*) jest proces, który zamawia usługę za pomocą zbioru operacji tworzących *interfejs klienta* (ang. *client interface*). Niekiedy definiuje się interfejs niskiego poziomu do rzeczywistej współpracy między maszynami – nazywamy go *interfejsem międzymaszynowym* (ang. *intermachine interface*).

Używając zdefiniowanej powyżej terminologii, możemy powiedzieć, że system plików dostarcza klientom usług plikowych. Interfejs klienta służący do korzystania z pliku tworzy się za pomocą zbioru elementarnych *operacji plikowych*, takich jak tworzenie pliku, usuwanie pliku, czytanie pliku i pisanie do pliku. Elementarną składową sprzętową nadzorowaną przez serwer plików jest zbiór lokalnych urządzeń pamięci pomocniczej (najczęściej dysków magnetycznych), na których pliki są przechowywane i z których odzyskuje się je stosownie do potrzeb klienta.

W rozproszonym systemie plików klienci, serwery i urządzenia pamięci są rozmieszczone na różnych maszynach. Tak więc działania usługowe muszą być wykonywane w sieci, a zamiast pojedynczego, skoncentrowanego magazynu danych istnieje wiele niezależnych urządzeń pamięci. Jak się przekonamy, konkretne konfiguracje i implementacje rozproszonych systemów plików mogą różnić się między sobą. Istnieją konfiguracje, w których serwery pracują na wydzielonych maszynach, a także konfiguracje, w których jedna maszyna może być zarówno serwerem, jak i klientem. Rozproszony system plików można zrealizować jako część rozproszonego systemu operacyjnego albo też w warstwie oprogramowania, której zadaniem jest organizowanie komunikacji między konwencjonalnymi systemami operacyjnymi i systemami plików. Cechą wyróżniającą rozproszone systemy plików jest mnogość i niezależność klientów i serwerów w systemie.

Idealny rozproszony system plików powinien wydawać się swoim klientom zwykłym, skoncentrowanym systemem plików. Mnogość i rozproszenie jego serwerów i urządzeń pamięci powinny być przezroczyste. Oznacza to, że w interfejsie klienta rozproszonego systemu plików nie powinno dać się odróznić plików lokalnych od zdalnych. Lokalizowanie plików i organizowanie transportu danych powinno należeć do zadań rozproszonego systemu plików. Przezroczysty rozproszony system plików ułatwia użytkownikowi zmianę miejsca, przenosząc jego środowisko (tzn. osobisty katalog użytkownika) tam, gdzie użytkownik rejestruje się w systemie.

Najważniejszą miarą *wydajności* rozproszonego systemu plików jest czas potrzebny do spełnienia rozmaitych zamówień na usługi. W systemach konwencjonalnych czas ten składa się z czasu dostępu do dysku i małej porcji czasu procesora. Natomiast w rozproszonych systemach plików dostęp zdalny jest obarczony dodatkowym kosztem, charakterystycznym dla struktury rozproszonej. Koszt ten obejmuje zarówno czas potrzebny na dostarczenie zamówienia do serwera, jak i czas przesyłania klientowi odpowiedzi przez sieć. W każdym kierunku do rzeczywistego przesyłania informacji dochodzi jeszcze czas zużywany przez procesor na wykonanie zaprogramowanego protokołu komunikacji. Wydajność rozproszonego systemu plików można traktować jako dodatkowy wymiar jego przezroczystości. Należy przez to rozumieć, że wydajność idealnego rozproszonego systemu plików powinna być porównywalna z działaniem zwykłego systemu plików.

Fakt, że rozproszony system plików zarządza zbiorom rozrzuconych w przestrzeni urządzeń pamięci jest kluczowym wyróżnikiem tego rodzaju systemu. Cała przestrzeń pamięci zarządzana przez rozproszony system plików składa się z różnych, odlegle położonych, mniejszych obszarów pamięci. Na ogół istnieje odpowiedność między tymi składowymi elementami przestrzeni pamięci a zbiorami plików. Terminu *jednostka składowa* (ang. *component unit*) używamy w celu określenia najbliższego zbioru plików, który można przechować w jednej maszynie niezależnie od innych jednostek. Wszystkie pliki należące do tej samej jednostki składowej muszą pozostać w tym samym miejscu.

17.2 ■ Nazewnictwo i przezroczystość

Nazwanie (ang. *naming*) jest odwzorowaniem między obiektami logicznymi a fizycznymi. Na przykład użytkownicy mają do czynienia z logicznymi obiektami danych reprezentowanymi przez nazwy plików, podczas gdy system manipuluje fizycznymi blokami danych przechowywanymi na ściązkach dyskowych. Użytkownik na ogół odwołuje się do pliku za pomocą nazwy tekstowej, która jest odwzorowywana na niższym poziomie na identyfikator liczbowy, a ten z kolei – na bloki dyskowe. To wielopoziomowe odwzorowywanie tworzy dla użytkowników abstrakcję pliku, która ukrywa szczegóły związane z tym, jak i gdzie plik jest rzeczywiście przechowywany na dysku.

W *przezroczystym* (ang. *transparent*) rozproszonym systemie plików abstrakcja pliku otrzymuje nowy wymiar – jest nim ukrywanie miejsca przechowywania pliku w sieci. W zwykłym systemie plików dziedziną odwzorowywania nazw są adresy na dysku. W rozproszonym systemie plików dziedzina ta jest poszerzona o specyfikację maszyny, na której dysku dany plik

jest przechowywany. Jeszcze jeden krok dalej w traktowaniu plików jako abstrakcji prowadzi do możliwości *zwielokrotniania pliku* (ang. *file replication*). Dla danej nazwy pliku wartością odwzorowania jest zbiór umiejscowień kopii tego pliku. W tej abstrakcji ukrywa się zarówno istnienie wielu kopii pliku, jak i ich położenie.

17.2.1 Struktury nazewnicze

Istnieją dwa powiązane ze sobą pojęcia dotyczące odwzorowywania nazw w rozproszonym systemie plików, które należy rozróżnić:

- **Przeczcystość położenia:** Nazwa pliku nie daje jakiegokolwiek wskaźnika odnośnie do fizycznego miejsca przechowywania pliku.
- **Niezależność położenia:** Nazwy pliku nie trzeba zmieniać, gdy plik zmienia swoje fizyczne umiejscowienie.

Obie definicje są związane z uprzednio omówionymi poziomami nazewniczymi, ponieważ pliki mają różne nazwy na różnych poziomach (tj. na poziomie nazw tekstowych użytkownika i systemowym poziomie identyfikatorów liczbowych). Niezależny od położenia pliku schemat nazywania jest odwzorowaniem dynamicznym, ponieważ w różnych chwilach temu samemu plikowi mogą odpowiadać różne miejsca. Dlatego niezależność położenia pliku jest silniejszą cechą niż przeczcystość jego położenia.

W praktyce większość istniejących rozproszonych systemów plików realizuje statyczne, przezczyste pod względem położenia odwzorowywanie nazw z poziomu użytkownika. Systemy te jednak nie umożliwiają wędrówek plików. Oznacza to, że nie jest możliwa automatyczna zmiana umiejscowienia pliku. Stąd pojęcie niezależności położenia nie ma w tych systemach żadnego znaczenia. Pliki są na stałe przypisane do określonego zbioru bloków dyskowych. Pliki i dyski można ręcznie przenosić między maszynami, natomiast wędrówka plików oznacza działanie automatyczne, inicjowane przez system operacyjny. Tylko system Andrew (zob. p. 17.6.3) i niektóre eksperymentalne systemy plików urzeczywistniają niezależność położenia i mobilność plików. W systemie Andrew mobilność plików służy głównie celom administracyjnym. Protokół systemu Andrew umożliwia wędrówkę jednostek składowych tego systemu w celu spełniania wysokopoziomowych życzeń użytkownika bez zmiany nazw z poziomu użytkownika oraz niskopoziomowych nazw odpowiednich plików.

Istnieje kilka aspektów głębiej różnicujących pojęcia niezależności położenia i statycznej przeczcystości położenia plików. Oto one:

- Rozdział danych od ich umiejscowienia – przejawiający się w niezależności położenia pliku – polepsza abstrakcyjność plików. Nazwa pliku powinna określać najistotniejsze atrybuty pliku, do których należy jego zawartość, lecz nie położenie. Pliki niezależne od położenia można traktować jako logiczne pojemniki danych, nie związane z żadnym szczególnym miejscem w pamięci. Jeśli istnieje tylko statyczna przezroczystość położenia, to nazwa pliku nieustannie określa specyficzny, choć ukryty, zbiór fizycznych bloków dyskowych.
- Dzięki statycznej przezroczystości położenia plików użytkownicy mogą wygodnie realizować dzielenie danych. Mogą oni wspólnie korzystać ze zdalnych plików po prostu przez nazywanie ich w konwencji przezroczystej wobec ich położenia, tak jak w odniesieniu do plików lokalnych. Niemniej jednak takie dzielenie przestrzeni pamięci jest uciążliwe, gdyż nazwy logiczne są wciąż statycznie powiązane z fizycznymi urządzeniami pamięci. Niezależność położenia pozwala zarówno na dzielenie obszarów samej pamięci, jak i na dzielenie obiektów danych. Jeśli pliki są mobilne, to przestrzeń pamięci całego systemu wygląda jak jeden zasób wirtualny. Potencjalną korzyścią takiego ujęcia jest możliwość równoważenia wykorzystania dysków w obrębie systemu.
- Niezależność położenia oddziela hierarchię nazewniczą od hierarchii urządzeń pamięci oraz od struktury międzykomputerowej. Dla porównania, przy użyciu statycznej przezroczystości położenia (choćże nazwy są przezroczyste) łatwo można uwidoczyć związek między jednostkami składowymi a maszynami. Maszyny są skonfigurowane w układ podobny do struktury nazw. Może to nakładać niepotrzebne ograniczenia na architekturę systemu i powodować konflikty z innymi zagadnieniami. Serwer odpowiedzialny za katalog główny jest przykładem struktury wynikającej z hierarchii nazewniczej i zaprzeczeniem reguł decentralizacji.

W wyniku oddzielenia nazwy od położenia pliku do plików rezydujących w systemach zdalnych serwerów mogą mieć dostęp różni klienci. W istocie, mogą to być klienci bezdyskowi, którzy polegają na serwerach co do wszystkich plików, łącznie z jądrem systemu operacyjnego. Rozruch takich maszyn wymaga jednak specjalnego postępowania. Rozważmy kwestię umieszczenia jądra systemu operacyjnego w bezdyskowej stacji roboczej. Pozbawiona dysku stacja robocza nie ma jądra, nie może zatem skorzystać z oprogramowania rozproszonego systemu plików w celu pozyskania jądra. Zamiast tego wywołuje się specjalny protokół startowy, zapamiętany u klienta w pamięci zdatnej tylko do czytania (ROM). Protokół ten umożliwia pracę w sieci i służy do odzyskania tylko jednego, specjalnego pliku (jądra lub programu rozru-

chowego) z ustalonego miejsca. Gdy już jądro systemu zostanie przekopiowane przez sieć i załadowane, wówczas jego rozproszony system plików udostępnia wszystkie inne pliki systemu operacyjnego. Bez dyskowi klienci mają wiele zalet, do których należy ich niski koszt (ponieważ żadna z tych maszyn nie potrzebuje dysku) i większa wygoda (gdy pojawią się ulepszenia w systemie operacyjnym, wówczas wystarczy tylko zmodyfikować kopię serwera, a nie wszystkich klientów). Do wad zalicza się dodatkową złożoność protokołów rozruchowych i spowolnienie działania, spowodowane korzystaniem z sieci zamiast lokalnego dysku.

Obecnie istnieje tendencja do zaopatrywania klientów w dyski. Rośnie pojemność napędów dysków, gwałtownie maleją ich ceny i niemal co roku pojawiają się ich nowe generacje. Stwierdzenia tego nie można odnieść do sieci, których cykl rozwojowy wynosi od 5 do 10 lat. Systemy – ogólnie biorąc – rosną szybciej niż sieci, toteż należy dokładać specjalnych starań, aby ograniczać dostęp do sieci w celu poprawiania przepustowości systemu.

17.2.2 Schematy nazewnicze

Są trzy główne podejścia do schematów nazywania w rozproszonych systemach plików. W najprostszym ujęciu pliki są nazywane za pomocą pewnych kombinacji nazw ich stacji macierzystych i nazw lokalnych, co gwarantuje jednoznaczność nazw w obrębie całego systemu. Na przykład w systemie Ibis plik jest identyfikowany jednoznacznie za pomocą nazwy *stacja-macierzysta:nazwa-lokalna*, przy czym *nazwa-lokalna* jest ścieżką taką jak w systemie UNIX. Ten schemat nazewniczy nie jest ani przezroczysty, ani niezależny od położenia. Niemniej jednak zarówno w odniesieniu do plików lokalnych, jak i zdalnych można stosować te same operacje plikowe. Strukturę rozproszonego systemu plików tworzy zbiór izolowanych jednostek składowych, będących zupełnie zwykłymi systemami plików. W tej metodzie jednostki składowe pozostają wyizolowane, choć istnieją środki odwoływanego się do plików zdalnych. W niniejszym tekście nie będziemy się więcej zajmować tym schematem.

Drugie podejście spopularyzowało się dzięki sieciowemu systemowi plików (ang. *Network File System* – NFS) dla komputerów Sun. System plików NFS wchodzi w skład pakietu sieciowego ONC+, który będzie dostarczany przez wielu dostawców systemu UNIX. System NFS oferuje środki do przyłączania katalogów zdalnych do katalogów lokalnych, co daje wrażenie spójnego drzewa katalogów. Wczesne wersje systemu NFS umożliwiały tylko przezroczysty dostęp do uprzednio zamontowanych katalogów. Po wprowadzeniu możliwości automontażu montowania dokonuje się na żądanie na postawie tablicy punktów montażu i nazw struktury plików. Istnieje też pewna

integracja składowych systemu, wspomagająca przezroczyste dzielenie. Integracja ta jest jednak ograniczona i niejednolita, ponieważ każda maszyna może przyłączać różne zdalne katalogi do swojego drzewa. Wynikowa struktura jest uniwersalna. Jest to zazwyczaj las drzew systemu UNIX z dzielonymi poddrzewami.

Pełną integrację składowych systemów plików osiąga się przy użyciu trzeciego podejścia. Jedna globalna struktura nazw obejmuje wszystkie pliki w systemie. W idealnej sytuacji struktura połączonych systemów plików powinna być izomorficzna ze strukturą zwykłego systemu plików. Jednak w praktyce istnieje wiele specjalnych plików (np. pliki urządzeń w systemie UNIX i zależne od maszyny katalogi binarne), które powodują, że trudno ten cel osiągnąć. Różne odmiany tego podejścia przeanalizujemy przy omawianiu systemów UNIX United, Locus, Sprite i Andrew w p. 17.6.

Ważnym kryterium oceny struktur nazewnictwowych jest ich złożoność administracyjna. Najbardziej złożona i najtrudniejsza w utrzymaniu jest struktura systemu NFS. Ponieważ dowolny katalog zdalny może być przyłączony gdziekolwiek w lokalnym drzewie katalogów, struktura wynikowej hierarchii może być bardzo zagmatwana. Efektem nieoperatywności jakiegoś serwera może być niedostępność pewnego dowolnego zbioru katalogów na różnych maszynach. Ponadto oddzielny mechanizm akredytacji ustala, która maszyna jaki katalog może dołączyć do swojego drzewa. Mechanizm ten może powodować sytuacje, w których w komputerze jednego klienta użytkownik może mieć dostęp do zdalnego drzewa katalogów, podczas gdy w innym komputerze temu samemu użytkownikowi dostępu tego się zabrania.

17.2.3 Techniki implementacji

Implementacja przezroczystego nazewnictwa wymaga zapewnienia środków odwzorowywania nazwy pliku na przynależne mu miejsce. Konieczność utrzymywania tego odwzorowania w stanie zdatnym do użytku skłania do zestawiania zbiorów plików w jednostki składowe i dokonywania odwzorowań całych jednostek składowych, a nie pojedynczych plików. Takie grupowanie służy również celom administracyjnym. W systemach uniksopodobnych stosuje się hierarchiczne drzewo katalogów w celu umożliwienia odwzorowywania nazw na położenia plików i rekurencyjnego grupowania plików w katalogach.

W celu polepszenia dostępności istotnych informacji odwzorowujących można posłużyć się takimi metodami, jak zwielokrotnianie oraz lokalna pamięć podręczna – lub użyć jednocześnie obu metod. Jak już zauważyliśmy, niezależność od położenia oznacza możliwość zmian odwzorowania w czasie. Wobec tego powielanie odwzorowań powoduje, że proste, lecz spójne ich uaktualnianie staje się niemożliwe. Techniką obejścia tej przeszkody jest

wprowadzenie niskopoziomowych identyfikatorów plików niezależnych od położenia (ang. *location-independent file identifiers*). Tekstowe nazwy plików są odwzorowywane na identyfikatory plików na niższym poziomie, które wskazują, do której jednostki składowej dany plik należy. Identyfikatory te są również niezależne od położenia. Można je swobodnie zwielokrotniać i przechowywać w pamięciach podręcznych, bez obawy o ich unieważnienie powodowane wędrówką jednostek składowych. Nieuniknioną ceną jest jednak drugi poziom odwzorowań, służący do odwzorowywania jednostek składowych na ich umiejscowienie i wymagający prostego, lecz spójnego mechanizmu aktualniania. Zastosowanie takich niskopoziomowych, niezależnych od położenia identyfikatorów do implementacji drzew katalogów, podobnych do stosowanych w systemie UNIX, czyni całą hierarchię niewrażliwą na przemieszczanie jednostek składowych. Jedynym elementem, który się zmienia, jest odwzorowanie położenia jednostki składowej.

Powszechnym sposobem implementowania takich niskopoziomowych identyfikatorów jest użycie *nazw strukturalnych*. Są to ciągi bitów składające się przeważnie z dwóch części. Pierwsza część identyfikuje jednostkę składową, do której dany plik należy. Część druga określa konkretny plik w obrębie jednostki. Są możliwe warianty z większą liczbą części. Nieznieniącą cechą nazw strukturalnych jest jednak to, że poszczególne części nazwy są jednoznaczne we wszystkich przypadkach tylko w kontekście pozostałych części. Jednoznaczność we wszystkich przypadkach można osiągnąć, dopilnowując, aby nie użyto ponownie nazwy będącej jeszcze w użyciu, albo odpowiednio zwiększając liczbę bitów (tę metodę stosuje się w systemie Andrew), albo używając znaczników czasu jako jednej z części nazwy (jak w systemie Apollo Domain). Innym sposobem może być zastosowanie systemu realizującego przezroczystość położenia (takiego jak Ibis) i dodanie nowego poziomu abstrakcji w celu tworzenia schematu nazewniczego niezależnego od położenia plików.

Sposób użycia technik grupowania plików w jednostki składowe oraz zastosowanie niskopoziomowych, niezależnych od położenia identyfikatorów plików widać na przykładzie systemów Andrew i Locus.

17.3 ■ Zdalny dostęp do plików

Weźmy pod uwagę użytkownika, który zgłasza chęć korzystania ze zdalnego pliku. Założmy, że serwer przechowujący dany plik został zlokalizowany za pomocą odpowiedniego schematu nazewniczego. Aby spełnić oczekiwania użytkownika na zdalny dostęp do danych, trzeba będzie spowodować rzeczywiste przesyłanie danych.

Jednym ze sposobów umożliwiających to przesyłanie jest metoda *obsługi zdalnej* (ang. *remote service*). Zamówienia na dostępy są kierowane do serwera. Maszyna serwera wykonuje odpowiednie operacje, a ich wyniki są przekazywane do użytkownika. Jednym z najpopularniejszych sposobów implementowania obsługi zdalnej jest schemat *zdalonego wywołania procedury* (RPC), który omówiliśmy w p. 16.3.1. Zwracamy uwagę na bezpośrednią analogię między metodami dostępu do dysku w zwykłym systemie plików a metodą dostępu zdalnego w rozproszonym systemie plików. Metoda dostępu zdalnego jest podobna do wykonywania dostępu do dysku przy każdym zamówieniu.

Aby zapewnić zadowalające działanie schematu obsługi zdalnej, można zastosować odmianę *przechowywania podręcznego*. W zwykłym systemie plików przechowywanie podręczne uzasadnia się dążeniem do zmniejszenia liczby dyskowych operacji wejścia-wyjścia (co poprawia wydajność systemu). W rozproszonych systemach plików celem takiego postępowania jest zarówno zmniejszenie liczby dyskowych operacji wejścia-wyjścia, jak i zmniejszenie ruchu w sieci. W następujących dalej punktach omówimy różne zagadnienia dotyczące implementowania pamięci podręcznej w rozproszonym systemie plików i porównamy je z podstawowym schematem realizowania usług zdalnych.

17.3.1 Podstawowy schemat przechowywania podręcznego

Idea pamięci podręcznej jest prosta. Jeśli danych potrzebnych do wykonania zamówienia nie ma jeszcze w pamięci podręcznej, to ich kopię przenosi się z serwera do systemu klienta. Dostęp wykonyuje się na kopii przechowanej w pamięci podręcznej. Pomyśl polega na pozostawianiu ostatnio używanego bloku w pamięci podręcznej, aby powtórny dostęp do tej samej informacji mógł się odbyć lokalnie, bez dodatkowego ruchu w sieci. W celu utrzymywania ograniczonego rozmiaru pamięci podręcznej stosuje się jakąś politykę zastępowania zawartych w niej danych (np. LRU, czyli usuwanie danych używanych najdawniej). Nie ma bezpośredniego związku między dostępmi a odwołaniami do serwera. Pliki są zawsze utożsamiane z jedną, główną kopią rezydującą w maszynie usługodawczej, ale ich kopie (części) są rozrzucone w różnych pamięciach podręcznych. Kiedy przechowywana podręcznie kopia pliku ulega zmianom, wtedy zmiany te muszą znaleźć odzwierciedlenie w kopii głównej, aby zachować niezbędną semantykę spójności danych. Zapewnianie zgodności kopii przechowywanych podręcznie z plikiem głównym nosi miano *problemu spójności pamięci podręcznej* (ang. *cache consistency problem*): omówimy go w p. 17.3.4. Uważny czytelnik dostrzeże, że podręczne przechowywanie danych w rozproszonym systemie plików z powodzeniem można by nazwać *sieciową pamięcią wirtualną*, działa ono podobnie jak pa-

mięć wirtualna stronnicowana na żądanie, z tym że pamięć pomocnicza nie znajduje się na lokalnym dysku, lecz w zdalnym serwerze.

Ziarnistość danych przechowywanych w pamięciach podręcznych może się zmieniać od bloków pliku aż po całe pliki. Na ogół przechowuje się podręcznie więcej danych, niż potrzeba ich do uzyskania pojedynczego dostępu, tak że dane w pamięci podręcznej mogą posłużyć do wielu odwołań. Procedura ta przypomina czytanie dysku z wyprzedzeniem (zob. p. 11.5.2). W systemie Andrew przechowuje się podręcznie pliki w dużych kawałkach (64 KB). Inne systemy omawiane w tym rozdziale umożliwiają podręczne przechowywanie poszczególnych bloków sterowane zamówieniami klientów. Zwiększenie jednostki przechowywania podręcznego powiększa współczynnik trasowania, lecz również zwiększa „karę” za chybienie, gdyż w razie nieznalezienia informacji w pamięci podręcznej jest wymagane przesłanie większej porcji danych. Rośnie wówczas również zagrożenie niespójnością danych. Przy wyborze jednostki przechowywanej w pamięci podręcznej bierze się pod uwagę takie parametry, jak jednostka przesyłania w sieci i jednostka usługi - protokołu zdalnych wywołań procedur (jeśli używa się tego protokołu). Sieciowa jednostka przesyłania (dla sieci Ethernet – pakiet) wynosi około 1,5 KB, toteż większe jednostki danych przechowywanych podręcznie wymagają dzielenia na części przy wysyłaniu i scalania przy odbiorze.

Zarówno rozmiar bloku, jak i ogólny rozmiar pamięci podręcznej jest oczywiście ważny w schematach podręcznego przechowywania bloków. W systemach uniksopodobnych typowy rozmiar bloku wynosi 4 KB lub 8 KB. Dla dużych pamięci podręcznych (powyżej 1 MB) godne polecenia są większe bloki (powyżej 8 KB). W małych pamięciach podręcznych duże bloki są mniej przydatne, ponieważ w takich pamięciach mieści się ich mniej.

17.3.2 Umiejscowienie pamięci podręcznej

Przejdziemy teraz do zagadnienia wyboru miejsca dla podręcznie przechowywanych danych. Dyskowe pamięci podręczne mają jedną wyraźną przewagę nad pamięcią podręczną organizowaną w pamięci głównej – niezawodność. Jeśli pamięć podręczna znajduje się w pamięci ulotnej, to zmiany wprowadzone w przechowywanych w niej danych są tracone w wypadku awarii. Co więcej, jeśli dane są przechowywane podręcznie na dysku, to pozostają tam podczas usuwania skutków awarii i nie trzeba sprowadzać ich ponownie. Z drugiej strony, pamięci podręczne organizowane w pamięci głównej również mają kilka zalet:

- Jeśli pamięć podręczna jest w pamięci głównej, to można korzystać z bezdyskowych stacji roboczych.

- Dostęp do danych przechowywanych podręcznie w pamięci głównej jest szybszy niż do danych na dysku.
- Rozwój technologii zmierza w kierunku wytwarzania coraz większych i tańszych pamięci operacyjnych. Osiągany wzrost efektywności tych pamięci zdaje się zapowiadać przewyższenie zalet dyskowych pamięci podręcznych.
- Pamięci podręczne serwerów (te, które służą do przyspieszania dyskowych operacji wejścia-wyjścia) będą organizowane w pamięci głównej niezależnie od tego, gdzie znajdują się pamięci podręczne użytkownika; umieszczenie pamięci podręcznej w pamięci głównej również w przypadku maszyny użytkownika pozwala zbudować jeden mechanizm przechowywania podręcznego, z którego mogą korzystać zarówno serwery, jak i użytkownicy (zrobiono tak w systemie Sprite).

Wiele implementacji zdalnego dostępu można uważać za hybrydę pamięci podręcznej i obsługi zdalnej. Na przykład w systemach NFS i Locus implementacje są oparte na usługach zdalnych, lecz są rozszerzone o przechowywanie podręczne w celu polepszania wydajności. Z kolei podstawą implementacji systemu Sprite jest przechowywanie podręczne, lecz w pewnych sytuacjach zaadaptowano w niej metodę obsługi zdalnej. Zatem, jeśli oceniamy obie te metody, to w rzeczywistości stwierdzamy, w jakim stopniu jedna z nich przeważa nad drugą w danym systemie.

17.3.3 Polityka aktualniania

Polityka zastosowana przy odsyłaniu zmienionych bloków danych z powrotem do głównej kopii pliku w serwerze ma decydujące znaczenie dla działania i niezawodności systemu. Najprostsza polityka polega na przepisywaniu danych na dysk zaraz po tym, gdy znajdą się w dowolnej pamięci podręcznej. Zaletą *przepisywalności* (ang. *write-through*) jest niezawodność. W wypadku awarii systemu klient traci się mało danych. Jednak ta polityka wymaga przy każdym zapisie oczekiwania na przesłanie danych do serwera, co powoduje słabą wydajność operacji pisania. Stosowanie przepisywalnej pamięci podręcznej jest równoważne ze zdalnym obsługiwaniem operacji pisania i korzystaniem z pamięci podręcznej tylko przy czytaniu. Dostęp połączony z przepisywaniem zastosowano w systemie NFS.

Alternatywna polityka polega na opóźnianiu aktualizowania kopii głównej. Zmiany są wprowadzane do pamięci podręcznej, a do serwera są przepisywane w późniejszym czasie. Polityka ta ma dwie zalety w stosunku do ustawnicznego przepisywania. Po pierwsze, ponieważ operacje pisania odnoszą

się do pamięci podręcznej, dostęp do pisania trwa więc znacznie krócej. Po drugie, dane mogą być ponownie zmienione, zanim zostaną przepisane z powrotem, co oznacza, że unika się zbędnego zapisywania ich poprzednich wersji. Niestety, schematy opóźnianego pisania (ang. *delayed-write*) pogarszają niezawodność, gdyż nie zapisane dane zginą przy każdej awarii maszyny użytkownika.

Istnieje kilka odmian polityki opóźnianego pisania, różniących się okresem czasu, w którym zmienione dane są posyłane serwera. Jedna możliwość polega na postaniu bloku do serwera wtedy, kiedy zanosi się na usunięcie bloku z pamięci podręcznej klienta. Dobrze wpływa to na wydajność, lecz niektóre bloki mogą przebywać w pamięci podręcznej klienta przez długi czas, zanim zostaną zapisane z powrotem w serwerze. Wyjściem kompromisowym między tą metodą a polityką ustawniczego przepisywania jest przeglądanie pamięci podręcznej w regularnych odstępach czasu i wysyłanie z niej bloków zmienionych po ostatnim przeglądaniu. W systemie Sprite zastosowano tę politykę z odstępami 30-sekundowymi.

Jeszcze inną odmianą opóźnianego pisania jest zapisywanie danych z powrotem w serwerze dopiero w chwili zamknięcia pliku. Taką politykę pisania przy zamknięciu (ang. *write-on-close*) zastosowano w systemie Andrew. W przypadku plików otwieranych na bardzo krótkie okresy lub rzadko modyfikowanych polityka ta nie zmniejsza istotnie ruchu w sieci. Ponadto polityka pisania przy zamknięciu wymaga opóźniania procesu zamknięcia pliku przez czas przepisywania tego pliku, co zmniejsza zaletę wydajności tak opóźnianego pisania. Zaleta wydajności tej polityki w porównaniu z opóźnianym pisaniem z częściej wykonywanym opróżnianiem bufora staje się widoczna przy plikach otwieranych na długo i często zmienianych.

17.3.4 Spójność

Maszyna klienta musi sprawdzać, czy kopia danych przechowywana w lokalnej pamięci podręcznej jest spojna z kopią główną (więc może być używana), czy też nie. Jeśli okaże się, że dane przechowywane w pamięci podręcznej są nieaktualne, to maszyna klienta przestanie udzielać do nich dostępu. Do pamięci podręcznej trzeba będzie sprowadzić aktualną kopię danych. Są dwie metody weryfikowania aktualności danych przechowywanych podręcznie:

- **Podejście polegające na inicjatywie klienta:** Klient inicjuje sprawdzenie ważności danych, kontaktując się z serwerem i sprawdzając, czy dane lokalne są spojne z kopią główną. Najistotniejsza w tym podejściu jest częstotliwość sprawdzania ważności; to ona przesądza o wynikowej semantyczce spójności. Może się ona ważyć między sprawdzaniem przy każdym do-

stępnie a sprawdzaniem tylko przy pierwszym dostępie do pliku (zasadniczo przy otwieraniu pliku). Kazdy dostęp połączony ze sprawdzaniem ważności jest opóźniony w porównaniu z dostępem obsługowanym przez pamięć podręczną natychmiast. Inna możliwość polega na sprawdzaniu w stałych odstępach czasu. W zależności od częstości sprawdzanie ważności obciąża zarówno sieć, jak i serwer.

- **Podejście polegające na inicjatywie serwera:** Serwer rejestruje, dla każdego klienta z osobna, pliki (lub ich części), które wysyła do pamięci podręcznej. Gdy serwer wykryje możliwość niespójności, wówczas musi reagować. Potencjalna niespójność powstaje wtedy, kiedy plik jest użytkowany w pamięciach podręcznych dwóch klientów, którzy pracują w trybach powodujących konflikty. Jeśli jest zaimplementowana semantyka sesji (zob. p. 10.5.2), to ilekroć serwer otrzyma zamówienie na zamknięcie pliku, który został zmieniony, tylekroć powinien zareagować, powiadając klientów, aby dane w swoich pamięciach podręcznych uznali za nieważne i usunęły je. Klienci, którzy w tym czasie mają otwarty dany plik, usuwają swoje kopie na końcu sesji. Inni klienci usuwają kopie pliku natychmiast. Zgodnie z semantyką sesji serwer nie musi być informowany o otwieraniu plików znajdujących się już w pamięciach podręcznych. Serwer jest pobudzany do działania tylko przy zamknięciu sesji pisania, więc tylko ten rodzaj sesji jest opóźniany. W systemie Andrew zrealizowano semantykę sesji, jak również stosuje się metodę nazywaną *przywołaniem* (ang. *callback*), w której inicjatywa należy do serwera (zob. p. 17.6.3).

Z kolei przy zaimplementowaniu ostrzejszej semantyki spojności, w rozdzaju semantyki uniksowej (zob. p. 10.5.1), serwer musi odgrywać aktywniejszą rolę. Musi on być powiadomiany o każdym otwarciu pliku, a przy każdym otwarciu pliku musi być ujawniony zamierzony tryb jego użycia (czytanie lub pisanie). Jeśli istnieją takie powiadomienia, to serwer może reagować: kiedy wykryje równoczesne otwarcie pliku w trybach mogących powodować konflikty, wtedy może zakazać przechowywania w pamięci podręcznej tego konkretnego pliku (jest tak w systemie Sprite). Zakaz używania pamięci podręcznej powoduje w rzeczywistości przejście do trybu obsługi zdalnej.

17.3.5 Porównanie przechowywania podręcznego i obsługi zdalnej

Wybór między przechowywaniem podręcznym a usługą zdalną oznacza w istocie przeciwstawienie potencjalnie lepszej wydajności zmniejszonej prostocie. Kompromis ten ocenimy, wyliczając zalety i wady obu metod:

- Znaczna liczba dostępów zdalnych może być skutecznie obsłużona za pomocą lokalnej pamięci podręcznej. Położenie nacisku na lokalność wzorców dostępu do plików zwiększa jeszcze atrakcyjność pamięci podręcznej. Dzięki temu większość dostępów zdalnych jest obsługiwana tak szybko jak dostęp lokalny. Co więcej, kontakt z serwerami jest okazjonalny, nie musi następować przy każdym dostępie do danych. W konsekwencji zmniejsza się obciążenie serwera i ruch w sieci, a jednocześnie wzrasta potencjalna skalowalność systemu. W porównaniu z tym, przy użyciu metody obsługi zdalnej każdy dostęp zdalny jest obsługiwany przez sieć. Kara w postaci ruchu w sieci, obciążenia serwera oraz spadku wydajności jest oczywista.
- Całkowity koszt przesyłania siecią dużych porcji danych (przy stosowaniu pamięci podręcznej) jest mniejszy niż przy przesyłaniu serii odpowiedzi na poszczególne zamówienia (co występuje w metodzie obsługi zdalnej).
- Procedury dostępu do dysku serwera mogą być lepiej optymalizowane, jeśli wiadomo, że zamówienia będą zawsze dotyczyć dużych ciągłych segmentów danych, a nie losowych bloków dyskowych.
- Problem spójności pamięci podręcznej jest główną wadą przechowywania podręcznego. Używanie pamięci podręcznej ma przewagę wówczas, gdy dostęp do pisania zdarza się rzadko. Jednak gdy często występuje pisanie, wówczas mechanizmy stosowane do pokonywania problemu spójności wprowadzają istotny dodatkowy koszt, zaznaczający się w wydajności, ruchu w sieci i obciążeniu serwera.
- Aby przechowywanie podręczne przynosiło korzyść, działania powinny przebiegać na maszynach wyposażonych w lokalne dyski lub duże pamięci operacyjne. Dostęp zdalny na maszynach bezdyskowych o małej pojemności pamięci operacyjnej powinien być realizowany za pomocą metody obsługi zdalnej.
- Przy zastosowaniu przechowywania podręcznego, kiedy dane są przesyłane między serwerem a klientem hurtowo, a nie w odpowiedzi na specyficzne zapotrzebowania operacji plikowych, interfejs międzymaszynowy niższego poziomu jest zupełnie różny od zlokalizowanego na wyższym poziomie interfejsu użytkownika. Schemat obsługi zdalnej jest z kolei zwykłym rozwinięciem interfejsu lokalnego systemu plików na sieć. To też interfejs międzymaszynowy odzwierciedla tu lokalny interfejs między użytkownikiem a systemem plików.

17.4 ■ Obsługa doglądana i niedoglądana

Są dwa podejścia odnośnie do informacji przechowywanej po stronie serwera. Serwer odnotowuje każdy dostęp klienta do pliku albo po prostu dostarcza bloki na zamówienie klienta, nie wnikając w sposób ich używania.

Typowy scenariusz *doglądanej obsługi pliku* (ang. *stateful file service*) jest następujący. Przed dostępem do pliku klient musi go otworzyć. Serwer pobiera informację o pliku z dysku, zapamiętuje ją w swojej pamięci i daje klientowi identyfikator połączenia, jednoznaczny w zbiorze klientów i otwartych plików. (W terminach systemu UNIX serwer pobiera i-węzeł pliku, klientowi zaś przekazuje deskryptor pliku, który służy jako indeks do przechowywanej w pamięci operacyjnej tablicy i-węzłów). Identyfikator ten jest używany przy następnych dostępach, aż do końca sesji. Obsługę doglądaną charakteryzuje związek między klientem i serwerem podczas sesji. Po zamknięciu pliku lub uruchomieniu mechanizmu odśmiecania pamięci serwer oddaje do wtórnego użytku niepotrzebny już klientowi obszar w pamięci operacyjnej.

Zaletą obsługi doglądanej jest lepsze działanie systemu. Informacja o pliku jest schowana w pamięci operacyjnej i można ją łatwo uzyskać poprzez identyfikator łączący, dzięki czemu unika się dostępów do dysku. Serwer przechowujący stan będzie ponadto wiedzieć, czy plik został otwarty w trybie sekwencyjnym, więc może wtedy czytać bloki z wyprzedzeniem. Serwery nie przechowujące stanu nie mogą tego zrobić, ponieważ nic nie wiedzą o celu zamówień klienta. Obsługa doglądana nie pozostaje jednak bez wpływu na tolerowanie awarii, ponieważ serwer przechowuje w pamięci operacyjnej informacje o klientach.

Serwer bezstanowy (ang. *stateless file server*) unika informacji o stanie (obsługa niedoglądana), traktując każde zamówienie jako samowystarczalne. Oznacza to, że każde zamówienie w pełni identyfikuje plik i miejsce w pliku (do czytania lub pisania). Serwer nie musi utrzymywać tablicy otwartych plików w pamięci operacyjnej, choć zazwyczaj robi to ze względów wydajnościowych. Co więcej, nie ma powodu do ustanawiania i kończenia połączenia za pomocą operacji łączania i zamykania pliku. Są one zupełnie zbędne, ponieważ każda operacja plikowa jest samowystarczalna i nie jest traktowana jako część sesji. Proces klienta może otworzyć plik, ale nie spowoduje to komunikatu zdalnego. Operacje czytania i pisania będą oczywiście powodować wysyłanie komunikatów zdalnych (lub przeglądanie pamięci podręcznej). Końcowe zamknięcie pliku przez klienta również będzie tylko operacją lokalną.

Różnica między obsługą doglądaną a niedoglądaną stanie się ewidentna, jeśli rozważymy efekt awarii występującej podczas wykonywania usługi. Wskutek awarii usługa doglądana traci cały swój ulotny stan. Zapewnienie zgrabnej reaktywacji takiego serwera wymaga odtworzenia tego stanu – zwykle za pomocą protokołu rekonstrukcji opartego na dialogu z klientem. Mniej

gladkie wznowienie działania polega na zaniechaniu operacji, które były wykonywane w chwili awarii. Inny problem wynika z awarii klienta. Serwer musi wiedzieć o takich awariach, aby móc oddać do wtórnego użytku obszar pamięci przydzielony na zapamiętanie stanu procesów uszkodzonych klientów. To postępowanie jest czasami nazywane *wykrywaniem i eliminacją sieciot* (ang. *orphan detection and elimination*).

W przypadku serwera bezstanowego unika się tych problemów, ponieważ reaktywowany serwer może odpowiadać na samowystarczalne zamówienia bez żadnych trudności. Toteż skutki awarii serwera i jego przywracania do działania są prawie niezauważalne. Proces klienta nie odróżnia powolnego serwera od serwera rekonstruowanego. Klient powtarza zamówienia, jeśli nie otrzymuje odpowiedzi.

Za odporną, niedoglądaną obsługę płaci się dłuższymi komunikatami z zamówieniami i wolniejszym przetwarzaniem zamówień, ponieważ w pamięci operacyjnej nie ma żadnych informacji, które mogłyby je przyspieszyć. Ponadto obsługa niedoglądana nakłada dodatkowe ograniczenia na konstrukcję rozproszonego systemu plików. Po pierwsze, każde zamówienie określa plik docelowy, zatem w obrębie całego systemu musi być zastosowany jednolity schemat nazewniczy na niskim poziomie systemu. Tłumaczenie w każdym zamówieniu nazw zdalnych na lokalne powoduje dalsze spowolnienie realizacji zamówień. Po drugie, ponieważ klienci ponawiają wysyłanie zamówień na operacje plikowe, więc operacje te muszą być *idempotentne*, tzn. każda operacja musi powodować ten sam skutek i zwracać ten sam wynik nawet wtedy, gdy jest wykonywana kilka razy z rzędu. Samowystarczalne operacje czytania i pisania są dopóty *idempotentne*, dopóki używają bezwzględnego licznika bajtów do wskazania miejsca w pliku i nie posługują się adresowaniem przyrostowym (co występuje w uniksowych operacjach *read* i *write*). Ostrożność należy zachować przy implementowaniu operacji destrukcyjnych (takich jak usuwanie pliku), aby również i one były *idempotentne*.

W pewnych środowiskach obsługa doglądana jest koniecznością. Jeśli serwer stosuje inicjonowaną przez siebie metodę sprawdzania, czy zawartość pamięci podrzcznej jest aktualna, to jego działanie nie może przybrać formy obsługi niedoglądanej, ponieważ serwer utrzymuje spis aktualnie używanych plików i ich klientów.

Ze sposobu, w jaki system UNIX używa deskryptorów plików i niejawnych adresów względnych, wynika konieczność przechowywania stanu. Serwery muszą utrzymywać tablice do odwzorowywania deskryptorów plików na i-węzły oraz przechowywać bieżący adres względny w pliku. To właśnie powoduje, że w systemie NFS, w którym zastosowano serwery bezstanowe, nie używa się deskryptorów plików i wymaga się załączania jawnego określenia odległości w pliku przy każdym dostępie.

17.5 ■ Zwielokrotnianie pliku

Powielanie plików w różnych maszynach tworzy użyteczną nadmiarowość polepszającą dostępność. *Zwielokrotnianie* (ang. *replication*) informacji w wielu maszynach może mieć także korzystny wpływ na wydajność, gdyż wybór najbliższej kopii skraca czas obsługi zamówienia.

Podstawowym nakazem schematu zwielokrotniania jest umieszczanie różnych replik*, czyli kopii tego samego pliku, w maszynach, które są od siebie niezależne w wypadku awarii. Oznacza to, że na dostępność jednej kopii nie ma wpływu dostępność pozostałych kopii. Ten oczywisty wymóg implikuje, że zarządzanie zwielokrotnianiem z natury ukrywa lokalizację. Muszą istnieć możliwości umieszczania kopii na poszczególnych maszynach.

Jest wskazane, aby szczegóły zwielokrotniania były ukryte przed użytkownikami. Odwzorowywanie zwielokrotnionego pliku na jego poszczególne kopie musi być przewidziane w schemacie nazewnictwym. Istnienie kopii nie powinno być widoczne na wyższych poziomach systemu. Jednakże kopie powinny być odróżnialne za pomocą różnych nazw na niższym poziomie. Inną kwestią związaną z przezroczystością jest umożliwienie sterowania zwielokrotnianiem na wyższych poziomach. Sterowanie zwielokrotnianiem obejmuje określanie stopnia zwielokrotnienia i dozór nad rozmieszczeniem kopii. W pewnych warunkach jest pożądane ukazanie tych szczegółów użytkownikom. Na przykład system Locus dostarcza użytkownikom i administratorom systemu mechanizmów kontroli schematu zwielokrotniania.

Podstawowym problemem dotyczącym kopii jest ich uaktualnianie. Z punktu widzenia użytkownika kopie pliku reprezentują tę samą jednostkę logiczną, zatem uaktualnienie dowolnej kopii musi znaleźć odzwierciedlenie we wszystkich pozostałych kopach. Dokładniej możemy to wyrazić w ten sposób, że musi być zachowana niezbędną spójność semantyki wtedy, kiedy dostępy do kopii są traktowane jako wirtualne dostępy do plików logicznych kopii. Jeśli spójność nie ma zasadniczej wagi, to można ją poświęcić na rzecz dostępności i wydajności. Konieczność tego wyboru uwiadcznia podstawowy dylemat z obszaru tolerowania uszkodzeń. Trzeba dokonać wyboru między utrzymywaniem spójności za wszelką cenę, co tworzy potencjalne niebezpieczeństwo nieskończonego blokowania, a poświęceniem spójności w warunkach pewnych (należy mieć nadzieję, że rzadkich) katastrofalnych awarii na rzecz zagwarantowania postępu. Pośród systemów objętych naszym przeglądem, w Locusie stosuje się zwielokrotnianie w szerokim zakresie i poświęca spójność w przypadkach podziału sieci na rzecz dostępności plików zarówno do czytania, jak i do pisania (zob. szczegóły w p. 17.6.5).

* Dalej używamy po prostu terminu „kopii”. – Przyp. tłum.

Jako ilustrację tych koncepcji opiszemy schemat zwielokrotniania stosowany w systemie Ibis, w którym występuje pewna odmiana metody kopii podstawowej. Dziedzinę odwzorowywania nazw tworzy para <*identyfikator-kopii-podstawowej, identyfikator-kopii-lokalnej*>. Gdy nie ma kopii lokalnych, co może się zdarzyć, wówczas stosuje się specjalną wartość. Tak więc odwzorowanie to odnosi się do maszyny. Jeśli kopia lokalna jest zarazem podstawową, to para zawiera dwa identyczne identyfikatory. System Ibis umożliwia zwielokrotnianie na żądanie, będące metodą automatycznego sterowania zwielokrotnianiem (podobną do podręcznego przechowywania całych plików). Zwielokrotnianie na żądanie oznacza, że czytanie nielokalnej kopii powoduje umieszczenie jej w lokalnej pamięci podręcznej. Jest to równoznaczne z utworzeniem nowej, niepodstawowej kopii. Uaktualnia się tylko kopię podstawową, czemu towarzyszy wysłanie do wszystkich innych kopii komunikatu powodującego unieważnienie tych kopii. Nie gwarantuje się unieważniania wszystkich kopii niepodstawowych w sposób niepodzielny i szeregowalny. Zatem może się zdarzyć, że przeterminowana kopia będzie uznana za poprawną. Aby umożliwić zdalne dostępy do pisania, wysyła się kopię podstawową do zamawiającej ją maszyny.

17.6 ■ Przykłady systemów

W tym punkcie zilustrujemy wspólne koncepcje leżące u podstaw rozproszonych systemów plików, analizując pięć różnych i godnych uwagi systemów: UNIX United, NFS, Andrew, Sprite i Locus.

17.6.1 UNIX United

Projekt UNIX United, opracowany w University of Newcastle nad Tyne w Anglii, jest jedną z najwcześniejszych prób powiększenia uniksowego systemu plików do systemu rozproszonego, bez modyfikowania jądra systemu UNIX. W systemie UNIX United do każdego z połączonych ze sobą systemów UNIX (zwanych systemami *składowymi* lub *tworzącymi*) dodano podsystem oprogramowania, tak aby skonstruować system rozproszony, funkcjonalnie nieodróżnialny od zwykłego, skcentralizowanego systemu UNIX. System UNIX United opisujemy na dwu poziomach szczegółowości. Najpierw dokonamy przeglądu tego systemu. Następnie omówimy implementację warstwy Newcastle Connection i niektóre kwestie sieciowe i międzsieciowe.

17.6.1.1 Przegląd

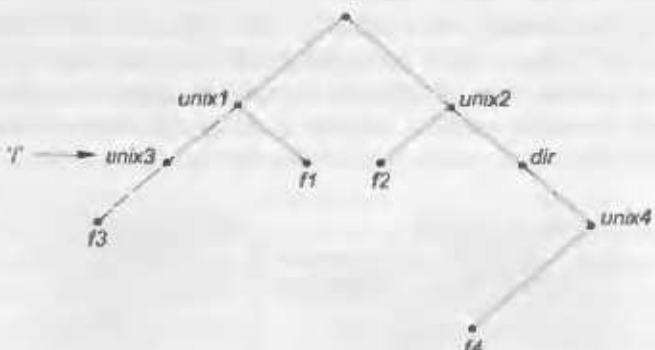
Aby utworzyć system UNIX United, można połączyć ze sobą dowolną liczbę systemów UNIX. Ich struktury nazw (pliki, urządzenia, katalogi i polecenia)

są łączone razem w jedną strukturę nazewniczą, w której każdy system składowy jest pod każdym względem traktowany jak katalog. Pomijając na razie kwestie pełnomocnictw i kontroli dostępu, w wynikowym systemie każdy użytkownik może czytać i pisać dowolny plik, używać dowolnego urządzenia, wykonywać dowolne polecenie i przeglądać katalogi niezależnie od systemu, do którego one należą.

Jednostka składowa jest pełnym drzewem katalogów systemu UNIX, należącym do pewnej maszyny. Pozycja jednostek składowych w hierarchii nazewniczej jest dowolna. Mogą się one pojawiać w strukturze nazewniczej na pozycjach podległych innym jednostkom składowym (bezpośrednio lub poprzez pośrednie katalogi).

Korzenie jednostek składowych mają przypisane nazwy, tak aby mogły być dostępne i rozróżnialne z zewnątrz. Korzeń dowolnego systemu plików, niezmiennie oznaczany jako „/”, jest początkiem nazw wszystkich ścieżek rozpoczynających się od znaku /. Jednak podległy system plików może się odwoływać do systemu zwierzchniego za pomocą oznaczenia korzenia swojego przodka (tj. /.). Zatem istnieje tylko jeden korzeń, który jest przodkiem samego siebie i który nie ma przypisanej nazwy w postaci ciągu znaków, mianowicie – korzeń złożonej struktury nazw; jest to węzeł wirtualny, potrzebny po to, aby z całej struktury utworzyć pojedyncze drzewo. W tej konwencji nie istnieje pojęcie bezwzględnej nazwy ścieżki. Każda nazwa ścieżki jest względna w pewnym kontekście – względem bieżącego katalogu roboczego albo bieżącej jednostki składowej.

Na rysunku 17.1 napisy *unix1*, *unix2*, *unix3* i *unix4* są nazwami systemów składowych. Aby zilustrować względne nazwy ścieżek, zauważmy, że we wnętrzu systemu *unix1* plik *f2* z systemu *unix2* jest określany jako *././unix2/f2*; w obrębie systemu *unix3* plik ten jest określany jako *./././unix2/f2*. Założymy teraz, że bieżącym korzeniem (/) jest ten, na który wskazuje strzałka. Wów-



Rys. 17.1 Przykład struktury katalogowej systemu UNIX United

czas do pliku *f3* można się odnieść przez nazwę *f3*, plik *f1* jest osiągany za pomocą nazwy *./f1*, plik *f2* – przez nazwę *././unix2/f2* i wreszcie plik *f4* – za pośrednictwem nazwy *././unix2/dir/unix4/f4*.

Zauważmy, że użytkownicy są świadomi górnych ograniczeń ich bieżącej jednostki składowej, gdyż muszą używać składni „*./*”, ilekroć chcą przejść w górę, poza ich bieżącą maszynę. Toteż UNIX United nie zapewnia pełnej przezroczystości położenia.

Tradycyjne katalogi korzeniowe (np. */dev*, */temp*) są utrzymywane dla każdej maszyny oddzielnie. Dzięki względności schematu nazewniczego są one nazwane w obrębie systemów składowych dokładnie tak samo jak w zwykłym systemie UNIX.

Każdy system składowy ma własny zbiór użytkowników i własnego administratora (superużytkownika). Ten ostatni jest odpowiedzialny za akredytację użytkowników w jego systemie, a także za dołączanie użytkowników zdalnych. Identyfikator użytkownika zdalnego jest, dla zachowania jednoznaczności, poprzedzony przedrostkiem z nazwą systemu, z którego dany użytkownik pochodzi. Dostępami rządzą standardowe mechanizmy ochrony plików systemu UNIX – nawet wówczas, gdy dostępy przekraczają granice składowych. Tak więc nie ma potrzeby, aby użytkownicy dokonywali specjalnych rejestracji lub podawali hasła przy dostępie do zdalnych plików. Jednak użytkownicy, którzy chcą korzystać z plików w systemach zdalnych, muszą to z osobna uzgodnić z administratorami poszczególnych systemów.

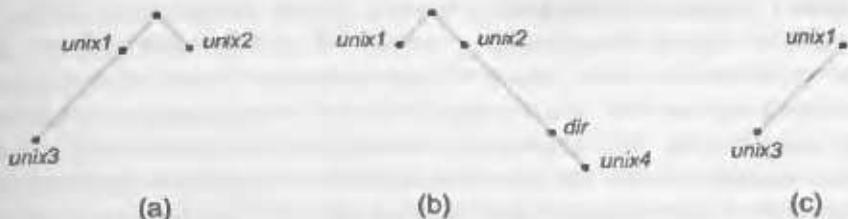
Strukturę nazewniczą często wygodnie jest tak ukształtować, aby odzwierciedlała hierarchię organizacyjną środowiska, w którym system istnieje.

17.6.1.2 Warstwa Newcastle Connection

Newcastle Connection jest warstwą oprogramowania (poziomu użytkownika) wielonową do każdego systemu składowego. Jest to warstwa połączeń, która występuje między jądrem systemu UNIX a programami poziomu użytkownego (rys. 17.2). Przechwytuje ona wszystkie odwołania do systemu dotyczące plików i odfiltrowuje te, które mają być skierowane do systemów zdalnych. Warstwa połączeń przejmuje również wywołania systemowe skierowane do niej z innych systemów. Zdalne warstwy komunikują się ze sobą za pomocą protokołu RPC (zdalnego wywoływanego procedur).



Rys. 17.2 Schemat architektury systemu UNIX United



Rys. 17.3 Systemy plików: (a) *unix1*, (b) *unix2*, (c) *unix3*

Warstwa połączeń zachowuje ten sam interfejs odwołań do systemu UNIX, który występuje w jądrze tego systemu, mimo wykonywanych przez system w szerokim zakresie operacji zdalnych. Ceną za pozostawienie nienikietego jądra jest implementacja usług w formie procesów-demonów działających na poziomie użytkownika, co spowalnia operacje zdalne.

Każda warstwa połączeń* przechowuje częściowy szkielet całej struktury nazewniczej. Oczywiście, każdy system przechowuje lokalnie własny system plików. Ponadto każdy system przechowuje fragmenty całej struktury nazewniczej odnoszące się do systemów sąsiadujących z nim w tej strukturze (tj. systemów, do których można dotrzeć w drzewie nazw bez przechodzenia przez inne systemy). Na rysunku 17.3 są pokazane częściowe szkielety hierarchii systemu plików z rys. 17.1, utrzymywane odpowiednio przez systemy *unix1*, *unix2* i *unix3* (pokazano tylko niezbędne części).

Fragmenty struktury utrzymywane przez różne systemy zachodzą na siebie, muszą więc pozostawać spójne. Ten wymóg powoduje, że zmiany całej struktury należą do rzadkości. Niektóre przechowywane lokalnie liście struktur częściowych odpowiadają zdalnym korzeniom innych części globalnego systemu plików. Liście te są specjalnie oznakowane i zawierają adresy odpowiednich stanowisk pamięci podległych systemów plików. Po napotkaniu takiego oznakowanego listu przejścia wzduż nazw ścieżek muszą być kontynuowane zdalnie i w rzeczywistości mogą wieść poprzez kilka systemów, zanim zostanie zlokalizowany plik docelowy. Po dotarciu do pliku według nazwy i po jego otwarciu następne dostępy do niego odbywają się za pomocą deskryptorów pliku. Warstwa połączeń oznacza deskryptory odnoszące się do zdalnych plików oraz przechowuje adresy sieciowe plików i informacje o prowadzących do nich trasach w należących do poszczególnych procesów tablicach.

Rzeczywiste dostępy zdalne do pliku są wykonywane za pomocą zbioru procesów usługowych pracujących w systemie docelowym. Każdy klient ma własny proces serwera plików, z którym komunikuje się bezpośrednio. Początkowe połączenie jest ustalone z pomocą *procesu rozmawiającego* (ang.

* To znaczy warstwa połączeń w każdym z systemów składowych. – Przyp. tłum.

spawner)¹, który ma standardową, ustaloną nazwę, dzięki czemu można go wywołać z każdego zewnętrznego procesu. Ów proces rozmnazający wykonuje sprawdzenie zdalnych praw dostępu stosownie do pary identyfikującej maszynę i użytkownika. On również zamienia tę identyfikację na odpowiednią nazwę lokalną. Aby semantyka systemu UNIX została zachowana, gdy proces klienta rozwidla się, wówczas rozwidleniu ulega także proces obsługujący plik. Ten schemat nie jest zbyt doskonały pod względem odporności. W przypadku jednoczesnej awarii serwera i klienta muszą być podejmowane specjalne działania ratunkowe. Jednakże warstwa połączeń stara się maskować i izolować błędy wynikające z tego, że system jest rozproszony.

17.6.2 Sieciowy system plików komputerów Sun

Sieciowy system plików komputerów Sun (ang. *Network File System* – NFS) jest zarówno implementacją, jak i specyfikacją systemu oprogramowania umożliwiającego dostęp do plików zdalnych w sieciach lokalnych (lub nawet w sieciach rozległych). System plików NFS jest częścią pakietu ONC+, dostarczanego² przez większość dostawców systemu UNIX. Implementacja jest częścią systemu operacyjnego Solaris, który jest modyfikowaną wersją systemu UNIX SVR4 pracującego na stacjach roboczych Sun i innym sprzęcie. W systemie tym używa się zawodnego protokołu datagramowego (UDP/IP) i sieci Ethernet (lub innego systemu sieciowego). Specyfikacja i implementacja przeplatają się ze sobą w naszym opisie systemu NFS. Ilekroć jest wymagany poziom szczegółowy, odwołujemy się do implementacji systemu dla komputerów Sun: wszędzie tam, gdzie opis jest wystarczająco ogólny, dotyczy on również specyfikacji.

17.6.2.1 Przegląd

W systemie NFS zbiór połączonych ze sobą stacji roboczych traktuje się jak zbiór niezależnych maszyn z niezależnymi systemami plików. Celem systemu NFS jest umożliwienie w pewnym stopniu dzielenia zasobów między tymi systemami plików (dokonywanego na jawne zamówienia) w sposób przejrzysty. Dzielenie odbywa się na zasadzie powiązań klient-serwer. Maszyna może być, i często jest, zarówno klientem, jak i serwerem. Dzielenie zasobów może dotyczyć dowolnych dwóch maszyn – nie muszą to być wyłącznie maszyny wyznaczone do pełnienia funkcji serwerów. W celu zapewnienia niezależności maszyn dzielenie zdalnego systemu plików odnosi się tylko do maszyny klienta i nie dotyczy żadnej innej.

¹ Dosłownie: *ikrzak*; tu: generator procesów obsługi. – Przyp. tłum.

² Por. p. 17.2.2. – Przyp. tłum.

Aby zdalny katalog stał się w przezroczysty sposób dostępny z konkretnej maszyny, powiedzmy – *M1*, klient tej maszyny musi najpierw wykonać operację montowania. Semantyka tej operacji jest następująca: katalog zdalny montuje się na miejsce katalogu lokalnego systemu plików. Po zakończeniu operacji montowania zamontowany katalog wygląda jak integralne poddrzewo lokalnego systemu plików, zastępujące poddrzewo wychodzące z lokalnego katalogu. Katalog lokalny staje się nazwą korzenia nowo zamontowanego katalogu. Określenie zdalnego katalogu jako argumentu w operacji montowania wykonuje się w sposób nieprzezroczysty. Musi być podane położenie (tj. nazwa komputera w sieci) katalogu zdalnego. Wszakże po dokonaniu tego użytkownicy maszyny *M1* mogą mieć dostęp do plików w zdalnym katalogu w sposób zupełnie przezroczysty.

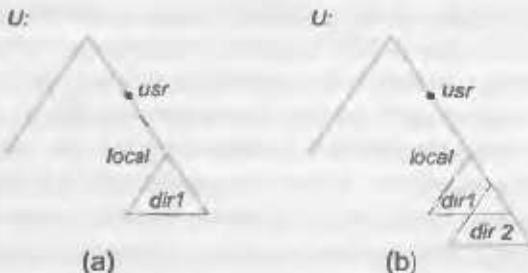
W celu zilustrowania montowania pliku rozważymy system plików przedstawiony na rys. 17.4, na którym trójkąty reprezentują interesujące nas poddrzewa katalogów. Na rysunku pokazano trzy niezależne systemy plików maszyn nazwanych *U*, *S1* i *S2*. W przedstawionej sytuacji w każdej z maszyn są dostępne tylko pliki lokalne. Na rysunku 17.5(a) pokazano efekt zamontowania katalogu *S1:/usr/shared* w katalogu *U:/usr/local*. Rysunek ten ukazuje, jak widzą swój system plików użytkownicy maszyny *U*. Zauważmy, że mają oni dostęp do dowolnego pliku w katalogu *dir1*, używając na przykład przedrostka */usr/local/dir1* na maszynie *U*, oczywiście po zakończeniu montowania. Oryginalna zawartość katalogu */usr/local* na tej maszynie przestaje być widoczna.

Uwzględniając udzielanie stosownych praw dostępu, dowolny system plików, lub występujący w nim katalog, można zamontować zdalnie u wierzchołka dowolnego lokalnego katalogu. Bez dyskowe stacje robocze mogą z serwerów montować nawet swoje katalogi korzeniowe.

Dopuszcza się montowanie kaskadowe. Należy przez to rozumieć, że system plików może być zamontowany w innym systemie plików, który nie jest lokalny, lecz zamontowany zdalnie. Jednak maszyny dotyczą tylko te montaże, które zostały wywołane przez nią samą.



Rys. 17.4 Trzy niezależne systemy plików



Rys. 17.5 Operacja montowania w systemie NFS. (a) zamontowanie, (b) montowanie kaskadowe

Przez zamontowanie zdalnego systemu plików klient nie zyskuje dostępu do innych systemów plików, które ewentualnie były zamontowane w danym systemie poprzednio. Zatem mechanizm montowania nie wykazuje cechy przechodniości. Na rysunku 17.5(b) jest pokazany montaż kaskadowy, będący kontynuacją poprzedniego przykładu. Na rysunku widać wynik zamontowania katalogu $S2:/dir2/dir3$ w katalogu $U:/usr/local/dir1$, który jest już zdalnie zamontowany z maszyny $S1$. Użytkownicy mogą sięgać do plików w katalogu $dir3$ na maszynie U , stosując przedrostek $/usr/local/dir1$. Jeśli dzielony system plików jest zamontowany w katalogach prywatnych użytkownika na wszystkich maszynach w sieci, to użytkownik może rejestrować się na każdej stacji roboczej i dysponować własnym, prywatnym środowiskiem. Cechą ta jest nazywana *mobilnością użytkownika* (ang. *user mobility*).

Jednym z założeń systemu NFS było działanie w heterogenicznym środowisku różnych maszyn, systemów operacyjnych i architektur sieciowych. Specyfikacja systemu NFS jest niezależna od tych mediów, co sprzyja różnym implementacjom. Niezależność ta została osiągnięta dzięki użyciu elementarnych wywołań RPC zbudowanych powyżej protokołu zewnętrznej reprezentacji danych (ang. *External Data Representation – XDR*) używanego między dwoma interfejsami niezależnymi od implementacji. Jeśli więc system składa się z heterogenicznych maszyn i systemów plików właściwie przyłączonych do systemu NFS, to jest możliwe zarówno lokalne, jak i zdalne montowanie systemów plików różnych typów.

W specyfikacji systemu NFS odróżniono usługi świadczone przez mechanizm montażu od rzeczywistej obsługi zdalnego dostępu do plików. Stosownie do tego są zdefiniowane dwa osobne protokoły: *protokół montowania* (ang. *mount protocol*) i protokół dostępu zdalnego, zwany protokołem NFS. Protokoły te są zdefiniowane jako zbiory wywołań procedur zdalnych (RPC) – swoistych klocków, z których można budować przezroczysty, zdalny dostęp do plików.

17.6.2.2 Protokół montowania

Protokół montowania znajduje zastosowanie przy nawiązywaniu wstępnego, logicznego połączenia między serwerem a klientem. W implementacji dla komputerów Sun każda maszyna ma na zewnątrz jądra proces usługowy (serwer), który wypełnia funkcje protokołu.

Operacja montowania zawiera nazwę katalogu zdalnego, który ma być zamontowany, i nazwę przechowującej go maszyny usługodawczej. Zamówienie montażu zostaje odwzorowane na odpowiednie wywołanie procedury zdalnej i przekazane do serwera montażu działającego w konkretnej maszynie usługodawczej. Serwer ten utrzymuje *listę eksportową* (w systemie UNIX jest nią plik */etc/exports* redagowany tylko przez superużytkownika), określającą lokalne systemy plików udostępniane przez serwer do zamontowania, oraz nazwy maszyn, na których montaż tych systemów jest dozwolony. Lista ta ma, niestety, ustaloną z góry długość, co ogranicza skalowalność systemu NFS. Przypomnijmy, że dowolny katalog w obrębie eksportowanego systemu plików może zostać zamontowany zdalnie przez upoważnioną maszynę. Zatem katalog taki jest jednostką składową (zob. p. 17.1). Gdy serwer otrzyma zamówienie montażu zgadzające się z jego listą eksportową, wówczas zwraca klientowi *uchwyt plikowy*, służący jako klucz do dalszych dostępów do plików w zamontowanym systemie plików. Uchwyty plikowe zawierają wszystkie informacje niezbędne do rozróżnienia przez serwer poszczególnych, przechowywanych w nim plików. W kategoriach systemu UNIX uchwyty plikowe składają się z identyfikatora systemu plików oraz z numeru i-węzła dokładnie identyfikującego zamontowany katalog wewnętrz eksportowanego systemu plików.

Serwer utrzymuje również wykaz maszyn klientów i odpowiednich, aktualnie zamontowanych katalogów. Wykaz ten służy głównie do celów administracyjnych – na przykład do powiadamiania wszystkich klientów, że praca serwera dobija końca. Dodanie lub usunięcie pozycji w tym wykazie jest jedną sytuacją, w której protokół montowania może oddziaływać na stan serwera.

System ma zazwyczaj statyczną, wstępna konfigurację montażu, określającą podczas jego rozruchu (plik */etc/fstab* w systemie UNIX), niemniej stan ten może ulec zmianie. Protokół montowania zawiera oprócz właściwej procedury montowania kilka innych procedur, takich jak procedura demontowania i procedura przekazania wykazu eksportowego.

17.6.2.3 Protokół NFS

Protokół NFS dostarcza zbioru wywołań procedur zdalnych do obsługi zdalnych operacji plikowych. Procedury te umożliwiają następujące działania:

- szukanie pliku w obrębie katalogu;
- czytanie zbioru wpisów katalogowych;

- manipulowanie dowiązaniem i katalogami;
- dostęp do atrybutów pliku;
- czytanie i pisanie plików.

Procedury można wywoływać dopiero po uzyskaniu uchwytu do zdalnie zamontowanego katalogu.

Ominięcie operacji otwarcia i zamknięcia jest zamierzone. Wazną cechą serwerów NFS jest to, że są *bezstanowe*. Serwery nie utrzymują informacji o klientach między jednym dostępem a drugim. Po stronie serwera nic ma odpowiedników uniksowych tablic otwartych plików lub struktur plików. Wskutek tego każda zamawiana operacja musi zawierać pełny zbiór argumentów, włącznie z jednoznacznym identyfikatorem pliku i bezwzględnym określeniem miejsca wewnątrz pliku. Wynikowa konstrukcja jest niewrażliwa na uszkodzenia – przy przywracaniu serwera do pracy po awarii nie trzeba podejmować żadnych specjalnych kroków. W tym celu jednak operacje plikowe muszą być *idempotentne*.

Wspomniane uprzednio utrzymywanie wykazu klientów zdaje się naruszać bezstanowość serwera. Jednak wykaż ten, jako nieistotny dla poprawnego działania klienta lub serwera, nie musi być odwzorowany po awarii serwera. Może więc zawierać niespójne dane i powinien być traktowany tylko jako pomocniczy.

Kolejną implikacją koncepcji serwera bezstanowego oraz wynikiem synchroniczności zdalnych wywołań procedur jest to, że zmienione dane (w tym bloki pośrednie i bloki stanu) muszą być przed przekazaniem ich do klienta zatwierdzone na dysku serwera. Klient może zatem przechowywać podręczne zapisywane przez siebie bloki, a kiedy odsyła je do serwera, wtedy zakłada, że docierają one na dysk serwera. Tym samym awarie serwera i usuwanie ich skutków są niewidoczne dla klienta; wszystkie bloki utrzymywane przez serwer dla klienta pozostają nietknięte. Ponoszone z tego powodu straty wydajności mogą być znaczne, gdyż traci się zalety płynące z przechowywania podręcznego. Obecnie na rynku znajduje się kilka produktów, w których na te niedostatki protokołu NFS zwraca się szczególną uwagę i zaopatruje systemy w szybkie pamięci trwałe (na ogół jest to pamięć z awaryjnymi bateriami zasilającymi), w których można przechowywać bloki zapisywane przez system NFS. Bloki takie pozostają nienaruszone nawet po załamaniu systemu. Okresowo przepisuje się je z pamięci trwalej na dysk.

Pojedynczemu wywołaniu procedury pisania gwarantuje się w systemie NFS wykonanie niepodzielne i nie zmieszane ze skutkami innych operacji pisania odnoszących się do tego samego pliku. Jednak protokół NFS nie dostarcza mechanizmów sterowania współbieżnością. Systemowa funkcja `write`

może być podzielona na kilka zdalnych operacji pisania, ponieważ każda operacja pisania lub czytania systemu NFS może zawierać najwyżej do 8 KB danych, a pakiety UDP są ograniczone do 1500 B. Wskutek tego dwaj użytkownicy piszący do tego samego pliku zdalnego mogą otrzymać dane przemieszane. Wynika z tego wniosek, że zarządzanie blokowaniem plików* – jako czynność z natury doglądana – powinno być usługą spoza protokołu NFS (i tak postąpiono w systemie Solaris). Użytkownikom doradza się, aby koordynowali dostęp do plików dzielonych, stosując w tym celu mechanizmy spoza obszaru systemu NFS.

17.6.2.4 Architektura systemu NFS

System NFS składa się z trzech głównych warstw, co schematycznie jest pokazane na rys. 17.6. Pierwszą warstwą jest interfejs systemu plików UNIX oparty na systemowych funkcjach otwierania, czytania, pisania i zamknięcia plików oraz na deskryptorach plików.

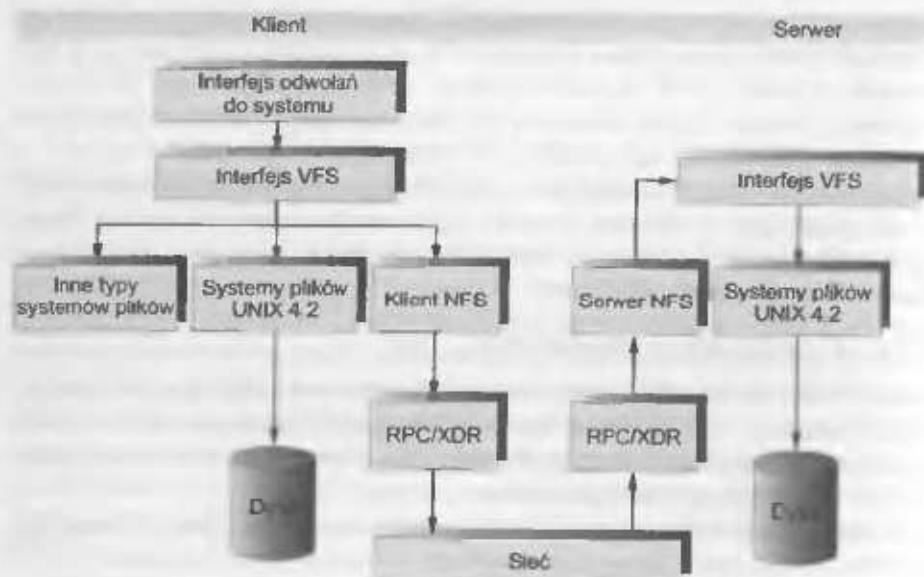
Druga warstwa nosi nazwę *wirtualnego systemu plików* (ang. *Virtual File System* – VFS) i pełni dwie ważne funkcje:

- Przez zdefiniowanie czystego interfejsu wirtualnego systemu plików (VFS) oddziela ogólne operacje plikowe od ich implementacji. W tej samej maszynie może koegzystować kilka implementacji interfejsu VFS, co umożliwia przezroczysty dostęp do lokalnie zamontowanych systemów plików różnego typu.
- Warstwa VFS opiera się na strukturze reprezentacji pliku zwanej *v-węzlem* (ang. *vnode*), zawierającej jednoznaczny w obrębie całej sieci liczbowy oznacznik pliku. (Przypomnijmy, że i-węzły w systemie UNIX są niepowtarzalne tylko w obrębie jednego systemu plików). Jądro utrzymuje po każdej strukturze v-węzła dla każdego aktywnego węzła (pliku lub katalogu).

Zatem warstwa VFS odróżnia pliki lokalne od zdalnych, przy czym pliki lokalne są dalej różnicowane zgodnie z typami ich systemów plików.

Podobnie jak w standardowym systemie UNIX, jądro utrzymuje tablicę (w systemie UNIX – */etc/mtab*) przechowującą szczegóły montaży, w których wzięło udział jako klient. Ponadto w pamięci są przez cały czas przechowywane i oznakowane v-węzły dla każdego zamontowanego katalogu, tak że zamówienia dotyczące takich katalogów są kierowane przez tablicę montaży do odpowiedniego, zamontowanego systemu plików. Struktury v-węzłów,

* Chodzi tu o okresowe zajmowanie plików na zasadzie wyłączności (ang. *lock management*) – Przyp. tłum.



Rys. 17.6 Schemat architektury systemu NFS

uzupełniane przez tablicę montaży, dostarczają w istocie dla każdego pliku wskaźnik do jego macierzystego systemu plików, a także wskaźnik do systemu plików, w którym ten system plików jest zamontowany.

Warstwa VFS uaktywnia specyficzne dla systemu plików operacje realizowania lokalnych zamówień, właściwe do typu systemu plików, oraz wywołuje procedury protokołu NFS dla zamówień zdalnych. Uchwyty plikowe są konstruowane na podstawie odpowiednich v-węzłów i przekazywane jako argumenty tych procedur. Warstwa implementująca protokół NFS znajduje się na spodzie architektury systemu i nazywa się warstwą obsługi systemu NFS.

Aby zilustrować tę architekturę, prześledźmy wykonywanie operacji na już otwartym, zdalnym pliku (idąc za przykładem na rys. 17.6). Klient rozpoczyna operację za pomocą zwykłego odwołania do systemu. Warstwa systemu operacyjnego odwzorowuje to odwołanie na operację systemu VFS na odpowiednim v-węźle. Warstwa VFS rozpoznaje plik jako zdalny i wywołuje odpowiednią procedurę NFS. Następuje zdalne wywołanie procedury odnoszącej się do warstwy obsługi systemu NFS w zdalnym serwerze. Wywołanie to przenika do warstwy VFS w systemie zdalnym, który identyfikuje je jako lokalne i wywołuje odpowiednią operację systemu plików. Ta droga jest odzwierciedlana przy przekazywaniu wyniku. Zaletą takiej architektury jest identyczność klienta i serwera. Zatem maszyna może być klientem, serwerem albo jednym i drugim.

Rzeczywista obsługa na każdym serwerze jest wykonywana przez kilka procesów jądrowych, które tworzą czasową namiastkę mechanizmu procesów lekkich (wątków).

17.6.2.5 Tłumaczenie nazwy ścieżki

Tłumaczenie nazwy ścieżki odbywa się przez pojęcie ścieżki na nazwy składowe i wykonanie osobnego przeszukania NFS (ang. *lookup call*) dla każdej pary złożonej z nazwy składowej i v-węzła katalogu. Przy przechodzeniu punktu zamontowania poszukiwanie każdej składowej powoduje osobne zdalne wywołanie procedury w serwerze (rys. 17.7). Ten kosztowny obchód nazwy ścieżki jest nieodzowny, ponieważ każdy klient ma sobie właściwy układ logicznej przestrzeni nazw, określony przez wykonane przez niego montaż. Byłyby znacznie wydajniej przekazać serwerowi po napotkaniu punktu zamontowania nazwę ścieżki i otrzymać docelowy v-węzeł. W każdym punkcie może jednak istnieć inny punkt zamontowania, należący do danego klienta, o którym serwer bezstanowy nie ma żadnych informacji.

W celu przyspieszenia przeszukiwania w pamięci podręcznej nazw katalogów po stronie klienta są przechowywane v-węzły odpowiadające nazwom katalogów zdalnych. Korzystanie z pamięci podręcznej przyspiesza odwołania do plików mających taką samą początkową nazwę ścieżki. Jeśli atrybuty przekazane przez serwer nie pasują do atrybutów v-węzła przechowanego podręcznie, to wpis w pamięci podręcznej traci ważność.

Przypomnijmy, że w systemie NFS jest dozwolone zamontowanie zdalnego systemu plików w innym, już zamontowanym zdalnym systemie plików (montowanie kaskadowe). Jednak serwer nie może działać jako pośrednik między klientem a innym serwerem. Zamiast tego klient musi nawiązać z drugim serwerem bezpośrednie połączenie serwer-klient przez bezpośrednie zamontowanie wymaganego katalogu. Gdy klient ma kaskadę zamontowaną, wówczas w obchód nazwy ścieżki może być zaangażowany więcej niż jeden serwer. Niemniej jednak szukanie każdej składowej dokonuje się między



Rys. 17.7 Tłumaczenie nazw ścieżek

pierwotnym klientem i jakimś serwerem. Dlatego gdy jakiś klient przeszukuje katalog, w którym dany serwer zamontował system plików, wówczas zamiast zamontowanego katalogu klient widzi katalog leżący u podłożu zamontowanego katalogu.

17.6.2.6 Operacje zdalne

Z wyjątkiem otwierania i zamknięcia plików, między wywołaniami zwykłych operacji plikowych systemu UNIX a protokołem zdalnych wywołań procedur systemu NFS istnieje niemal pełna odpowiedniość. Zatem zdalna operacja plikowa może być tłumaczona bezpośrednio na odpowiednie wywołanie RPC. Pod względem koncepcji system NFS przystaje do zasad obsługi zdalnej, lecz w praktyce ze względu na wydajność stosuje się buforowanie i pamięci podręczne. Nie ma prostej odpowiedniości między operacją zdalną i wywoływaniem zdalnej procedury. Przeciwnie – dostarczane przez zdalne wywołania procedur bloki i atrybuty plików są umieszczane w lokalnych pamięciach podręcznych. Późniejsze operacje zdalne używają tych danych z zachowaniem wymogów dotyczących spójności.

Istnieją dwie pamięci podręczne – na atrybuty plików (informacje z i-węzłów) oraz na bloki plików. Przy otwieraniu pliku jądro sprawdza z udziałem zdalnego serwera, czy należy sprowadzić atrybuty do pamięci podręcznej, czy też można uznać je za nadal aktualne. Pamięć podręczna atrybutów jest uaktualniana wraz z każdym nadaniem nowych atrybutów od serwera. Atrybuty plików są (domyślnie) usuwane z pamięci podręcznej po upływie 60 s. Między serwerem a klientem stosuje się zarówno technikę czytania z wyprzedzeniem, jak i technikę opóźnionego pisania. Klienci dopóty nie zwalniają zapisanych bloków, dopóki serwer nie upewni ich, że dane zostały zapisane na dysku. W przeciwnieństwie do systemu Sprite, opóźnione pisanie praktykuje się nawet wówczas, gdy plik został otwarty współbieżnie w kolidujących ze sobą trybach. Tak więc semantyka systemu UNIX nie jest zachowana.

Nastawienie systemu NFS na wydajność powoduje trudności w scharakteryzowaniu jego semantyki spójności. Utworzone na jednej maszynie nowe pliki mogą być gdzie indziej niewidoczne przez 30 s. Nie jest określone, czy pisanie do pliku na jednym stanowisku jest widoczne na innych stanowiskach, mających dany plik otwarty do czytania. Ponowne otwarcie takiego pliku spowoduje, że będą widziane tylko te zmiany, które były już posłane do serwera. System NFS nie emuluje zatem ściśle semantyki systemu UNIX ani semantyki sesji systemu Andrew. Pomimo tych wad użyteczność i duża wydajność mechanizmu NFS sprawia, że jest to najszerzej używany, rozpowszechniany przez wielu dostawców system rozproszony.

17.6.3 System Andrew

System Andrew jest rozproszonym środowiskiem obliczeniowym, nad którym pracowano w Carnegie-Mellon University od 1983 r. Tym samym jest to jeden z najnowszych rozproszonych systemów plików. System plików Andrew – AFS – tworzy podstawowy mechanizm dzielenia informacji między klientami środowiska. Handlowa implementacja systemu AFS, znana pod nazwą DFS (ang. *Distributed File System*), wchodzi w skład rozprozonego środowiska obliczeniowego DCE opracowanego przez konsorcjum OSF. Wielu dostawców systemów uniksowych, jak i pochodzących z firmy Microsoft, oznajmiło, że ich systemy zawierają możliwości wdrożenia systemu Andrew. Jedną z najbardziej znamienitych cech systemu Andrew jest jego skalowalność. System Andrew jest przeznaczony do obsługi ponad 5000 stacji roboczych.

17.6.3.1 Przegląd

W systemie Andrew odróżnia się *maszyny klientów* (czasami nazywane stacjami roboczymi) od wydzielonych *maszyn serwerów*. Serwery oraz klienci pracują pod nadzorem systemu operacyjnego UNIX 4.2BSD. Wszystkie maszyny są połączone za pomocą sieci złożonej z sieci lokalnych.

Przestrzenie nazw plików klientów mają po dwa obszary: *lokalną przestrzeń plików* i *dzieloną*, czyli *wspólną przestrzeń plików*. Dedykowane serwery – od nazwy wykonywanego przez nich oprogramowania nazywane łącznie *Vice* – prezentują klientom dzieloną przestrzeń nazw w postaci jednorodnej, identycznej i przezroczystej pod względem położenia hierarchii plików. Korzeniowy system plików stacji roboczej tworzy lokalną przestrzeń nazw, z której wywodzi się dzielona przestrzeń nazw. Stacje robocze wykonują protokół *Virtue*, za pomocą którego komunikują się z procesami Vice, przy czym muszą one mieć dyski lokalne, na których utrzymują swoje lokalne przestrzenie nazw. Serwery są wspólnie odpowiedzialne za przechowywanie i zarządzanie dzieloną przestrzenią nazw. Lokalna przestrzeń nazw jest mała, inna dla każdej stacji roboczej, i zawiera programy systemowe niezbędne do autonomicznej pracy i lepszej wydajności. Lokalne są również pliki istniejące okresowo oraz pliki, których lokalnego przechowywania właściciel stacji roboczej zaząda jawnie z przyczyn prywatnych.

Rozpatrując zagadnienie na większym poziomie szczegółowości, zauważymy, że klienci i serwery tworzą grupy (grona) połączone siecią rozległą. Każda grupa składa się ze zbioru stacji roboczych i przedstawiciela systemu Vice*, zwanego *serwerem grupy*, oraz jest połączona z siecią rozległą za po-

* Vice i Venus (zob. dalej) są procesami użytkowymi systemu Unix. Jednak w niniejszym opisie nazwą Vice obejmuje się także cały zespół oprogramowania usługodawczego. Z konieczności wynikającej z kontekstu będziemy więc czasem używać określenia „system Vice”. – Przyp. tłum.

mocą *rutera* (ang. *router*), czyli zarządcy tras. Podzielenie systemu na grupy służy przede wszystkim skalowalności. W celu optymalizowania wydajności stacje robocze powinny przede wszystkim korzystać z lokalnego serwera, aby odwołania między grupami występuły stosunkowo rzadko.

Kwestia skali wpłynęła również na architekturę systemu plików. Podstawowym założeniem było przerzucenie pracy z serwerów na klientów, ponieważ doświadczenia wykazały, że szybkość jednostki centralnej serwera jest wąskim gardłem systemu. Korzystając z tych doświadczeń, za kluczowy mechanizm działań na plikach zdalnych przyjęto podręczne przechowywanie dużych fragmentów plików (64 KB). Dzięki tej właściwości zmniejszono opóźnienia wynikające z otwierania plików i umożliwiono odnoszenie czytania i pisania do kopii przechowywanej podręcznie, bez angażowania serwerów.

Z systemem Andrew wiążą się zagadnienia, którymi nie będziemy się tu zajmować – ujmując w skrócie, są to:

- **Mobilność klienta:** Klienci mogą mieć dostęp z dowolnej stacji roboczej do każdego pliku w dzielonej przestrzeni nazw. Jednym skutkiem dostrzegalnym dla klientów przy dostępie do plików nie znajdujących się na ich stacjach roboczych jest pewne początkowe spowolnienie działania powodowane przesyaniem plików do pamięci podręcznej.
- **Bezpieczeństwo:** Interfejs Vice traktuje się jako granicę zaufania, ponieważ żaden z programów klientów nie działa na maszynach przeznaczonych do wykonywania oprogramowania Vice. Uwierzytelnianie i funkcje bezpiecznych transmisji są częścią działającego na zasadzie połączonowej pakietu komunikacyjnego, opartego na schemacie zdalnych wywołań procedur (RPC). Po wzajemnym sprawdzeniu tożsamości serwer Vice i klient komunikują się za pomocą zaszyfrowanych komunikatów. Szyfrowaniem zajmują się urządzenia sprzętowe lub dokonuje się go środkami programowymi (co działa wolniej). Informacje o klientach i grupach są przechowywane w bazie danych ochrony, powielanej w każdym serwerze.
- **Ochrona:** Do ochrony katalogów system Andrew stosuje wykazy dostępów, a do ochrony plików standardowe bity systemu UNIX. Wykaz dostępów może zawierać informacje o użytkownikach uprawnionych do dostępu do katalogu lub informacje o użytkownikach, którym dostęp taki jest zabroniony. Dzięki temu rozwiązaniu łatwo można określić, że każdy z wyjątkiem – dajmy na to – Kuby, ma prawo dostępu do katalogu. W systemie Andrew wyróżnia się takie rodzaje dostępu, jak czytanie, pisanie, przeszukiwanie, wstawianie, zarządzanie, blokowanie i usuwanie.

- **Heterogeniczność:** Zdefiniowanie przejrzystego interfejsu z systemem Vice jest środkiem do integracji stacji roboczych różnorodnych pod względem sprzętu i pracujących pod różnymi systemami operacyjnymi. W celu ułatwienia heterogeniczności niektóre pliki w lokalnym katalogu */bin* są dowiązaniem symbolicznym, wskazującym na rezydujące w systemie Vice pliki wykonywalne przeznaczone dla określonej maszyny.

17.6.3.2 Dzielona przestrzeń nazw

Dzielona przestrzeń nazw systemu Andrew składa się z jednostek składowych zwanych *tomami* (ang. *volumes*). Tomy systemu Andrew są niezwykle małymi jednostkami składowymi. Na ogół są nimi pliki jednego klienta. Kilka tomów rezyduje w obrębie jednej strefy dyskowej (ang. *disk partition*), ich rozmiar może zwiększać się (do pewnej granicy) lub maleć. Tomy są zespalane ze sobą przy użyciu mechanizmu podobnego do uniksowego montowania. Jest jednak istotna różnica w ziarnistości tych operacji, ponieważ w systemie UNIX zamontowaniu może podlegać tylko cała strefa dyskowa (zawierająca system plików). Tomy są podstawową jednostką administracyjną i odgrywają istotną rolę w identyfikowaniu i lokalizowaniu poszczególnych plików.

Katalog systemu Vice jest identyfikowany za pomocą niskopoziomowego identyfikatora, nazywanego *fid* (ang. *file identifier*). Każdy wpis katalogowy systemu Andrew odzworowuje składową nazwy ścieżki na identyfikator *fid*. Identyfikator *fid* ma 96 bitów i składa się z trzech części o jednakowej długości: *numeru tomu*, *numeru v-węzła* oraz *wyróżnika* (ang. *uniquifier*). Numer v-węzła jest używany jako indeks do tablicy zawierającej i-węzły plików pojedynczego tomu. Wyróżnik pozwala na ponowne używanie numerów v-węzłów, umożliwiając zachowanie zawartości pewnych struktur danych. Identyfikatory *fid* są przezroczyste pod względem polożenia, toteż przemieszczenia pliku między serwerami nie unieważniają zawartości katalogu przechowywanej w pamięci podręcznej.

Informacja o położeniu tomu jest przechowywana w *bazie danych o położeniu tomów* (ang. *volume location database*), której kopia znajduje się w każdym serwerze. Przez kierowanie zapytań do tej bazy klient może poznać umiejscowienie każdego tomu w systemie. Dzięki gromadzeniu plików w tomy bazę położen tomów można utrzymywać w rozmiarze zdatnym do zarządzania.

W celu równoważenia ilości dostępnej przestrzeni dyskowej i stopnia wykorzystania serwerów trzeba przemieszczać tomy między strefami dyskowymi i serwerami. Gdy tom jest wysyłany na nowe miejsce, wówczas w jego pierwotnym serwerze pozostawia się informację wyprzedzającą chwilowo stan docelowy, dzięki czemu baza położen nie musi być aktualniana synchronicznie.

nicznie. Podczas przekazywania tomu jego pierwotny serwer może nieprzerwanie przyjmować zamówienia na aktualizację tomu, które później prześle do nowego serwera. W pewnej chwili tom staje się na krótko niedostępny, aby można było wykonać w nim najnowsze zmiany, po czym przywraca się jego dostępność – już na nowym stanowisku. Operacja przemieszczenia tomu jest niepodzielna – jeśli którykolwiek z serwerów ulegnie awarii, to operacja będzie zaniechana.

Na poziomie całych tomów stosuje się zwielokrotnienia do czytania w odniesieniu do wykonywalnych plików systemowych i rzadko uaktualnianych plików górnych warstw przestrzeni nazw systemu Vice. Baza danych o położeniu tomów określa serwer zawierający jedyną kopię tomu przeznaczoną do czytania i pisania oraz wykaz stanowisk z kopiami przeznaczonymi tylko do czytania.

17.6.3.3 Działania na plikach i semantyka spójności

Fundamentalną zasadą architektoniczną w systemie Andrew jest podręczne przechowywanie całych plików pochodzących z serwerów. Zgodnie z tym, stacja robocza klienta współpracuje z serwerami Vice tylko podczas otwierania i zamykania plików, przy czym nawet wtedy nie zawsze jest to konieczne. Czytanie lub pisanie plików nie powoduje żadnej zdalnej współpracy (w przeciwieństwie do metody obsługi zdalnej). Ta zasadnicza różnica ma głęboki wpływ na wydajność systemu i semantykę operacji plikowych.

System operacyjny każdej stacji roboczej przyjmuje wywołania systemu plików i przekazuje je do procesu na poziomie klienta* w danej stacji. Proces ten, zwany *Venus*, przechowuje podręczne pliki nadchodzące od serwerów Vice w chwilach ich otwierania, a kiedy pliki są zamykane, wtedy odsyła ich zmienione kopie z powrotem do serwerów. Proces Venus może się kontaktować z oprogramowaniem Vice tylko przy otwieraniu lub zamykaniu pliku. Czytanie i pisanie poszczególnych bajtów pliku odbywa się z pominięciem procesu Venus i dotyczy bezpośrednio kopii przechowywanej w pamięci podręcznej. Wskutek tego wyniki operacji pisania na pewnych stanowiskach nie są natychmiast widoczne na innych stanowiskach.

Z pamięci podręcznych korzysta się przy późniejszym otwieraniu przechowywanych w nich plików. Proces Venus zakłada, że przechowywane w pamięciach podręcznych jednostki (pliki lub katalogi) są ważne dopóty, dopóki nie zostanie powiadomiony, że tak nie jest. Venus nie musi więc kontaktować się z serwerem Vice przy otwieraniu pliku, aby upewnić się, że kopia przechowywana w pamięci podręcznej jest aktualna. Mechanizm realizujący taką politykę, zwany *przywołaniem* (ang. *callback*), radykalnie zmniej-

* To znaczy na poziomie użytkowym. – Przyp. tłum.

sza liczbę zamówień na sprawdzenie ważności danych w pamięci podręcznej, otrzymywanych przez serwery. Działa on następująco. Gdy klient umieszcza plik lub katalog w pamięci podręcznej, wówczas serwer uaktualnia swoje dane, odnotowując to przechowanie. Mówimy, że klient ma w odniesieniu do tego pliku gwarancję przywołania. Serwer powiadamia klienta, zanim pozwoli na modyfikację pliku przez innego klienta. W takim przypadku mówimy, że serwer cofa gwarancję przywołania udzieloną klientowi w związku z tym plikiem*. Klient może otwierać przechowywany podręcznie plik tylko wtedy, kiedy plik jest objęty gwarancją przywołania. Jeśli klient zamknie plik po wykonaniu w nim zmiany, to wszyscy inni klienci przechowujący podręcznie dany plik tracą swoje gwarancje przywołania. Jeśli zatem tacy klienci będą otwierali dany plik później, to będą musieli pobrać jego nową wersję od serwera.

Czytanie i zapisywanie bajtów pliku jest wykonywane na kopii w pamięci podręcznej wprost przez jądro, bez udziału procesu Venus. Venus obejmuje ponownie kontrolę przy zamknięciu pliku i jeśli plik został lokalnie zmieniony, to uaktualnia go w odpowiednim serwerze. Zatem jedynymi okazjami, przy których Venus kontaktuje się z serwerami Vice, są operacje otwierania plików, których nie ma w pamięci podręcznej, lub takich, w odniesieniu do których cofnięto gwarancję przywołania, oraz operacje zamknięcia lokalnie zmienionych plików.

W systemie Andrew w zasadzie zrealizowano semantykę sesji. Jedynymi wyjątkami są operacje plikowe inne niż elementarne operacje czytania i pisania (takie jak wprowadzanie zmian w ochronie na poziomie katalogu), których skutki są widoczne w całej sieci natychmiast po zakończeniu operacji.

Pomimo mechanizmu przywoływania stale istnieje niewielki przepływ danych związany ze sprawdzaniem ważności informacji w pamięciach podręcznych. Wynika on zwykle z odwarzania gwarancji przywołań utraconych wskutek awarii maszyny lub sieci. Po wznowieniu pracy stacji roboczej proces Venus traktuje wszystkie podręcznie przechowane pliki i katalogi jako niepewne i przy pierwszym użyciu któregokolwiek z nich zamawia sprawdzenie jego aktualności.

Mechanizm przywołania zmusza wszystkie serwery do utrzymywania informacji o gwarancjach przywołań, a wszystkich klientów do utrzymywania informacji o ważności danych. Jeśli ilość przechowywanej przez serwer informacji o przywołaniach staje się nadmierna, to serwer może jednostronnie cofnąć gwarancje przywołań i odzyskać trochę pamięci, powiadamiając o tym klientów i unieważniając ich pliki w pamięciach podręcznych. Gdyby doszło

* Przejrzysty opis działania mechanizmu przywołania (zawiadomienia) można znaleźć w p. 8.3 książki G. Coulourisa i in. *Systemy rozproszone. Podstawy i projektowanie*, która ukazała się nakładem WNT w 1998 r. – Przyp. tłum.

do utraty synchronizacji stanu gwarancji przywołań utrzymywanego przez proces Venus i odpowiedniego stanu utrzymywanego przez serwery, to mogłyby nastąpić naruszenie spójności.

W celu tłumaczenia ścieżek system Venus przechowuje również w pamięciach podręcznych zawartości katalogów i dowiązania symboliczne. Pobiera się każdą składową nazwy ścieżki i jeśli nie ma jej jeszcze w pamięci podręcznej lub jeśli klient nie ma w odniesieniu do niej gwarancji przywołania, to udziela się takiej gwarancji. Przeszukiwania są wykonywane przez proces Venus lokalnie, na sprowadzonych katalogach, z użyciem identyfikatorów *fid*. Nie ma posyłania zamówień od jednego serwera do drugiego. Po zakończeniu obchodu nazwy ścieżki wszystkie pośrednie katalogi i docelowy plik znajdują się w pamięci podręcznej z gwarancjami przywołania. Przyszłe operacje otwierania tego pliku nie będą wymagały żadnej komunikacji przez sieć, chyba że nastąpi cofnięcie gwarancji przywołania w odniesieniu do jakiejś części nazwy ścieżki.

Jedynymi wyjątkami od zasady używania pamięci podręcznej są zmiany w katalogach, które – ze względu na spójność – wykonuje się bezpośrednio w serwerze odpowiedzialnym za dany katalog. Celom tym służą dobrze określone operacje interfejsu Vice. Proces Venus uwzględnia te zmiany w swoich kopiiach podręcznych, aby uniknąć ponownego sprowadzania katalogu.

17.6.3.4 Implementacja

Procesy klientów są sprzęgane z jądrem systemu UNIX za pomocą zwykłego zbioru odwołań do systemu. Jądro jest nieco zmienione, aby mogło wykrywać w stosownych operacjach odwołania do plików administrowanych przez procesy Vice oraz przesyłać zamówienia do użytkowego procesu Venus w stacji roboczej.

Proces Venus wykonuje tłumaczenie nazwy ścieżki składową po skądowej, jak to opisano powyżej. Ma on pamięć podręczną odwzorowań, kojarzącą tomy z umiejscowieniem serwerów, w celu unikania pytania serwerów o już poznane położenie tomu. Jeśli informacji o tomie nie ma w tej pamięci podręcznej, to Venus kontaktuje się z dowolnym serwerem, z którym ma aktualnie połączenie, zamawia informację o położeniu tomu i umieszcza ją w pamięci podręcznej odwzorowań. Jeśli proces Venus nie miał do tej pory połączenia z serwerem, to nawiązuje je. Następnie używa tego połączenia do pobierania pliku lub katalogu. Ustanowienie połączenia jest potrzebne w celu sprawdzenia tożsamości i zapewnienia bezpieczeństwa. Z chwilą odnalezienia docelowego pliku i umieszczenia go w pamięci podręcznej na lokalnym dysku powstaje jego kopię. Venus przekazuje wtedy sterowanie do jądra, które otwiera kopię pliku w pamięci podręcznej i zwraca jego uchwyt do procesu klienta.

System plików UNIX jest stosowany jako system pamięci niskiego poziomu zarówno dla serwerów, jak i dla klientów. Pamięć podręczna klienta

jest lokalnym katalogiem na dysku stacji roboczej. Wewnątrz tego katalogu znajdują się pliki, których nazwy rezerwują miejsce na wpisy w pamięci podręcznej. Zarówno Venus, jak i procesy serwerów mają do plików systemu UNIX dostęp bezpośredni za pomocą uniksowych i-węzłów. Unika się w ten sposób kosztownej procedury tłumaczenia nazw ścieżek na i-węzły (*namei*). Ponieważ wewnętrzny interfejs i-węzłów jest niewidoczny dla procesów klientów (Venus oraz procesy serwerów są procesami na poziomie klienta), dodano odpowiedni zbiór uzupełniających odwołań do systemu.

Proces Venus zarządza dwiema osobnymi pamięciami podręcznymi: jedną przeznaczoną na stan, a drugą na dane. Do utrzymania ograniczonego rozmiaru każdej z nich stosuje się prosty algorytm usuwania danych „najmniej ostatnio używanych” (LRU). Kiedy plik jest usuwany z pamięci podręcznej, wtedy proces Venus powiadamia odpowiedni serwer, że jest zwolniony z obowiązku przywołania go w związku z tym plikiem. Pamięć podręczna stanu znajduje się w pamięci wirtualnej, by umożliwić natychmiastową obsługę wywołań systemowych stat (przekazywanie stanu pliku). Pamięć podręczna danych rezyduje na dysku lokalnym, choć mechanizm buforowania wejścia-wyjścia systemu UNIX utrzymuje podręcznie pewną liczbę bloków w pamięci operacyjnej, co jest dla procesu Venus przezroczyste.

Pojedyncze procesy z poziomu klienta pełnią funkcje serwerów obsługujących wszystkie pochodzące od klientów zamówienia na dostęp do plików. Każdy z tych procesów do jednoczesnej obsługi wielu żądań od klientów stosuje pakiet procesów lekkich z planowaniem niewywłaszczeniowym. Z pakietem procesów lekkich jest zintegrowany pakiet zdalnych wywołań procedur (RPC). Umożliwia to serwerowi plików współbieżne wykonywanie lub obsługiwanie wywołań RPC – każdego przez osobny proces lekki. Mechanizm zdalnych wywołań procedur jest zbudowany powyżej abstrakcji datagramów niskiego poziomu. Całość przesyłania plików jest zrealizowana jako efekt uboczny zdalnych wywołań procedur. Na każdego klienta przypada jedno połączenie RPC, lecz nie istnieje z góry powiązanie procesów lekkich z tymi połączeniami. Zamiast tego zamówienia klientów we wszystkich połączeniach są obsługiwane przez pulę procesów lekkich. Zastosowanie jednego, wielowątkowego procesu serwera umożliwia podręczne przechowywanie struktur danych potrzebnych do obsługi zamówień. Z drugiej strony, awaria takiego procesu ma zgubne skutki w postaci sparalizowania danego serwera.

17.6.4 System Sprite

System Sprite jest eksperymentalnym systemem rozproszonym, który opracowano w University of California w Berkeley. Główny nacisk położono w nim na badania zagadnień sieciowych systemów plików, wędrówki procesów oraz

wysokowydajnych systemów plików. System Sprite pracuje na stacjach roboczych Sun i DEC. Jest używany do codziennych zastosowań przez dziesiątki studentów, nauczycieli akademickich i innych pracowników uczelni.

17.6.4.1 Przegląd

Projektanci systemu Sprite założyli, że następne generacje stacji roboczych będą maszynami o dużej mocy obliczeniowej, z rozległymi pamięciami fizycznymi. System Sprite jest przeznaczony dla konfiguracji maszynowej złożonej z wielkich i szybkich dysków skoncentrowanych w niewielu maszynach usługodawczych, które mają zaspokajać zapotrzebowanie na pamięć pochodzące od setek bezdyskowych stacji roboczych. Stacje robocze są połączone za pomocą wielu sieci lokalnych. Ponieważ stosuje się przechowywanie podręczne całych plików, wielkie pamięci fizyczne mają kompensować brak dysków lokalnych.

Ogólny interfejs udostępniany przez system Sprite, a zwłaszcza jego interfejs do systemu plików, jest dość podobny do interfejsu w systemie UNIX. System plików wygląda jak pojedyncze drzewo uniksowe, obejmujące wszystkie pliki i urządzenia sieci, czyniące je jednakowo przezroczyste dostępne z dowolnej stacji roboczej. Przejrzystość położenia zasobów w systemie Sprite jest pełna. Nie ma sposobu wykrycia położenia pliku w sieci na postawie jego nazwy.

W przeciwieństwie do systemu NFS, system Sprite wymusza spójność plików dzielonych. Każdemu odwołaniu do systemu za pomocą funkcji *read* zapewnia się otrzymanie w pełni aktualnych danych z pliku, nawet jeśli jest on jednocześnie otwarty przez wiele procesów zdalnych. Zatem system Sprite emuluje w środowisku rozproszonym pojedynczy system UNIX z podziałem czasu.

Wyróżniającą cechą systemu Sprite jest jego współdziałanie z systemem pamięci wirtualnej. Większość wersji systemu UNIX używa specjalnej strefy na dysku jako obszaru wymiany do celów związanych z działaniem pamięci wirtualnej. W przeciwieństwie do tego, do pamiętania danych i stosów wykonywanych procesów system Sprite używa zwykłych plików, zwanych *plikami pomocniczymi* (ang. *backing files*). Pozwala to uprościć przemieszczanie procesów oraz uelastycznia przestrzeń przeznaczoną na wymianę procesów i umożliwia jej dzielenie. Pliki pomocnicze są przechowywane w pamięciach podręcznych znajdujących się w pamięciach operacyjnych serwerów, tak jak inne pliki. Projektanci uważają, że klienci powinni móc czytać losowo strony z (fizycznej) pamięci podręcznej serwera szybciej niż z lokalnego dysku. Oznacza to, że serwer z wielką pamięcią podręczną może zapewniać wydajniejsze stronowanie niż lokalny dysk.

Pamięć wirtualna i system plików dzielą tę samą pamięć podręczną i negocują ze sobą jej podział stosownie do ich konfliktowych potrzeb. System

Sprite pozwala, aby rozmiary pamięci podręcznych plików na poszczególnych maszynach zwiększały się bądź malały w zależności od zmieniających się potrzeb ich pamięci wirtualnych i systemów plików. Schemat ten jest podobny do zastosowanego w systemie operacyjnym Apollo Domain, w którym rozmiar obszaru wymiany jest ustalany dynamicznie.

Opiszemy pokrótce niektóre inne cechy systemu Sprite. W odróżnieniu od systemu UNIX, w którym dzieleniu między procesy może podlegać tylko kod wykonywalny, Sprite zawiera mechanizm dzielenia między procesami klientów przestrzeni adresowej w stacji roboczej. Istnieje także możliwość wędrówki procesów – przezroczysta zarówno dla klientów, jak i dla przenoszonych procesów.

17.6.4.2 Tablice przedrostków

System Sprite przedstawia się klientowi w postaci jednej hierarchii systemu plików. Hierarchia ta składa się z wielu poddrzew, zwanych *domenami* (ang. *domains*) – to określenie jest używane w systemie Sprite w znaczeniu jednostki składowej. Każdy serwer zapewnia pamięć jednej lub kilku domenom. Każda maszyna utrzymuje tzw. *tablicę przedrostków* (ang. *prefix table*), której zadaniem jest odwzorowywanie domen na serwery. Odwzorowanie to jest tworzone i aktualniane dynamicznie za pomocą protokołu rozglaszania, który umieszcza w sieci komunikaty przeznaczone do czytania przez wszystkich pozostałych członków sieci. Opiszemy najpierw, jak używa się tych tablic przy poszukiwaniu nazw, a potem – jak dynamicznie zmienia się ich zawartość.

Każdy wpis w tablicy przedrostków odpowiada jednej domenie. Zawiera on nazwę szczytowego katalogu domeny (zwanego przedrostkiem domeną), adres sieciowy serwera przechowującego domenę oraz okreśnik numeryczny, identyfikujący główny katalog domeny dla przechowującego ją serwera. Okreśnik ten jest zazwyczaj indeksem w tablicy otwartych plików administrowanej przez serwer. Oszczędza się w ten sposób kosztownego tłumaczenia nazw.

Każda operacja szukania bezwzględnej nazwy ścieżki zaczyna się od przeglądnięcia przez klienta jego tablicy przedrostków w celu odnalezienia najbliższego przedrostka pasującego do danej nazwy pliku. Klient oddziela dopasowany przedrostek od nazwy pliku i posyła resztę nazwy do serwera wraz z okreśnikiem z tablicy przedrostków. Serwer używa tego okreśnika do zlokalizowania głównego katalogu domeny, po czym kontynuuje tłumaczenie reszty nazwy pliku za pomocą zwykłej procedury systemu UNIX. Jeśli serwer zakończy po myślnie tłumaczenie, to okreśnik otwartego pliku przekazuje do klienta.

Istnieje kilka sytuacji, w których serwer nie kończy operacji poszukiwania:

- Gdy serwer napotka w dowiązaniu symbolicznym bezwzględną nazwę ścieżki, wówczas natychmiast przekazuje ją do klienta. Klient szuka no-

wej nazwy w swojej tablicy przedrostków i rozpoczyna przeglądanie we współpracy z nowym serwerem.

- Nazwa ścieżki może prowadzić w górę, do korzenia domeny (wskutek użycia składowej „..” symbolizującej poprzednika). W takim przypadku serwer zwraca resztę ścieżki klientowi. Ten łączy otrzymaną resztę z przedrostkiem domeny, z której właśnie nastąpiło wyjście, aby uformować nową bezwzględną ścieżki.
- Nazwa ścieżki może także prowadzić w dół, do nowej domeny. Może się tu zdarzyć wtedy, kiedy domena nie jest wpisana do tablicy, wskutek czego przedrostek domeny powyżej pominiętej domeny jest najdłuższym z dających się dopasować. Wybrany serwer nie może zakończyć obchodu nazwy ścieżki, gdyż przekazana mu droga schodzi poniżej jego domeny. W alternatywnej sytuacji, gdy korzeń domeny jest poniżej katalogu roboczego i ścieżka do pliku w tej domenie ma postać nazwy względnej, serwer również nie może dokonać tłumaczenia. Rozwiązaniem w takich przypadkach jest umieszczenie znacznika wskazującego granice domeny (punkt montażu, w terminologii systemu NFS). Znacznik jest plikiem specjalnego rodzaju, zwany zdalnym dowiązaniem. Zawiera on, na podobieństwo dowiązania symbolicznego, nazwę pliku – w tym przypadku jest to jego własna nazwa. Kiedy serwer napotka zdalne dowiązanie, wtedy przekazuje jego nazwę klientowi.

Nazwy względne ścieżek są traktowane podobnie jak w zwykłym systemie UNIX. Gdy proces określa nowy katalog roboczy, wówczas w celu otwarcia tego katalogu stosuje się mechanizm przedrostków, po czym zarówno adres serwera, jak i okreśnik zostają przechowane w danych o stanie procesu. Jeśli operacja przeszukiwania wykryje nazwę względną ścieżki, to wysyła nazwę ścieżki wprost do serwera bieżącego roboczego katalogu, wraz z okreśnikiem katalogu. Dla serwera nie ma zatem różnicy w poszukiwaniach według nazw względnych i bezwzględnych.

Dotychczas zasadniczą różnicą, w porównaniu z odwzorowywaniem z udziałem uniksowego mechanizmu montowania, był początkowy krok dopasowywania nazwy pliku do tablicy przedrostków, następujący przeglądanie jej kolejnych składowych. W systemach (takich jak NFS i konwencjonalny UNIX), w których do poszukiwania nazw używa się pamięci podręcznych, występuje podobny efekt unikania przeglądania składowej za składową, z chwilą gdy w pamięci podręcznej znajdują się odpowiednie informacje.

O odrębności mechanizmu tablic przedrostków świadczy głównie sposób ich tworzenia i zmieniania. Serwer, który napotyka zdalne dowiązanie, otrzymuje wskazówkę, że klientowi brakuje wpisu domeny, której zdalne dowią-

zanie odnaleziono. Aby otrzymać brakującą informację przedrostkową, klient musi *ogłościć* w sieci komunikat z nazwą pliku. Rozgłoszany komunikat jest odbierany przez wszystkie systemy komputerowe w sieci. Serwer, który przechowuje dany plik, w odpowiedzi wysyla dotyczący tego pliku wpis z tablicy przedrostków, zawierający ciąg znaków, który może zostać użyty jako przedrostek, adres serwera i określnik odpowiadający korzeniowi domenę. Klient może wówczas wpisać te szczegóły do swojej tablicy przedrostków.

Na początku każdy klient ma pustą tablicę przedrostków. W celu odnalezienia wpisu domeny korzeniowej stosuje się protokół rozgłaszenia. Stopniowo, w miarę potrzeby, przybywa więcej wpisów; domena, do której nigdy się nie odwołano, nie pojawi się w tablicy.

Zawarte w tablicy przedrostków informacje o położeniu serwerów są wskazówkami, które koryguje się, jeśli okażą się złe. Jeżeli zatem klient próbuje otworzyć plik i nie otrzymuje odpowiedzi od serwera, to unieważnia wpis w tablicy przedrostków i próbuje dowiedzieć się czegoś, rozgłaszając w tej sprawie zapytanie. Jeśli serwer został przywrócony do działania, to odpowiada na ogłoszenie i wpis w tablicy przedrostków nabiera ważności. Ten sam mechanizm działa przy wznowieniu pracy serwera pod innym adresem sieciowym lub wówczas, gdy jego domeny zostały przeniesione do innych serwerów.

Mechanizm przedrostków zapewnia, że niezależnie od miejsca, w którym działa serwer przechowujący domenę, pliki domeny mogą być otwierane i można mieć do nich dostęp z dowolnej maszyny – bez względu na stan serwerów domen znajdujących się powyżej danej domeny. W istocie, wbudowany protokół rozgłaszenia umożliwia dynamiczne konfigurowanie i pewien stopień tolerowania uszkodzeń. Jeśli przedrostek domeny istnieje w tablicy klienta, to bezpośrednie połączenie między klientem a serwerem będzie nawiązane, jeżeli tylko klient spróbuje otworzyć plik w danej domenie (w przeciwnieństwie do schematów obchodu nazw ścieżek).

Jeśli na lokalnym dysku maszyny mają być przechowywane pewne prywatne pliki, to można umieścić w jej tablicy przedrostków wpis prywatnej domeny i odmawiać odpowiedzi na ogłoszenia dotyczące tej domeny. Jednym z zastosowań takiej możliwości może być katalog */usr/tmp*, który przechowuje pliki tymczasowe, tworzone przez wiele programów systemu UNIX. Każda stacja robocza potrzebuje dostępu do katalogu */usr/tmp*. Jednak w przypadku stacji z lokalnymi dyskami byłoby zapewne lepiej, aby używały one własnych dysków do organizowania przestrzeni roboczej. Przypomnijmy, że twórcy systemu Sprite przewidują, że operacje czytania z pamięci podręcznych w serwerach będą szybsze niż z dysków lokalnych, lecz nie odnoszą tej zasady do operacji pisania. Można utworzyć prywatne domeny */usr/tmp*, a klientów nie mających dysków obsługiwać za pomocą serwera sieciowego udostępniającego publiczną wersję tej domeny. Wszystkie rozgłoszane w sieci

zyczenia dotyczące katalogu `/usr/tmp` będą obsługiwane przez serwer publiczny.

Istnieje również elementarny sposób zwiększenia informacji przeznaczonej wyłącznie do czytania. Można spowodować, aby serwery przy przechowywaniu kopii domeny zaopatrywały różnych klientów w różne wpisy przedrostków (odpowiadające różnym kopiom) tej samej domeny. Tą samą technikę można zastosować przy dzieleniu plików binarnych przez różnego rodzaju sprzęt komputerowy.

Ponieważ tablica przedrostków pozwala ominąć część mechanizmu przeglądania katalogu, pomija się również sprawdzanie pozwoleń przy przeszukiwaniu. W rezultacie wszystkie programy mają niejawne prawo przeszukiwania wszystkich ścieżek reprezentujących przedrostki domen. Jeżeli jest wymagane ograniczenie dostępu do domeny, to musi ono wystąpić od korzenia domeny lub poniżej tego korzenia.

17.6.4.3 Przechowywanie podrzczne a spójność

Istotnym aspektem projektu systemu plików Sprite jest poszerzone zastosowanie technik pamięci podrzcznej. Przyjęcie za podstawę systemu wielkich pamięci operacyjnych i promowanie używania bez dyskowych stacji roboczych spowodowało, że pliki są przechowywane podrzecznie w pamięciach operacyjnych, a nie na lokalnych dyskach (jak w systemie Andrew). Pamięci podrzcznych używają zarówno klienci, jak i stacje usługodawcze. Podstawową jednostką organizacji pamięci podrzcznej są bloki, a nie pliki (jak w systemie Andrew). Rozmiar bloku wynosi obecnie 4 KB. Każdy blok w pamięci podrzcznej jest wirtualnie zaadresowany za pomocą określnika pliku i położenia bloku w pliku. Używanie adresów wirtualnych zamiast fizycznych adresów dyskowych umożliwia klientom tworzenie nowych bloków w pamięci podrzcznej i odnajdywanie dowolnego bloku bez pobierania i-węzła pliku od serwera.

Gdy wystąpi odwołanie do jądra w celu przeczytania bloku pliku, wówczas jądro najpierw sprawdzi swoją pamięć podrzczną. Jeżeli jądro znajdzie blok w pamięci podrzcznej, to go przekaze. W przeciwnym razie jądro przeczyta blok z dysku (jeśli plik jest przechowywany lokalnie) lub wyśle zamówienie do serwera. W każdym przypadku nastąpi dodanie bloku do pamięci podrzcznej połączone z usunięciem bloku używanego najdawniej. Jeżeli blok jest zamawiany w serwerze, to serwer sprawdza swoją pamięć podrzczną przed wywołaniem dyskowej operacji wejścia-wyjścia i dodaje blok do pamięci podrzcznej, jeśli go tam nie było. Obecnie w systemie Sprite nie stosuje się czytania z wyprzedzeniem w celu przyspieszania dostępu sekwencyjnego (w przeciwieństwie do NFS).

Przy wykonywaniu zmian w plikach stosuje się metodę opóźnianego pisania. Gdy program użytkowy wydaje jądru polecenie pisania, wówczas jądro

po prostu zapisuje blok do swojej pamięci podręcznej i przekazuje sterowanie do programu. Blok ten dopóty nie będzie przepisany do pamięci podręcznej serwera ani na dysk, dopóki nie trzeba go będzie z niej wyrzucić lub nie minie około 30 s od czasu jego ostatniej zmiany. Tak więc blok zapisany w maszynie klienta zostanie zapisany w pamięci podręcznej serwera nie później niż po 30 s. a na dysku serwera znajdzie się po następnych 30 s. Takie postępowanie poprawia wydajność systemu kosztem możliwości utraty ostatnich zmian wskutek awarii.

Do wymuszenia spójności plików dzielonych użyto w systemie Sprite schematu numerów wersji. Numer wersji pliku jest zwiększany przy każdym otwarciu pliku do pisania. Kiedy klient otwiera plik, wtedy otrzymuje od serwera bieżący numer wersji pliku. Ten numer wersji klient porównuje z numerem wersji przypisanym do bloków tego pliku znajdujących się w pamięci podręcznej. Jeśli numery są różne, to klient rezygnuje ze wszystkich bloków pliku w pamięci podręcznej i wprowadza je do niej ponownie z serwera w miarę potrzeby. Z powodu stosowania metody opóźnionego pisania serwer nie zawsze ma aktualne dane pliku. Serwery nadzorują tę sytuację przez utrzymywanie dla każdego pliku śladu ostatniego klienta wykonującego operację pisania. Jeśli plik otwiera klient inny niż ten, który ostatnio dany plik zapisywał, to serwer zmusza klienta, który pisał jako ostatni, aby wszystkie zmienione bloki danych odesłał mu do jego pamięci podręcznej.

Jeśli serwer wykryje (przy operacji otwierania pliku), że plik jest otwarty w kilku stacjach roboczych i przynajmniej jedna z nich używa tego pliku do pisania, to zakazuje klientowi umieszczania danego pliku w pamięci podręcznej. Wszystkie następne operacje pisania i czytania są wykonywane za pośrednictwem serwera, który je szereguje. Zakaz przechowywania w pamięci podręcznej odnosi się do pliku, wobec czego dotyczy on tylko klientów z otwartymi plikami. Przy zakazie używania pamięci podręcznej występuje, rzecz jasna, istotny spadek wydajności. Plik objęty zakazem przechowywania w pamięciach podręcznych będzie można ponownie w nich przechowywać wówczas, gdy zostanie on zamknięty przez wszystkich klientów. Zezwala się na jednoczesne przechowywanie pliku w pamięciach podręcznych wielu procesów czytających.

Podejście to zależy od powiadamiania serwera o każdym otwarciu lub zamknięciu pliku. Zadanie takiego powiadamiania uniemożliwia taką optymalizację działania, jak przechowywanie podręczne nazw, przy którym klienci otwierają pliki bez kontaktowania się z serwerami plików. W zasadzie serwery pełnią tu funkcję scentralizowanych punktów kontroli spójności pamięci podręcznych. Aby móc spełniać to zadanie, muszą utrzymywać informacje o stanie otwartych plików.

17.6.5 System Locus

Celem projektu Locus opracowanego w University of California w Los Angeles było zbudowanie rozproszonego systemu operacyjnego w pełnej skali. System Locus jest nadbudową systemu UNIX, lecz w odróżnieniu od systemów NFS, UNIX United i innych systemów rozproszonych wywodzących się z systemu UNIX, ma poważne rozszerzenia, wymagające całkowicie nowego, a nie tylko zmienionego, jądra.

17.6.5.1 Przegląd

System plików Locus dla klientów i programów użytkowych jest jedną hierarchią nazewniczą o strukturze drzewiastej. Struktura ta obejmuje wszystkie obiekty (pliki, katalogi, pliki wykonywalne i urządzenia) wszystkich maszyn w systemie. Nazwy w systemie Locus są w pełni przezroczyste – nie można na podstawie nazwy obiektu określić położenia tego obiektu w sieci. W pierwszym przybliżeniu nie można prawie dostrzec różnicy między strukturą nazewniczą systemu Locus a standardowym drzewem systemu UNIX.

W systemie Locus jednemu plikowi może odpowiadać zbiór kopii rozproszonych na różnych stanowiskach. Wprowadzono dodatkowy wymiar przezroczystości, polegający na tym, że system odpowiada za aktualność wszystkich kopii i zapewnia, że zamówienia na dostęp są kierowane do najnowszych wersji informacji. Klienci mogą sprawować pieczę zarówno nad liczbą, jak i umiejscowieniem kopii plików, lecz wolno im też być całkowicie nieświadomymi schematu zwielokrotniania. Zwielokrotnianie pliku w systemie Locus służy głównie do zwiększenia dostępności danych do czytania na wypadek awarii i podziału sieci. Modyfikacje wykonuje się na zasadzie uaktualniania kopii podstawowej.

Semantyka dostępu do plików w systemie Locus naśladuje tę, którą prezentuje klientom standardowy UNIX. System Locus realizuje tę semantykę w rozproszonym i zwielokrotnionym środowisku, w którym działa. Istnieje również alternatywny mechanizm pomocniczego i wymuszonego blokowania* plików i ich części. Ponadto za pomocą systemowych funkcji *commit* i *abort* można uaktualniać pliki w sposób niepodzielny.

W projekcie systemu Locus położono nacisk na działanie w warunkach awarii i podziałów sieci. Dopóki jest dostępna kopia pliku, dopóty mogą być obsługiwane zamówienia na czytanie, przy czym jest gwarantowane, że czytana wersja będzie najnowszą z dostępnych. Podczas dołączania oderwanych od sieci stanowisk pamięci stosuje się automatyczne mechanizmy uaktualniania przestarzałych kopii plików.

* Tj. zajmowania na wyłączny użytk. – Przyp. tłum.

Nacisk na wysoką wydajność doprowadził w projekcie Locus do włączania funkcji sieciowych (takich jak formatowanie danych, obsługa kolejek i obukierunkowe przesyłanie komunikatów) do systemu operacyjnego. Do komunikacji między jądrami opracowano specjalne protokoły operacji zdalnych. Kontrastuje to z typowym podejściem polegającym na używaniu protokołu RPC lub innych, istniejących protokołów. Dzięki zmniejszeniu liczby warstw sieci osiągnięto wysoką efektywność operacji zdalnych. Z drugiej strony wyspecjalizowany protokół utrudnia przenoszenie systemu Locus do innych sieci i systemów plików.

Do obsługi zdalnych zamówień utworzono wydajny, choć ograniczony mechanizm procesowy, nazywany *procesami serverów* (ang. *server processes*). Są to procesy lekkie, które nie mają żadnej nieuprzywilejowanej przestrzeni pamięci. Cały ich kod i stosy rezydują w rdzeniu systemu operacyjnego. Mogą one bezpośrednio wywoływać wewnętrzne procedury systemu operacyjnego i dzielić pewne dane. Procesy te są przypisane do obsługi zamówień sieciowych, które gromadzą się w kolejce systemowej. System jest skonfigurowany za pomocą pewnej liczby tych procesów, przy czym liczba ta jest automatycznie i dynamicznie zmieniana podczas działania systemu.

17.6.5.2 Struktura nazewnicza

Logiczna struktura nazewnicza ukrywa przed klientami i aplikacjami zarówno szczegóły położenia, jak i zwieleniowania plików. W rezultacie, połączenie ze sobą logicznych grup plików tworzy jednolitą strukturę. Logiczna grupa plików jest odwzorowywana fizycznie na wiele fizycznych kontenerów (nazywanych również paczkami), które rezydują na różnych stanowiskach i przechowują kopie plików danej grupy. Para *<numer-logicznej-grupy-pliku, numer-i-węzła>*, którą przyjęto nazywać *oznacznikiem pliku* (ang. *designator*), służy jako globalnie jednoznaczna, niskopoziomowa nazwa pliku. Zauważmy, że sam oznacznik ukrywa zarówno szczegóły lokalizacji, jak i zwieleniowania pliku.

Każde stanowisko ma zwarty i pełny obraz logicznej struktury nazw. Tabela logicznych montaży jest powielona globalnie i zawiera wpisy wszystkich logicznych grup plików. Pozycja tej tablicy zawiera oznacznik katalogu, w którym grupa plików jest zamontowana logicznie, oraz informację o tym, które stanowisko jest bieżąco odpowiedzialne za synchronizację dostępu w obrębie grupy pliku. Zadania takiego stanowiska wyjaśniamy w dalszej części tego punktu. Ponadto każde stanowisko przechowujące kopię katalogu, w którym jest zamontowane poddrzewo, musi przechowywać w pamięci i-węzeł tego katalogu z adnotacją, że jest nad nim nadbudowa. Przechowywanie i-węzła w pamięci służy przechwytywaniu każdego dostępu do tego katalogu, pochodzącego z dowolnego stanowiska, i umożliwianiu działania standardowego, pośredniego mechanizmu montowania systemu UNIX (przez

tablicę logicznych montaży, zob. p. 11.1.2). Niezbędne uaktualnienia tablic logicznych montaży na wszystkich stanowiskach wykonuje się według protokołu zaimplementowanego w systemie Locus za pomocą funkcji systemowych **mount** i **umount**.

Na poziomie fizycznym kontenery fizyczne odpowiadają strefom (partyjom) dyskowym i mają przypisane numery paczek, które – wraz z numerem logicznej grupy plików – identyfikują poszczególne paczki. Jedna z paczek jest oznaczona jako *kopia podstawowa*. Plik musi być przechowyany na stanowisku kopii podstawowej, a poza tym może być pamiętany w dowolnym podzbiorze innych stanowisk, na których istnieje paczka odpowiadająca jego grupie. Tak więc kopia podstawowa przechowuje pełną grupę plików, podczas gdy pozostałe paczki mogą być częściowe.

Zwielokrotnianie jest szczególnie pozytyczne w przypadku katalogów na wysokich poziomach hierarchii nazw. Katalogi takie są najczęściej dostępne wyłącznie do czytania i mają zasadnicze znaczenie przy tłumaczeniu nazw ścieżek większości plików.

Różne kopie pliku mają przypisany ten sam numer i-węzła we wszystkich paczkach grup plików. W związku z tym paczka ma pustą przegródkę dla i-węzła każdego pliku, którego nie przechowuje. Numery stron danych mogą być różne w różnych paczkach, toteż w odniesieniach do stron danych w sieci używa się logicznych numerów stron zamiast fizycznych. Każda paczka ma odwzorowanie tych logicznych numerów na odpowiednie numery fizyczne. Aby można było zautomatyzować zarządzanie zwielokrotnianiem, każdy i-węzeł kopii pliku zawiera numer wersji przesądzający o tym, która kopia zastąpi inne kopie.

Każde stanowisko ma tablicę kontenerów, która odwzorowuje numery logicznych grup plików na położenie na dysku tych grup, których paczki są lokalne na danym stanowisku. Gdy do stanowiska nadchodzą zamówienia na dostęp do plików przechowywanych lokalnie, wówczas system korzysta z tej tablicy, aby odwzorować oznacznik pliku na lokalny adres dyskowy.

Choć w większości przypadków globalna jednoznaczność nazw plików jest ważna, istnieją pewne pliki i katalogi specyficzne dla sprzętu i stanowiska (tj. katalog */bin*, który zależy od sprzętu, oraz */dev*, który jest charakterystyczny dla danego stanowiska). System Locus dostarcza przezroczystych środków tłumaczenia odniesień do tych tradycyjnych nazw plików na pliki zależne od sprzętu i stanowiska.

17.6.5.3 Operacje na plikach

Przyjęte w systemie Locus podejście do operacji na plikach z pewnością odbiega od typowego modelu współpracy między klientem a serwerem. Wprowadzenie plików zwielokrotnionych z dostępem synchronicznym wymaga

dodatkowych funkcji. W systemie Locus wyodrębniono trzy logiczne zadania związane z obsługą dostępu do pliku, z których każde może być pełnione przez inne stanowisko:

- **Stanowisko użytkujące (SU):** Stanowisko takie wydaje zamówienia na otwarcie i dostęp do pliku zdalnego.
- **Stanowisko przechowujące (SP):** Stanowisko takie jest wyznaczone do obsługi zamówień.
- **Bieżące stanowisko synchronizujące (BSS):** Stanowisko BSS wymusza globalne zasady synchronizacji w grupie plików oraz wybiera stanowisko przechowujące dla każdego zamówienia otwarcia, odnoszącego się do pliku w danej grupie plików. Dla każdej grupy plików istnieje co najwyżej jedno stanowisko synchronizujące w każdym zbiorze komunikujących się stanowisk (tj. strefie). Stanowisko synchronizujące utrzymuje numer wersji i wykaz fizycznych kontenerów dla każdego pliku w grupie plików.

Opiszymy teraz sposób wykonywania przez te stanowiska operacji otwierania, czytania, pisania, zamykania, zatwierdzania oraz zaniechania. Związane z tym zagadnienia synchronizacji są opisane osobno, w następnym punkcie.

Otwarcie pliku rozpoczyna się następująco. Stanowisko użytkujące określa właściwe, bieżące stanowisko synchronizujące w wyniku odnalezienia grupy plików w tablicy logicznych montażów, po czym przekazuje do niego zamówienie otwarcia pliku. Bieżące stanowisko synchronizujące odpytuje potencjalne stanowiska przechowujące dany plik, aby zdecydować, które z nich będzie działać jako rzeczywiste stanowisko SP. Do komunikatów z zapytaniami jest dołączony numer wersji konkretnego pliku, aby potencjalne stanowiska SP mogły przez porównanie tego numeru z własnymi zdecydować, czy ich kopie są aktualne. Stanowisko BSS po rozważeniu odpowiedzi, które otrzymuje od kandydujących stanowisk, wybiera stanowisko SP, po czym wysyła dane o jego tożsamości do stanowiska SU. Oba stanowiska – BSS i SP – przydzielają sobie w pamięci operacyjnej struktury i-węzła otwartego pliku. Stanowisko BSS potrzebuje tej informacji do podejmowania przyszłych decyzji synchronizacyjnych, a stanowisko przechowujące utrzymuje i-węzły w celu wydajnej obsługi przyszłych dostępów.

Po otwarciu pliku zamówienia na czytanie są przesyłane wprost do stanowiska przechowującego, bez interwencji stanowiska BSS. Zamówienie czytania zawiera oznacznik pliku, logiczny numer potrzebnej strony wewnętrz pliku oraz próbę określenia miejsca przechowania i-węzła w pamięci stanowiska SP. Po odnalezieniu i-węzła stanowisko SP tłumaczy logiczny numer strony na numer fizyczny i wywołuje standardową, niskopoziomową procedu-

rę w celu przydzielenia bufora i pobrania z dysku odpowiedniej strony. Bufor zostaje wstawiony do sieciowej kolejki zamówień oczekujących na wysłanie. Po nadaniu do stanowiska użytkującego bufor ten jest umieszczany w buforze jądra. Po dostarczeniu strony do stanowiska SU następne operacje czytania są obsługiwane przez bufor w jądrze. Tak jak w przypadku czytania z lokalnego dysku, do przyspieszenia czytania sekwencyjnego przydaje się czytanie z wyprzedzeniem – zarówno na stanowisku użytkującym, jak i przechowującym.

Jeśli proces traci połączenie ze zdalnie czytanym plikiem, to system próbuje otworzyć ponownie inną kopię tej samej wersji pliku.

Tłumaczenie nazwy ścieżki na oznacznik pliku odbywa się niemal tak samo jak konwencjonalny obchód nazwy ścieżki w systemie UNIX, ponieważ Locus używa uniksowych nazw ściezek, bez żadnych wyjątków (w odróżnieniu od systemu UNIX United). Każde poszukiwanie składowej nazwy ścieżki w obrębie katalogu powoduje otwarcie i przeczytanie tego katalogu. Zauważmy, że nie ma tu zbieżności z operacją przeszukiwania zdalnego stosowaną w systemie NFS oraz że faktycznie to klient, a nie serwer przeszukuje katalog.

Katalog, w którym przegląda się nazwę ścieżki, nie jest otwierany do zwykłego czytania, lecz do wewnętrznego, niesynchronizowanego czytania. Różnica polega na tym, że nie jest tu potrzebna żadna globalna synchronizacja, a podczas czytania nie trzeba blokować pliku, tzn. katalog może być aktualniany w trakcie tej czynności. Jeśli katalog jest lokalny, to bieżące stanowisko synchronizujące nie jest nawet informowane o takim dostępie.

W systemie Locus przy wprowadzaniu zmian do pliku stosuje się zasadę aktualniania kopii podstawowej. Bieżące stanowisko synchronizujące musi wybrać stanowisko z paczką kopii podstawowej jako stanowisko przechowujące, służące do udzielania dostępów do pisania. Modyfikowanie danych przybiera dwie postacie. Jeśli zmiana nie dotyczy całej strony, to poprzednia wersja strony jest czytana ze stanowiska SP za pomocą protokołu czytania. Jeśli zmiana obejmuje całą stronę, to bez żadnego czytania tworzy się bufor na stanowisku SU. W każdym przypadku po wprowadzeniu zmian, być może z opóźnionym zapisem, strona będzie przesłana z powrotem do stanowiska przechowującego. Zanim zmodyfikowany plik zostanie zamknięty, wszystkie zmienione strony muszą być odeslane do stanowiska przechowującego.

Jeśli plik został zamknięty przez ostatni proces klienta na stanowisku użytkującym, to stanowiska SP i BSS powinny o tym zostać powiadomione, aby mogły zwolnić miejsce po znajdujących się pamięci operacyjnej strukturach i-węzłów, a także po to, by stanowisko synchronizujące mogło zmienić stan danych, które mogłyby wpływać na jego następną decyzję synchronizacyjną.

Istnieją systemowe funkcje zatwierdzania i zaniechania. Zatwierdzenie pliku następuje przy jego zamknięciu. Jeśli plik jest otwarty do wykonywania

zmian przez więcej niż jeden proces, to wprowadzane zmiany dopóty nie są stałe, dopóki jeden z procesów nie zażąda od systemu ich zatwierdzenia (wywołanie **commit**) albo wszystkie procesy nie zamkną pliku.

Gdy do pliku wprowadza się zmiany, wówczas na stanowisku przechowującym są przydzielane strony-cienie. Aktualizuje się przechowywaną w pamięci operacyjnej kopię i-węzła, tak aby wskazywała nowe strony-cienie. I-węzeł na dysku pozostaje niezmieniony, wskazując na strony pierwotne. Niepodzielna operacja zatwierdzania polega na zastąpieniu i-węzła na dysku i-węzłem z pamięci operacyjnej. Od tej chwili plik zawiera nowe dane. Aby zaniechać wykonanych zmian, usuwa się po prostu i-węzeł przechowywany w pamięci operacyjnej i zwalnia przestrzeń na dysku wykorzystaną do pamiętania zmian. Stanowisko użytkujące nigdy nie działa na rzeczywistych stronach dyskowych, lecz na stronach logicznych. Dzięki temu cały mechanizm stron-cieni jest zaimplementowany na stanowisku przechowującym i jest przezroczysty dla stanowiska SU.

System Locus w pierwszej kolejności zatwierdza zmiany zawartości pliku w kopiach podstawowej. Dopiero potem komunikaty o zmianach w pliku są wysyłane do wszystkich innych stanowisk SP oraz do bieżącego stanowiska synchronizującego. Komunikaty te zawierają co najmniej identyfikację zmienionego pliku oraz jego nowy numer wersji (aby zapobiec próbom czytania starych wersji). Od tej chwili odpowiedzialność za aktualnienie pliku, przez przekazywanie go w całości lub tylko niezbędnych zmian, spada na te dodatkowe stanowiska przechowujące. W jądrze każdego stanowiska jest utrzymywana kolejka zamówień aktualizacji; obsługuje ją wydajnie proces jądrowy za pomocą odpowiednich zamówień na czytanie. W procedurze rozpowszechniania nowych danych stosuje się standardowy mechanizm zatwierdzania plików. Jeśli zatem zostanie utracony kontakt z plikiem zawierającym nowszą wersję, plik lokalny pozostaje z kopią nienaruszoną, choć ciągle nieaktualną.

Korzystając z mechanizmu zatwierdzania plików dysponuje się stale albo pierwotnym plikiem, albo plikiem całkowicie zmienionym, nigdy nie mając do czynienia ze zmianami częściowymi – nawet w wypadku awarii stanowiska.

17.6.5.4 Synchronizowany dostęp do plików

Przy dostępcach do plików w środowisku rozproszonym system Locus usiłuje emulować konwencjonalną semantykę uniksową. W standardowym systemie UNIX zezwala się na jednocześnie otwarcie tego samego pliku przez wiele procesów. Procesy takie wykonują systemowe funkcje **read** i **write**, a system zapewnia, że każda następna operacja widzi skutki operacji, która ją poprzedziła. Schemat ten można dość łatwo urzeczywistnić, jeśli procesy działają w tym samym systemie operacyjnym, dzieląc te same struktury danych i pamięci podrzeczną. W celu uszeregowania zamówień używa się wtedy bloko-

wania struktur danych. Ponieważ w systemie Locus istnieje przetwarzanie zdalne, mogą powstawać sytuacje, w których uczestniczące w dzieleniu zasobów procesy nie pozostają na tej samej maszynie, co znacznie komplikuje implementację.

Można rozważyć dwa tryby dzielenia zasobów. Po pierwsze, w systemie UNIX kilka procesów wywodzących się od tego samego przedka może dzielić tę samą, bieżącą pozycję (odległość) w pliku. Do zachowania tego specjalnego trybu dzielenia opracowano schemat z pojedynczym żetonem. Stanowisko może tylko wtedy wykonywać funkcje systemowe odnoszące się do pozycji w pliku, kiedy dysponuje żetonem.

Po drugie, w systemie UNIX wiele procesów może dzielić ten sam i-węzeł pliku, przechowywany w pamięci operacyjnej. W systemie Locus sytuacja ta jest o wiele bardziej skomplikowana, ponieważ i-węzeł pliku może się znajdować w pamięciach podręcznych wielu stanowisk. Również strony danych są przechowywane podręcznie na wielu stanowiskach. Do synchronizacji dzielenia i-węzła oraz danych pliku stosuje się schemat z wieloma żetonomi danych. Wymusza się politykę pojedynczego, działającego na prawach wyłączności, pisarza i wielu czytelników. Modyfikowaniem pliku może się zajmować tylko to stanowisko, które ma żeton uprawniający do pisania, a każde stanowisko z żetonem czytania może plik czytać. Oba schematy z żetonomi są koordynowane przez zarządców żetonów, działających na odpowiednich stanowiskach przechowujących pliki.

Poprawność danych na stronach przechowywanych podręcznie gwarantuje się tylko wtedy, gdy jest obecny żeton danych. Gdy żeton pisania jest zabierany ze stanowiska, wówczas i-węzeł oraz wszystkie zmienione strony kopiuje się z powrotem do stanowiska przechowującego. Ponieważ pod nieobecność żetona plik mógłby ulegać niekontrolowanym zmianom, z chwilą oddania żetona wszystkie przechowywane w pamięciach podręcznych bufora zostają unieważnione. Kiedy stanowisko otrzyma żeton danych, zarówno i-węzeł, jak i strony danych muszą być sprowadzone ze stanowiska SP. Od tej polityki istnieje kilka wyjątków. Czytanie i zapisywanie niektórych atrybutów (np. funkcja stat), a także operacje czytania i modyfikowania katalogu (np. lookup) nie podlegają ograniczeniom synchronizacji. Wywołania takich funkcji są przesyłane wprost do stanowisk przechowujących, na których wykonyuje się zmiany, zatwierdza je i upowszechnia na wszystkie stanowiska przechowujące i użytkujące.

Opisany mechanizm gwarantuje spójność. Każdy dostęp dotyczy najnowszych danych. Inną kwestią, związaną z synchronizacją dostępu, jest uszeregowanie dostępów. W tym celu system Locus umożliwia blokowanie całych plików lub ich części. Blokowanie może pełnić funkcję wspierającą (sprawdzane tylko w wyniku próby zablokowania) lub być wymuszone (sprawdzane

przy wszystkich operacjach czytania i pisania). Jeśli proces nie może natychmiast zablokować pliku, to może wybrać między swoim załamaniem a czekaniem na odblokowanie pliku.

17.6.5.5 Działanie w środowisku narażonym na błędy

Podstawową zasadą działania systemu Locus jest utrzymywanie ściszej synchronizacji między kopiami pliku w jednej strefie, tak aby wszyscy klienci tego pliku w obrębie tej strefy korzystali z najnowszej wersji pliku.

Zasada uaktualniania kopii podstawowej eliminuje możliwość konfliktowych aktualizacji, gdyż kopia podstawowa, aby mogła być dostępna i uaktualniana, musi znaleźć się razem z klientem w jednej strefie. Niemniej jednak pozostaje do rozwiązania zadanie wykrywania uaktualnień i przenoszenia ich do wszystkich pozostałych kopii, zwłaszcza że dopuszcza się aktualizacje w sieci podzielonej. W normalnych warunkach pracy protokół zatwierdzania zapewnia właściwe wykrywanie i upowszechnianie uaktualnień, co opisano w poprzednim punkcie. Jednak przywracanie do działania stanowisk wymagających uaktualnienia ich paczek wymaga bardziej złożonego schematu. W tym celu system utrzymuje dla każdej grupy plików *licznik zatwierdzeń* (ang. *commit count*), zliczający każde zatwierdzenie dowolnego pliku z grupy. Każda paczka ma *znacznik dolnego poziomu* (ang. *lower-water mark* – LWM), będący wartością licznika zatwierdzeń, poniżej której system zapewnia, że wszystkie poprzednie zatwierdzenia znalazły odzwierciedlenie w paczce. Paczka z kopią podstawową przechowuje ponadto w pamięci pomocniczej pełny wykaz wszystkich ostatnio wykonanych zatwierdzeń*. Podczas dołączania paczki do strefy następuje kontakt ze stanowiskiem kopii podstawowej i sprawdzenie, czy jej znacznik LWM mieści się w ramach ostatnich ograniczeń listy zatwierdzeń. Jeśli tak, to stanowisko przechowujące paczkę umieszcza w kolejce do procesora proces jądrowy, który doprowadzi paczkę do spójnego stanu przez wykonanie pominiętych aktualizacji. Jeśli paczka podstawowa jest niedostępna, to nie cezwała się na pisanie w danej strefie, lecz po wyborze nowego bieżącego stanowiska synchronizującego (BSS) jest możliwe czytanie. Nowe stanowisko BSS komunikuje się z członkami strefy, aby dla każdego pliku w grupie plików zorientować się co do jego najnowszej (w danej strefie) wersji. Po ustaleniu tych faktów przez nowe stanowisko synchronizujące inne stanowiska z paczkami mogą z nim uzgodnić stan bieżący. Jako wynik tego postępowania wszystkie komunikujące się stanowiska uzyskują ten sam obraz grupy plików – na tyle pełny, na ile jest to możliwe w danej strefie. Zauważmy, że skoro aktualizacje są dozwolone

* Pamiętajmy (p. 17.6.5.2), że w terminologii systemu Locus „paczka” (ang. *pack*) oznacza jedno z wielu fizycznych odwzorowań grupy plików. W tym sensie zwrot „paczka przechowuje” (ang. *pack keeps*) itp. odnosi się do utrzymującego ją stanowiska. – Przyp. tłum.

w strefie z kopią podstawową, a w pozostałych strefach zezwala się na czytanie, to może dochodzić do czytania nieaktualnych kopii pliku. Tak więc Locus poświęca spójność na rzecz możliwości dokonywania zarówno aktualizacji, jak i czytania plików w środowisku podzielonym.

Jeśli paczka jest znacznie przeterminowana (tzn. jej znacznik LWM ma wartość wcześniejszą od najwcześniejszą wartości licznika zatwierdzeń na wykazie zatwierdzeń kopii podstawowej), to system wywołuje proces z poziomu użytkownika w celu doprowadzenia danej grupy plików do stanu aktualnego. W takiej sytuacji systemowi brakuje wystarczającej wiedzy o ostatnich zatwierdzeniach, aby odtworzyć zmiany. W zamian stanowisko musi przejrzeć całą przestrzeń i-węzłów, aby ustalić które pliki w jego paczce są przeterminowane.

Jeśli stanowisko wypadło z działania w sieci Locus, to jest niezbędna procedura czyszczenia. Gdy tylko jakieś stanowisko ustali, że pewne konkretne stanowisko jest niedostępne, wówczas musi wywołać procedurę awaryjną w odniesieniu do wszystkich zasobów, które były w użyciu procesów z danego stanowiska. Ta niemała procedura czyszczenia stanowi cenę za przechowywanie informacji o stanie przez wszystkie trzy stanowiska uczestniczące w dostępie do pliku.

Ponieważ aktualnianie katalogu nie jest ograniczone do kopii podstawowej, mogą powstawać konflikty między aktualizacjami dokonywanymi w różnych strefach. Jednak ze względu na prostotę wprowadzania zmian we wpisie katalogowym sprawę tę załatwia procedura automatycznych uzgodnień. Działa ona na zasadzie porównywania par i-węzłów oraz tekstowych nazw kopii tego samego katalogu. Najbardziej skrajne działanie jest podejmowane wtedy, kiedy ta sama nazwa w postaci tekstuowej odpowiada dwu różnym i-węzłom. Zmienia się wówczas nieznacznie nazwę pliku, a jego właściciela informuje za pomocą poczty elektronicznej.

17.7 ■ Podsumowanie

Rozproszony system plików jest systemem usług plikowych, którego klienci, serwery i urządzenia pamięci znajdują się na różnych stanowiskach systemu rozproszonego. Wskutek tego usługi muszą być wykonywane za pośrednictwem sieci, a zamiast jednego, skoncentrowanego magazynu danych istnieje wiele niezależnych urządzeń pamięci.

Idealny rozproszony system plików powinien się wydawać jego klientom zwykłym systemem skoncentrowanym. Zwielokrotnienie i rozproszenie jego serwerów oraz urządzeń pamięci powinno być przezroczyste. Oznacza to, że interfejs klienta rozproszonego systemu plików nie powinien odróżnić plików

lokalnych od zdalnych. Obowiązkiem rozproszonego systemu plików jest odnajdywanie plików i organizowanie przesyłania danych. Przeczysty, rozproszony system plików umożliwia klientowi mobilność przez przenoszenie całego środowiska klienta do tego stanowiska, na którym klient się rejestruje.

W rozproszonych systemach plików istnieje kilka schematów nazewnictwowych. W najprostszym podejściu nazwy plików są tworzone jako pewne połączenia nazw ich maszyn macierzystych i nazw lokalnych, co gwarantuje jednoznaczność w ramach całego systemu. Inny sposób, popularzowany przez system NFS, umożliwia dodawanie zdalnych katalogów do katalogów lokalnych w celu utworzenia wrażenia jednolitego drzewa katalogów.

Zamówienia na dostęp do zdalnych plików są na ogół obsługiwane przy użyciu dwóch, wzajemnie uzupełniających się metod. W obsłudze zdalnej zamówienia na dostęp są dostarczane do serwera. Maszyna serwera realizuje dostęp do danych i przesyła wyniki z powrotem do klienta. W metodzie *przechowywania podręcznego* dane, których nie ma aktualnie w pamięci podręcznej, są sprowadzane (ich kopie) do klienta z serwera. Dostęp dotyczy kopii danych przechowywanych w pamięciach podręcznych. Obowiązuje zasada pozostawiania w pamięci podręcznej ostatnio używanych bloków dyskowych, aby powtórne zamówienia na dostęp do tych samych informacji mogły być obsłużone lokalnie, bez wzmagania ruchu w sieci. W celu ograniczenia rozmiarów pamięci podręcznej stosuje się algorytm zastępowania danych. Problem utrzymywania w pamięciach podręcznych kopii zgodnych z podstawową kopią pliku nazywa się *problemem spójności pamięci podręcznej*.

Są dwie metody postępowania z informacjami po stronie serwera. Albo serwer śledzi dostęp każdego klienta do każdego pliku, albo po prostu dostarcza bloki na zamówienie klienta, bez wnikanego sposobu ich użycia. Podejścia te odzwierciedlają przeciwnie koncepcje obsługi doglądanej i niedoglądanej (odpowiednio – serwerów przechowujących stan i serwerów bezstanowych).

Zwielokrotnienie plików na różnych maszynach tworzy użytkownika nadmiar, polepszający dostępność danych. Istnienie kopii plików w różnych maszynach poprawia również wydajność, ponieważ do spełnienia zamówienia na dostęp można wybrać kopię najbliższą, co skraca czas obsługi.

■ Ćwiczenia

- 17.1 Jakie są zalety rozproszonego systemu plików w porównaniu ze centralizowanym systemem plików?
- 17.2 Który z przykładowych rozproszonych systemów plików mógłby obsługiwać wielką, mającą wielu klientów bazę danych? Wyjaśnij swoją odpowiedź.

- 17.3 W jakich warunkach klient preferowałby rozproszony system plików, przezroczysty pod względem położenia? Kiedy warto wybrać rozproszony system plików odznaczający się cechą niezależności pod względem położenia? Podaj przyczyny takiego czy innego wyboru.
- 17.4 Jakie aspekty systemu rozproszonego warto by wybrać do systemu pracującego w całkowicie niezawodnej sieci?
- 17.5 Porównaj przeciwnie cechy metody przechowywania bloków dyskowych w lokalnej pamięci podręcznej u klienta i w zdalnej pamięci podręcznej serwera.
- 17.6 Jakie korzyści wynikają z odwzorowywania obiektów w pamięci wirtualnej, jak to czyni system Apollo Domain? Jakie z tego wynikają kłopoty?

Uwagi bibliograficzne

Rozproszony system plików oparty na optymistycznym sterowaniu wspólnością opisali Mullender i Tanenbaum [304]. Omówienie zagadnień spójności i sterowania rekonstrukcją zwielokrotnionych plików przedstawili Davcev i Burkhard [92]. Zarządzanie zwielokrotnionymi plikami w środowisku systemu UNIX rozpatrywali Brereton [53] oraz Purdin i in. [337]. Wah w artykule [437] rozważył zagadnienie umiejscowienia plików w rozproszonych systemach komputerowych.

System UNIX United przedstawili Brownbridge i in. w artykule [57]. System Locus omówili Popek i Walker [333]. System Sprite został opisany przez Ousterhouta i in. [320] oraz przez Nelsona i in. [308]. Prezentację sieciowego systemu plików NFS firmy Sun Microsystems można znaleźć w pracach Sandberga i in. [368], Sandberga [368], a także w firmowym opracowaniu Sun Microsystems [410]. System Andrew przedstawili: Morris i in. [297], Howard i in. [181] oraz Satyanarayanan [375]. W materiałach konferencyjnych [242] Leach i in. omówili system Apollo Domain.

Szczegółowy przegląd głównie scentralizowanych serwerów plików po dała Svobodowa w [412]. W artykule tym położono nacisk na środki realizacji transakcji niepodzielnych, w mniejszym stopniu zaś na przezroczystość położenia i nazewnictwo.

Rozdział 18

KOORDYNACJA ROZPROSZONA

W rozdziale 6 opisaliśmy różne mechanizmy umożliwiające procesom synchronizowanie swoich działań. Przedstawiliśmy również kilka schematów zapewniania cechy niepodzielności transakcji działającej w odizolowaniu lub współbieżnie z innymi transakcjami. W rozdziale 7 opisaliśmy rozmaite metody, za pomocą których system operacyjny może radzić sobie z zakleszczeniami. W tym rozdziale przeanalizujemy, w jaki sposób scentralizowany mechanizm synchronizacji można rozszerzyć na system rozproszony. Omówimy tu także różne metody postępowania z zakleszczeniami w systemie rozproszonym.

18.1 ■ Porządkowanie zdarzeń

W systemie scentralizowanym jest zawsze możliwe określenie kolejności, w której wystąpiły dwa zdarzenia, ponieważ jest w nim jedna wspólna pamięć i zegar. W wielu zastosowaniach możliwość określenia porządku jest niezmiernie ważna. Na przykład przy przydzielaniu zasobów określa się, że zasób może zostać użyty tylko po uzyskaniu zgody na jego przydzielanie. Jednak w systemie rozproszonym nie ma wspólnej pamięci ani wspólnego zegara. Niekiedy jest więc niemożliwe stwierdzenie, które z dwóch zdarzeń wystąpiło najpierw. Relacja *uprzedniości zdarzeń* określa tylko częściowy porządek zdarzeń w systemach rozproszonych. Ponieważ możliwość zdefiniowania porządku całkowitego jest decydującą w wielu zastosowaniach, prezentujemy algorytm rozproszony, który rozszerza relację *uprzedniości zdarzeń* na spojne i całkowite uporządkowanie wszystkich zdarzeń w systemie.

18.1.1 Relacja uprzedniości zdarzeń

Ponieważ rozważamy tylko procesy sekwencyjne, wszystkie zdarzenia występujące w pojedynczym procesie są całkowicie uporządkowane. Również komunikaty, zgodnie z prawem przyczynowości, mogą być odebrane dopiero po ich wysłaniu. Mozemy wobec tego zdefiniować *relację uprzedniości zdarzeń* (ang. *happened-before relation*), oznaczaną za pomocą znaku \rightarrow , na zbiorze zdarzeń (zakładając, że zarówno wysłanie, jak i odebranie komunikatu stanowi zdarzenie) w sposób następujący:

1. Jeśli A i B są zdarzeniami w tym samym procesie i A zostało wykonane przed B, to $A \rightarrow B$.
2. Jeśli A jest zdarzeniem wysłania komunikatu przez pewien proces i B jest zdarzeniem odebrania tego komunikatu przez inny proces, to $A \rightarrow B$.
3. Jeśli $A \rightarrow B$ i $B \rightarrow C$, to $A \rightarrow C$.

Ponieważ zdarzenie nie może wystąpić przed samym sobą, relacja \rightarrow jest przeciwwrotnym porządkiem częściowym.

Jeśli dwa zdarzenia, A i B, nie są związane relacją \rightarrow (tzn. A nie wystąpiło przed B ani B nie wystąpiło przed A), to mówimy, że takie dwa zdarzenia wystąpiły *współbieżnie* (ang. *concurrently*). W tej sytuacji żadne ze zdarzeń nie może przyczynowo oddziałać na drugie. Jeśli jednak $A \rightarrow B$, to jest możliwe, że zdarzenie A wpłynie przyczynowo na zdarzenie B.

Definicje współbieżności i *uprzedniości zdarzeń* można najlepiej zilustrować za pomocą diagramu czasoprzestrzennego, takiego jak na rys. 18.1. Kierunek poziomy reprezentuje przestrzeń (tzn. różne procesy), a kierunek pionowy reprezentuje czas. Opatrzone etykietami pionowe linie symbolizują procesy (lub procesory). Kropki z etykietami oznaczają zdarzenia. Linia falista symbolizuje komunikat przesłany od jednego procesu do drugiego. Z takiego diagramu wy-



Rys. 18.1 Czas względny dla trzech procesów współbieżnych

nika w sposób oczywisty, że zdarzenia A i B są współbieżne wtedy i tylko wtedy, gdy nie istnieje żadna ścieżka z A do B ani z B do A.

Przeanalizujmy na przykład rys. 18.1. Wśród zdarzeń pozostających w relacji uprzedniości są tam takie, jak:

$$\begin{aligned} p_1 &\rightarrow q_2, \\ r_0 &\rightarrow q_4, \\ q_3 &\rightarrow r_1, \\ p_1 &\rightarrow q_4 \text{ (ponieważ } p_1 \rightarrow q_2 \text{ i } q_2 \rightarrow q_4\text{).} \end{aligned}$$

Niektórymi spośród zdarzeń współbieżnych w tym systemie są

$$\begin{aligned} q_0 &\text{ i } p_2, \\ r_0 &\text{ i } q_3, \\ r_0 &\text{ i } p_3, \\ q_3 &\text{ i } p_3. \end{aligned}$$

Nie jesteśmy w stanie określić, które ze zdarzeń współbieżnych, takich jak q_0 i p_2 , wystąpiło pierwsze. Ponieważ jednak żadne ze zdarzeń nie oddziaływa na drugie (żadne z nich nie może poznac, czy drugie już wystąpiło), nie jest istotne, które z nich faktycznie wystąpi najpierw. Ważne jest tylko, aby do wolne dwa procesy, dla których porządek dwu zdarzeń współbieżnych ma znaczenie, uzgodniły jakąś ich kolejność.

18.1.2 Implementacja

Aby móc określić, że zdarzenie A wystąpiło przed zdarzeniem B jest potrzebny wspólny zegar albo zbiór idelnie zsynchronizowanych zegarów. Ponieważ w systemie rozproszonym nic takiego nie występuje, należy zdefiniować relację *uprzedniości zdarzeń* bez posługiwania się zegarami fizycznymi.

Z każdym zdarzeniem systemowym kojarzymy znacznik czasu (ang. *timestamp*). Możemy wówczas zdefiniować warunek uporządkowania całkowitego (ang. *global ordering*): dla każdej pary zdarzeń A i B, jeżeli A → B, to znacznik czasu A jest mniejszy niż znacznik czasu B. Dalej przekonamy się, że zależność odwrotna nie musi być prawdziwa.

Jak można narzucić warunek globalnego uporządkowania w środowisku rozproszonym? W każdym procesie P_i definiujemy zegar logiczny ZL_i . Zegar logiczny można implementować w postaci prostego licznika, zwiększanego między wystąpieniami każdego dwojek kolejnych zdarzeń w procesie. Ponieważ wartość zegara logicznego rośnie monotonicznie, więc przypisuje on jednoznaczna liczbę do każdego zdarzenia: jeśli zdarzenie A poprzedza zdarzenie B

w procesie P_n , to $ZL_i(A) < ZL_i(B)$. Wartość logicznego zegara zdarzenia jest znacznikiem czasu tego zdarzenia. Widac, że schemat ten zapewnia dla każdych dwóch zdarzeń w tym samym procesie spełnienie warunku uporządkowania całkowitego.

Niestety, ten schemat nie gwarantuje, że warunek uporządkowania całkowitego będzie spełniony dla procesów. Aby zilustrować to zagadnienie, rozważmy dwa komunikujące się ze sobą procesy P_1 i P_2 . Założymy, że P_1 wysyla komunikat do P_2 (zdarzenie A) z $ZL_1(A) = 200$, a proces P_2 odbiera ten komunikat (zdarzenie B) z $ZL_2(B) = 195$ (ponieważ procesor procesu P_2 jest wolniejszy od procesora P_1 , więc jego zegar logiczny tyka wolniej). Sytuacja ta narusza nasz warunek, gdyż $A \rightarrow B$, lecz znacznik czasu A jest większy od znacznika czasu B.

W celu usunięcia tej trudności będziemy wymagali, aby proces przesunął swój zegar logiczny, gdy otrzyma komunikat, którego znacznik czasu jest większy niż bieżąca wartość jego zegara logicznego. A szczególnie, jeśli proces P_1 otrzymuje komunikat (zdarzenie B) ze znacznikiem czasu t i $ZL_1(B) \leq t$, to powinien przesunąć swój zegar tak, by $ZL_1(B) = t + 1$. Tak więc, gdy w naszym przykładzie proces P_2 otrzyma komunikat od P_1 , wówczas powinien przestawić swój zegar logiczny tak, aby $ZL_2(B) = 201$.

Na koniec, aby uzyskać uporządkowanie całkowite, powinniśmy tylko zacząć serwować, że w naszym schemacie porządkowania według znaczników czasu równość dwóch znaczników czasu dwóch zdarzeń A i B oznacza współbieżność tych zdarzeń. Aby pokonać tę przeszkodę i utworzyć uporządkowanie całkowite, można posłużyć się liczbami identyfikującymi procesy. Zastosowanie znaczników czasu omawiamy w dalszej części tego rozdziału.

18.2 ■ Wzajemne wykluczanie

Przedstawimy teraz kilka różnych algorytmów implementowania wzajemnego wykluczania w środowisku rozproszonym. Zakładamy, że system składa się z n procesów, z których każdy rezyduje w innym procesorze. Aby uprościć nasze rozważania, założymy, że procesy są jednoznacznie ponumerowane od 1 do n oraz że między procesami i procesorami istnieje odwzorowanie jeden do jednego (tzn. każdy proces ma własny procesor).

18.2.1 Podejście scentralizowane

W podejściu scentralizowanym w celu zapewnienia wzajemnego wykluczania jeden z procesów w systemie zostaje wybrany do koordynowania wejść do sekcji krytycznej. Każdy proces, który chce spowodować wzajemne wyklu-

czanie wysyła komunikat z *zamówieniem* (ang. *request*) do koordynatora. Kiedy proces otrzyma od koordynatora komunikat z *odpowiedzią* (ang. *reply*), wtedy może wejść do swojej sekcji krytycznej. Po wyjściu z sekcji krytycznej proces wysyła do koordynatora komunikat *zwalniający* (ang. *release*) i kontynuuje działanie.

Po otrzymaniu komunikatu z zamówieniem koordynator sprawdza, czy jakiś inny proces jest w swojej sekcji krytycznej. Jeśli żaden proces nie znajduje się w sekcji krytycznej, to koordynator natychmiast wysyła komunikat z odpowiedzią. W przeciwnym razie zamówienie zostaje włączone do kolejki. Gdy koordynator otrzyma komunikat o zwolnieniu, wówczas usuwa jeden z komunikatów z zamówieniami z kolejki (zgodnie z pewnym algorytmem planowania) i wysyła komunikat z odpowiedzią do procesu zamawiającego.

Powinno być jasne, że ten algorytm gwarantuje wzajemne wykluczanie. Ponadto, jeśli zasady planowania wewnątrz koordynatora są sprawiedliwe (np. planowanie w trybie „pierwszy zgłoszony, pierwszy obsłużony”), to nie będzie występować głodzenie. Opisany schemat wymaga trzech komunikatów na wejściu do sekcji krytycznej: *zamówienia, odpowiedzi i zwolnienia*.

Jeśli proces koordynujący ulegnie awarii, to jego miejsce musi zająć nowy proces. W punkcie 18.6 opisujemy różne algorytmy wyboru nowego, jednoznacznego koordynatora. Gdy nowy koordynator zostanie wybrany, musi zebrać wszystkie procesy w systemie w celu zrekonstruowania kolejki zamówień. Po odtworzeniu kolejki można powrócić do obliczeń.

18.2.2 Podejście w pełni rozproszone

Jeśli chcemy rozłożyć podejmowanie decyzji na cały system, to rozwiązanie będzie o wiele bardziej skomplikowane. Zaprezentujemy teraz algorytm oparty na schemacie porządkowania zdarzeń opisany w p. 18.1.

Gdy proces P_i chce wejść do sekcji krytycznej, wówczas wytwarza nowy znacznik czasu ZC i wysyła komunikat *zamówienie(P_i , ZC)* do wszystkich innych procesów w systemie (także do siebie). Po otrzymaniu komunikatu zamawiającego proces może odpowiedzieć natychmiast (tj. wysłać komunikat z odpowiedzią do P_i) lub może zwlekać z wysłaniem odpowiedzi (bo np. właśnie znajduje się w sekcji krytycznej). Proces, który otrzyma komunikat z odpowiedzią od wszystkich innych procesów w systemie, może wejść do swojej sekcji krytycznej, powodując ustawianie nadchodzących zamówień w kolejce i ich odwlekanie. Po opuszczeniu sekcji krytycznej proces wysyła komunikaty z *odpowiedziami* na wszystkie opóźniane zamówienia.

Decyzja o natychmiastowej odpowiedzi procesu P_i na komunikat *zamówienie(P_i , ZC)* lub zwłoki w jej udzieleniu zależy od trzech czynników:

1. Jeśli proces P_i przebywa w sekcji krytycznej, to opóźnia odpowiedź do P_j .
2. Jeśli proces P_i nie chce wejść do sekcji krytycznej, to odpowiedź do P_j wysyła natychmiast.
3. Jeśli proces P_i chce wejść do sekcji krytycznej, lecz jeszcze do niej nie wszedł, to porównuje znacznik czasu swojego zamówienia ze znacznikiem czasu ZC zamówienia, które nadeszło od procesu P_j . Jeśli jego znacznik czasu jest większy od ZC , to natychmiast wysyła odpowiedź do P_j (P_i poprosił pierwszy). W przeciwnym razie zwleka z odpowiedzią.

Algorytm ten wykazuje następujące pożądane cechy:

- Uzyskuje się wzajemne wykluczanie.
- Jest zapewnione niewystępowanie zakleszczeń.
- Nie grozi głodzenie, ponieważ wejście do sekcji krytycznej jest planowane według znaczników czasu. Uporządkowanie według znaczników czasu gwarantuje obsługę procesów w porządku „pierwszy zgłoszony – pierwszy obsłużony”.
- Liczba komunikatów potrzebnych do wejścia do sekcji krytycznej wynosi $2 \times (n - 1)$. Jest to minimalna liczba komunikatów wymaganych do wejścia do sekcji krytycznej w warunkach niezależnego i wspólniebieżnego działania procesów.

W celu zilustrowania, jak działa opisany algorytm, rozważymy system składający się z procesów P_1 , P_2 i P_3 . Założymy, że procesy P_1 i P_3 chcą wejść do swoich sekcji krytycznych. Proces P_1 wysyła wtedy komunikat $zamówienie(P_1, \text{znacznik czasu} = 10)$ do procesów P_2 i P_3 , a proces P_3 wysyła komunikat $zamówienie(P_3, \text{znacznik czasu} = 4)$ do procesów P_1 i P_2 . Znaczniki czasu 4 i 10 otrzymano z zegarów logicznych opisanych w p. 18.1. Kiedy proces P_2 otrzyma komunikaty z zamówieniami, wtedy odpowie natychmiast. Gdy proces P_1 otrzyma zamówienie od procesu P_3 , wówczas odpowie natychmiast, gdyż jego własne znacznik czasu (10) jest większy niż znacznik czasu procesu $P_3(4)$. Gdy proces P_1 otrzyma zamówienie od procesu P_1 , wówczas opóźni swoją odpowiedź, ponieważ znacznik czasu w jego komunikacie z zamówieniem (4) jest mniejszy niż znacznik czasu komunikatu procesu P_1 (10). Po otrzymaniu odpowiedzi zarówno od procesu P_1 , jak i od procesu P_3 , proces P_2 może wejść do sekcji krytycznej. Po wyjściu z sekcji krytycznej proces P_2 wysyła odpowiedź do procesu P_1 , który będzie mógł wtedy wejść do sekcji krytycznej.

Zauważmy, że ten schemat wymaga udziału wszystkich procesów w systemie. Podejście to ma trzy niepożądane konsekwencje:

1. Procesy muszą znać identyfikatory wszystkich innych procesów w systemie. Jeśli nowy proces dołącza do grupy procesów uczestniczących w algorytmie wzajemnego wykluczania, to należy podjąć następujące działania:

- (a) Proces musi otrzymać nazwy wszystkich innych procesów w grupie.
- (b) Nazwa nowego procesu musi być przekazana wszystkim innym procesom w grupie.

Zadanie to nie jest takie banalne, jakby się wydawało, ponieważ w chwili, gdy nowy proces dołącza do grupy, w systemie mogą już krążyć jakieś komunikaty z zamówieniami lub odpowiedziami. Więcej szczegółów można znaleźć w literaturze przytoczonej na końcu rozdziału.

2. Jeśli jeden z procesów ulegnie awarii, to załamuje się cały schemat. Kłopot ten rozwiązuje się przez stałe nadzorowanie stanu wszystkich procesów w systemie. Jeśli jakiś proces ulega awarii, to wszystkie inne procesy są o tym powiadamiane, aby nie wysyłały dalej komunikatów z zamówieniami do procesu uszkodzonego. Gdy proces zostanie reaktywowany, musi wówczas zainicjować procedurę umożliwiającą mu ponowne dołączenie do grupy.
3. Procesy, które nie weszły do sekcji krytycznych, muszą często generować przerwania, aby upewniać inne procesy o swoim zamiarze wejścia do sekcji krytycznej. Z tych względów przedstawiony protokół jest odpowiedni dla małych, stabilnych zbiorów współpracujących procesów.

18.2.3 Metoda przekazywania żetonu

Inną metodą zapewniania wzajemnego wykluczania jest krążenie żetonu między procesami systemu. *Żeton* (ang. *token*) jest specjalnym rodzajem komunikatu przekazywanym w obrębie systemu. Posiadanie żetonu upoważnia do wejścia do sekcji krytycznej. Ponieważ w systemie jest tylko jeden żeton, więc w danej chwili tylko jeden proces może znaleźć się w sekcji krytycznej.

Założymy, że procesy w systemie są logicznie zorganizowane w strukturę pierścieniową. Fizyczna sieć komunikacyjna nie musi mieć kształtu pierścienia. Gdy tylko procesy mają ze sobą połączenie, można implementować pierścień logiczny. Aby realizować wzajemne wykluczanie, przekazuje się do pierścienia żeton. Gdy proces otrzyma żeton, może wejść do sekcji krytycz-

nej, przechowując żeton. Po wyjściu procesu z sekcji krytycznej, żeton znów zaczyna krążyć. Jeśli proces otrzymujący żeton nie chce wchodzić do sekcji krytycznej, to przekazuje żeton sąsiadowi. Schemat ten jest podobny do algorytmu 1 w rozdz. 6, z tym że funkcję zmiennej dzielonej pełni tu żeton.

Ponieważ istnieje tylko jeden żeton, więc tylko jeden proces może przebywać w sekcji krytycznej w danym czasie. Ponadto, jeśli pierścień jest jednokierunkowy, to jest gwarantowane unikanie głodzenia. Liczba komunikatów wymaganych do zrealizowania wzajemnego wykluczania może wachać się od jednego komunikatu na wejście – w przypadku dużego współzawodnictwa – do nieskończonej liczby komunikatów przy słabej rywalizacji (tzn. gdy żaden proces nie chce wchodzić do sekcji krytycznej).

Należy wziąć pod uwagę dwa typy awarii. Po pierwsze, jeśli żeton zostanie zagubiony, to trzeba dokonać elekcji w celu wytworzenia nowego żetona. Po drugie, jeśli proces ulegnie uszkodzeniu, to należy zbudować nowy pierścień logiczny. Istnieje kilka różnych algorytmów elekcji i rekonstrukcji pierścienia logicznego. W punkcie 18.6 przedstawiamy pewien algorytm elekcji. Opracowanie algorytmu rekonstrukcji pierścienia pozostawiamy czytelnikowi w postaci cw. 18.6.

18.3 ■ Niepodzielność

W rozdziale 6 wprowadziliśmy pojęcie transakcji niepodzielnej, czyli jednostki programu, która musi być wykonana *niepodzielnie* (ang. *atomically*). Należy przez to rozumieć, że wykonuje się wszystkie operacje wchodzące w skład takiej jednostki albo nie wykonuje się żadnej z nich. W porównaniu z systemem skoncentrowanym w systemie rozproszonym zapewnienie cechy niepodzielności transakcji znacznie się komplikuje. Trudności wynikają stąd, że w wykonaniu jednej transakcji może brać udział wiele stanowisk. Awaria któregoś z tych stanowisk lub uszkodzenie połączenia komunikacyjnego między nimi może spowodować bezsensowne obliczenia.

W systemie rozproszonym zapewnienie niepodzielności różnorodnych transakcji należy do zadań *koordynatora transakcji* (ang. *transaction coordinator*). Na każdym stanowisku działa lokalny koordynator transakcji, odpowiadający za koordynowanie wszystkich transakcji rozpoczętych na danym stanowisku. W odniesieniu do każdej takiej transakcji obowiązki koordynatora są następujące:

- rozpoczęwanie wykonania transakcji;
- dzielenie transakcji na pewną liczbę podtransakcji i rozmieszczanie tych podtransakcji na odpowiednich stanowiskach w celu wykonania;

- koordynowanie zakończenia transakcji, które może doprowadzić do zatwierdzenia transakcji na wszystkich stanowiskach lub do jej zaniechania.

Zakładamy, że na każdym lokalnym stanowisku jest utrzymywany *rejestr* (ang. *log*)^{*} na wypadek rekonstrukcji po awarii.

18.3.1 Protokół zatwierdzania dwufazowego

W celu zapewnienia niepodzielności transakcji T wszystkie stanowiska, na których jest ona wykonywana, muszą uzgodnić końcowy rezultat tych działań. Transakcja T musi być przez wszystkie stanowiska albo zatwierdzona, albo zaniechana. Aby to osiągnąć, koordynator transakcji musi w odniesieniu do transakcji T wykonać *protokół zatwierdzania* (ang. *commit protocol*). Do najprostszych i najczęściej stosowanych protokołów zatwierdzania należy *protokół zatwierdzania dwufazowego* (ang. *two-phase commit – 2PC*). Zajmiemy się teraz jego omówieniem.

Niech T oznacza transakcję rozpoczętą na stanowisku S , i niech K oznacza koordynatora transakcji na stanowisku S . Gdy transakcja kończy działanie, tzn. gdy wszystkie stanowiska, na których T była wykonywana, poinformują koordynatora K , że zakończyły związanego z nią pracę, wówczas koordynator K rozpoczęta wykonanie protokołu 2PC.

- Faza 1:** Koordynator K dopisuje do rejestru uwagę <przygotuj T > i nakazuje umieścić ją w pamięci trwałej. Następnie wysyła komunikat „przygotuj T ” do wszystkich stanowisk, na których wykonywano transakcję T . Po otrzymaniu takiego komunikatu zarządcę transakcji na danym stanowisku rozstrzyga, czy ma zamiar zatwierdzić swoją część transakcji T . Jeśli odpowiedź brzmi „nie”, to dodaje do rejestru rekord <nie T >, po czym odpowiada, wysyłając do K komunikat „zaniechaj T ”. Jesli odpowiedź brzmi „tak”, to zarządcę transakcji dopisuje do rejestru rekord < T jest gotowa> i powoduje zapamiętanie w pamięci trwałej wszystkich rekordów rejestru dotyczących T . Zarządcę transakcji wysyła wtedy do koordynatora K komunikat „gotowa T ”.
- Faza 2:** Gdy koordynator K otrzyma odpowiedź na komunikat „przygotuj T ” od wszystkich stanowisk lub gdy od chwili wysłania komunikatu „przygotuj T ” upłynie określony z góry czas, wówczas K może podjąć decyzję o zatwierdzeniu lub zaniechaniu transakcji T . Transakcja T może być zatwierdzona, jeśli K , otrzymał komunikat o gotowości do jej zatwierdzenia od wszystkich uczestniczących w niej stanowisk. W przeciwnym razie na-

* Inaczej: *dziennik transakcji*. – Przyp. tłum.

leży transakcji T zaniechać. W zależności od werdyktu wpisuje się do rejestru adnotację <zatwierdzaj T > albo <zaniechaj T > i zapamiętuje to w pamięci trwałej. W tej chwili los transakcji został przesądzony. Teraz koordynatorowi pozostaje już tylko wysłać komunikat o zatwierdzeniu T lub o jej zaniechaniu do wszystkich uczestniczących w niej stanowisk. Po otrzymaniu komunikatu stanowisko zapisuje go w rejestrze.

Stanowisko, na którym wykonywano transakcję T , może bezwarunkowo jej zaniechać w dowolnej chwili poprzedzającej wysłanie do koordynatora komunikatu o gotowości transakcji. Komunikat „gotowa T ” jest zatem zobowiązaniem ze strony stanowiska do zastosowania się do polecenia, zatwierdzenia transakcji T lub jej zaniechania, wydanego przez koordynatora. Stanowisko może złożyć taką obietnicę tylko wtedy, kiedy potrzebne informacje są zapamiętane w pamięci trwałej. W przeciwnym razie, gdyby doszło do awarii stanowiska po wysłaniu komunikatu o gotowości transakcji T , dotrzymanie obietnicy mogłoby okazać się niemożliwe.

Ponieważ do zatwierdzenia transakcji jest potrzebna jednogłośność, jej los przesąduje się, gdy choć jedno stanowisko odpowie „zaniechaj T ”. Ponieważ stanowisko S , koordynatora jest jednym z tych, na których transakcja T była wykonywana, koordynator może zdecydować jednostronnie o zaniechaniu transakcji. Ostateczna decyzja w sprawie transakcji zapada w chwili, gdy koordynator zapisuje werdykt (zatwierdzenie lub zaniechanie) w rejestrze i powoduje zapamiętanie go w pamięci trwałej. W niektórych implementacjach protokołu 2PC stanowisko na koniec drugiej fazy protokołu wysyła do koordynatora komunikat potwierdzenia transakcji T . Po otrzymaniu komunikatu potwierdzenia T od wszystkich stanowisk koordynator dopisuje do rejestru rekord < T zakończona>.

18.3.2 Działanie protokołu 2PC w przypadku awarii

Omówimy teraz szczegóły reagowania protokołu 2PC na różnego rodzaju awarie. Jak się przekonamy, podstawową wadą protokołu 2PC jest to, że awaria koordynatora może spowodować blokowanie i zwłokę z podjęciem decyzji o zatwierdzeniu lub zaniechaniu transakcji T , przeciągającą się do chwili przywrócenia koordynatora K , do pracy.

18.3.2.1 Awaria stanowiska uczestniczącego w transakcji

Gdy stanowisko biorące udział w transakcji jest przywracane do pracy po awarii, wówczas musi ono sprawdzić swój rejestr, aby określić los transakcji, które były wykonywane w chwili wystąpienia awarii. Niech T oznacza jedną z takich transakcji. Pod uwagę jest brany każdy z trzech możliwych przypadków:

- Rejestr zawiera rekord <zatwierdzaj T >. W tym przypadku stanowisko wykonuje operację **przywróć(T)**.
- Rejestr zawiera rekord <zaniechaj T >. W tym przypadku stanowisko wykonuje operację **wycofaj(T)**^{*}.
- Rejestr zawiera rekord < T jest gotowa>. W tym przypadku stanowisko musi skonsultować się w sprawie losu transakcji T z koordynatorem K . Jeżeli K działa, to powiadomi stanowisko S_k o tym, czy transakcję zatwierdzono czy zaniechano. W pierwszym przypadku stanowisko wykoną operację **przywróć(T)**, w drugim – **wycofaj(T)**. Jeśli koordynator K jest wyłączony, to stanowisko S_k musi spróbować określić los T , konsultując się z innymi stanowiskami. Robi to za pomocą komunikatu **pytanie-o-stan** (ang. *query-status*) wysyłanego w związku z transakcją T do wszystkich stanowisk systemu. Stanowisko, które otrzyma taki komunikat, musi sprawdzić swój rejestr, aby określić, czy transakcja była w nim wykonywana i jeśli tak, to czy zakończyła się zatwierdzeniem czy zaniechaniem; dowiedziawszy się tego, powiadomia stanowisko S_k . Jeśli żadne ze stanowisk nie ma odpowiedniej informacji (o zatwierdzeniu lub zaniechaniu transakcji), to stanowisko S_k ani nie wycofuje transakcji T , ani jej nie zatwierdza. Decyzja dotycząca T zostaje odłożona do czasu, aż stanowisko S_k będzie mogło otrzymać potrzebną informację. Stanowisko S_k musi zatem okresowo ponawiać wysyłanie komunikatu **pytanie-o-stan** do innych stanowisk. Robi to dopóty, dopóki nie stwierdzi, że dysponuje już potrzebną informacją. Zauważmy, że stanowisko, na którym działa koordynator K , zawsze ma potrzebne informacje.
- W rejestrze nie ma żadnych informacji kontrolnych (zaniechanie, zatwierdzenie, gotowość) dotyczących transakcji T . Z nieobecnością danych kontrolnych wynika, że stanowisko S_k uległo awarii przed udzieleniem odpowiedzi na komunikat „przygotuj T ” wysłany przez K . Ponieważ awaria S_k uniemożliwiła wysłanie odpowiedzi na ten komunikat, zgodnie z naszym algorytmem koordynator K musiał zaniechać transakcji T . Z tego powodu stanowisko S_k musi wykonać operację **wycofaj(T)**.

18.3.2.2 Awaria koordynatora

Jeżeli podczas wykonywania protokołu zatwierdzania transakcji T wystąpi awaria koordynatora, to o losie transakcji muszą zadecydować stanowiska biorące udział w jej wykonaniu. Zobaczmy, że w pewnych przypadkach stanowiska te nie będą w stanie rozstrzygnąć o tym, czy zatwierdzić transak-

* Oryginalne nazwy operacji: *redo* i *undo* (zob. p. 6.9.2.). – Przyp. tłum.

cję T czy też jej zaniechać. Z konieczności będą więc musiały poczekać na przywrócenie uszkodzonego koordynatora do pracy.

- Jeśli czynne stanowisko zawiera w rejestrze rekord <zatwierdzaj T >, to transakcję T należy zatwierdzić.
- Jeśli czynne stanowisko zawiera w rejestrze rekord <zaniechaj T >, to transakcji T należy zaniechać.
- Jeśli ktoś z czynnych stanowisk nie ma w rejestrze rekordu < T jest gotowa>, to uszkodzony koordynator K , nie mógł podjąć decyzji o zatwierdzeniu T . Można to wywnioskować na tej podstawie, że stanowisko, które nie ma w rejestrze rekordu < T jest gotowa> nie mogło wysłać do koordynatora K komunikatu „gotowa T ”. Koordynator mógł zadecydować o zaniechaniu transakcji T , ale nie o jej zatwierdzeniu. Zamiast oczekiwania na reaktywowanie koordynatora K , zaleca się zaniechanie transakcji T .
- Jeśli nie występuje żadna z poprzednich sytuacji, to wszystkie stanowiska muszą mieć w swoich rejestrach rekord < T jest gotowa> i tylko ten – bez żadnych innych rekordów kontrolnych (<zaniechaj T > lub <zatwierdzaj T >). Skoro koordynator uległ uszkodzeniu, to do czasu jego naprawienia nie można określić, czy – i jaką – podjęto decyzję. Aktywne stanowiska muszą więc czekać na wznowienie pracy koordynatora K . Ponieważ los transakcji T pozostaje wątpliwy, może ona utrzymywać zasoby systemowe. Jeśli na przykład stosuje się blokowanie (czyli zajmowanie zasobów), to transakcja T może blokować dane na czynnych stanowiskach. Jest to sytuacja niepożądana, ponieważ zanim koordynator K powróci do działania, mogą minąć godziny i dni. W tym czasie inne transakcje mogą być zmuszone do czekania na transakcję T . Doprzewodzi to do tego, że dane stają się niedostępne nie tylko na uszkodzonym stanowisku K_n , lecz także na stanowiskach czynnych. Ilość niedostępnych danych wzrasta wraz z wydłużającym się przestojem koordynatora K . Sytuacja taka nosi nazwę *problemu blokowania* (ang. *blocking*), ponieważ w trakcie przywracania do pracy stanowiska S , transakcja T pozostaje zablokowana.

18.3.2.3 Awaria sieci

Jeśli łącze ulega awarii, to wszystkie komunikaty, które były na trasie tego łącza nie docierają do miejsc docelowych w stanie nienaruszonym. Dla stanowisk podłączonych do takiego łącza wygląda to na awarię innych stanowisk. Dlatego można w tym przypadku postępować tak, jak opisano poprzednio.

Gdy uszkodzeniu ulega kilka łączy, wówczas może wystąpić podział sieci. W tym przypadku istnieją dwie możliwości. Koordynator wraz i innymi

uczestnikami transakcji może znaleźć się w jednej części; awaria nie wpływa wtedy na protokół zatwierdzania. Jeśli natomiast koordynator i jego kooperanci znajdują się w różnych częściach podzielonej sieci, to komunikaty między uczestnikami transakcji a jej koordynatorem przestają być wymieniane, co sprowadza ten przypadek do omówionej przed chwilą awarii łączna.

18.4 ■ Sterowanie współbieżnością

Pokażemy teraz, jak pewne schematy sterowania współbieżnością, omówione w rozdz. 6, dają się zmodyfikować do pracy w środowisku rozproszonym.

Zarządzanie wykonywaniem transakcji (lub podtransakcji), które korzystają z danych przechowywanych na lokalnym stanowisku, należy do obowiązków zarządcy transakcji (ang. *transaction manager*) systemu rozproszonej bazy danych. Zauważmy, że każda taka transakcja może być lokalna (tzn. działająca tylko na danym stanowisku) lub może być częścią transakcji globalnej (wykonywanej na kilku stanowiskach). Każdy zarządcę transakcji ma obowiązek utrzymywać rejestr na wypadek rekonstrukcji po awarii oraz pomocny w koordynowaniu wspólnego wykonywania transakcji na danym stanowisku. Jak się przekonamy, schematy sterowania współbieżnością z rozdz. 6 wymagają pewnych dostosowań przed użyciem w transakcjach rozproszonych.

18.4.1 Protokoły blokowania zasobów

Opisane w rozdz. 6 protokoły blokowania dwufazowego mogą być użyte w środowisku rozproszonym. Jedyną zmianą, jaką należy w nich wprowadzić, jest sposób implementacji zarządcy blokowania. W tym punkcie przedstawiamy kilka możliwych rozwiązań, z których pierwsze dotyczy przypadku, w którym nie zezwala się na zwielokrotnianie danych. Pozostałe schematy można stosować do ogólniejszego przypadku dopuszczającego zwielokrotnienie danych na kilku stanowiskach. Podobnie jak w rozdz. 6, założymy możliwość blokowania w trybie *wyłącznym* (ang. *exclusive*) i *wspólnym* (ang. *shared*).

18.4.1.1 Schemat bez zwielokrotnień

Jeśli w systemie nie ma zwielokrotniania danych, to schematem blokowania zasobów opisany w p. 6.9 można posłużyć się w sposób następujący. Każde stanowisko ma lokalnego zarządcę blokowania zasobów, który obsługuje zamówienia blokowania i odblokowywania odnoszące się do obiektów danych przechowywanych na danym stanowisku. W celu zablokowania obiektu danych Q na stanowisku S , transakcja wysyła po prostu komunikat do zarządcy blokowania na stanowisku S , żądając zablokowania zasobu (w określonym

trybie). Jeśli obiekt danych jest zablokowany w trybie nie dającym się pogodzić z nowym żądaniem, to realizacja zamówienia jest opóźniana do chwili, w której będzie możliwa. Kiedy stanie się wiadome, że zablokowanie zasobu jest wykonalne, wtedy zarządcą blokowania wyśle komunikat informujący inicjatora blokowania, że zamówienie zostało zrealizowane.

Zaletą tego schematu jest prostota implementacji. Do obsługi zamówienia blokowania zasobu są potrzebne dwa komunikaty, a do odblokowania zasobu wystarcza jeden komunikat. Jednakże wzrasta złożoność obsługi zakleszczeń. Ponieważ zamówienia blokowania i odblokowania nie są już wykonywane przez jedno stanowisko, różnorodne algorytmy postępowania z zakleszczeniami, omówione w rozdz. 7, muszą być zmienione – przedstawiamy to w p. 18.5.

18.4.1.2 Rozwiążanie z jednym koordynatorem

W rozwiązaniu z jednym koordynatorem na wybranym stanowisku – powiedzmy S_c – system utrzymuje jednego zarządcę blokowania. Wszystkie zamówienia blokowania i odblokowania są obsługiwane na stanowisku S_c . Zarządcą blokowania określa, czy zasób można zablokować natychmiast. Jeżeli tak, to wysyła komunikat z tą wiadomością do stanowiska, które poprosiło o zablokowanie zasobu. W przeciwnym razie zamówienie jest opóźniane do chwili, w której będzie je można zrealizować. Dopiero po oczekaniu na stosowny moment wysyła się do stanowiska inicjatora zamówienia blokowania odpowiednią wiadomość. Transakcja może czytać obiekt danych na dowolnym stanowisku, które zawiera jego kopię. W przypadku operacji pisania angażowane są wszystkie stanowiska przechowujące kopie obiektu danych.

Zalety tego schematu to:

- Prostota implementacji:** Schemat wymaga dwu komunikatów do obsługi zamówienia blokowania i jednego komunikatu do obsługi zamówienia odblokowania.
- Prostota obsługi zakleszczeń:** Ponieważ wszystkie zamówienia blokowania i odblokowania są obsługiwane na jednym stanowisku, można tu bezpośrednio stosować opisane w rozdz. 7 algorytmy postępowania z zakleszczeniami.

Do wad tego schematu należą m.in.:

- Wąskie gardło:** Stanowisko S_c staje się wąskim gardłem, gdyż musi przetwarzać wszystkie zamówienia.
- Podatność na awarie:** Jeśli stanowisko S_c ulegnie uszkodzeniu, to nadzorca wspólnieści przestaje istnieć. Przetwarzanie musi być wstrzymane lub trzeba podjąć działania naprawcze.

Kompromis między tymi zaletami a wadami można osiągnąć przez zastosowanie metody z wieloma koordynatorami (ang. *multiple-coordinator approach*), w której funkcję koordynatora blokowania pełni wiele stanowisk.

Każdy zarządcę blokowania zajmuje się obsługą zamówień blokowania i odblokowania w odniesieniu do podzbioru obiektów danych i przebywa na innym stanowisku. Takie rozproszenie zminniejsza zjawisko wąskiego gardła w odniesieniu do poszczególnych koordynatorów, lecz komplikuje obsługę zakleszczeń, gdyż zamówienia blokowania i odblokowania nie są wykonywane na jednym stanowisku.

18.4.1.3 Protokół większościowy

Protokół większościowy (ang. *majority protocol*)^{*} jest odmianą przedstawionego uprzednio schematu dotyczącego środowiska bcz zwielokrotnionych danych. System utrzymuje zarządcę blokowania na każdym stanowisku. Każdy zarządcę obsługuje zamówienia blokowania odnoszące się do wszystkich danych lub ich kopii, które przechowuje dane stanowisko. Jeśli transakcja chce zablokować obiekt danych Q , który jest zwielokrotniony na n różnych stanowiskach, to musi wysłać zamówienia blokowania do więcej niż połowy z tych n stanowisk, na których obiekt Q jest przechowywany. Każdy zarządcę blokowania rozstrzyga, czy zablokowanie może być wykonane natychmiast (w chwili rozpatrywania zamówienia). Tak jak poprzednio, z odpowiedzią zwleka się do czasu, kiedy zamówienie może być spełnione. Transakcja dopóty nie działa na obiekcie Q , dopóki nie uda jej się zablokować większości jego kopii.

W tym schemacie zwielokrotnione dane są nadzorowane w sposób rozproszony, dzięki czemu unika się wad sterowania centralnego. Ma on jednak własne niedoskonałości, do których należą:

- Implementacja:** Protokół większościowy jest bardziej złożony w implementacji niż poprzednie schematy. Wymaga on $2(n/2 + 1)$ komunikatów do obsługi zamówienia blokowania i $(n/2 + 1)$ komunikatów do obsługi zamówienia odblokowania.
- Obsługa zakleszczeń:** Ponieważ zamówienia blokowania i odblokowania nie są wykonywane na jednym stanowisku, algorytmy postępowania z zakleszczeniami należy zmodyfikować (zob. p. 18.5). Ponadto do zakleszczenia może dojść nawet wówczas, gdy jest blokowany tylko jeden obiekt danych. Aby to zilustrować, rozważmy system z czterema stanowiskami i pełnym zwielokrotnieniem. Przypuśćmy, że transakcje T_1 i T_2

* Protokół ten jest też znany pod nazwą *algorytmu głosowania* (ang. *voting algorithm*).
– Przyp. tłum.

chcą zablokować obiekt danych Q w trybie wyłącznym. Transakcji T_1 może się udać zablokowanie Q na stanowiskach S_1 i S_3 , natomiast transakcja T_2 może zdążyć zablokować obiekt Q na stanowiskach S_2 i S_4 . Wówczas każda z nich musi czekać na pozyskanie trzeciej blokady, a to już oznacza, że wystąpiło zakleszczenie.

18.4.1.4 Protokół tendencyjny

Protokół tendencyjny (ang. *biased protocol*) jest oparty na podobnym modelu jak protokół większościowy. Różnica polega na tym, że zamówienia na blokowanie do użytku wspólnego traktuje się łagodniej niż zamówienia blokowania na wyłączność. System utrzymuje zarządcę blokowania na każdym stanowisku. Każdy zarządcę obsługuje zamówienia blokowania dotyczące wszystkich obiektów danych na jego stanowisku. Blokowania wspólne i wyłączne odbywają się w różny sposób.

- **Blokowanie na użytk wstępny:** Transakcja chcącą zablokować obiekt danych Q zamawia po prostu jego zablokowanie u zarządcy blokowania na którymś ze stanowisk zawierających kopię obiektu Q .
- **Blokowanie na użytk wyłączny:** Transakcja, która chce zablokować obiekt danych Q , zamawia jego zablokowanie u wszystkich zarządców blokowania, których stanowiska zawierają obiekt Q .

Tak jak poprzednio, odpowiedź na zamówienie jest opóźniana do chwili, w której może ono być zrealizowane.

Zaletą takiego postępowania jest mniejsza niż w protokole większościowym pracochłonność operacji czytania. Jest to zaleta szczególnie istotna w typowych sytuacjach, w których liczba operacji czytania znacznie przewyższa liczbę operacji pisania. Dodatkowy nakład związany z pisaniem jest jednak wadą. Co więcej, udziałem protokołu tendencyjnego jest wada występująca w protokole większościowym, polegająca na złożoności obsługiwanego zakleszczeń.

18.4.1.5 Kopia podstawowa

W przypadku danych zwielokrotnionych możemy wybrać jedną z kopii jako podstawową (ang. *primary copy*). Przyjmuje się zatem, że dla każdego obiektu danych Q jego kopia podstawowa musi się znajdować tylko na jednym stanowisku, które nazywamy *stanowiskiem podstawowym obiektu Q*.

Transakcja, która potrzebuje zablokować obiekt danych Q , zamawia jego zablokowanie na stanowisku podstawowym tego obiektu. Jak poprzednio, odpowiedź na zamówienie jest opóźniana do chwili, w której zamówienie może być zrealizowane.

Tak więc metoda kopii podstawowej umożliwia sterowanie współbieżnością dostępu do zwielokrotnionych danych w sposób podobny do postępowania z danymi niezwielokrotnionymi. Metodę tę można łatwo zaimplementować. Jeśli jednak stanowisko przechowujące kopię podstawową Q ulegnie awarii, to obiekt Q będzie niedostępny nawet pomimo dostępności innych stanowisk zawierających jego kopię.

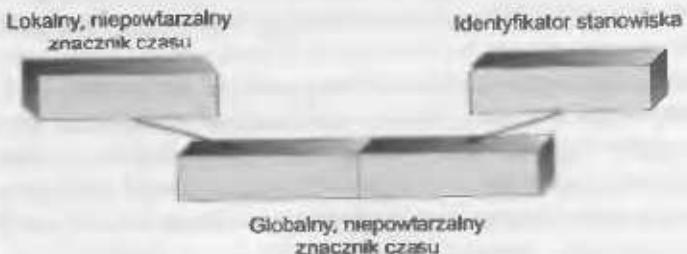
18.4.2 Zastosowanie znaczników czasu

Istota schematu z użyciem znaczników czasu, omówionego w p. 6.9, polega na nadawaniu każdej transakcji *niepowtarzanego* znacznika czasu, z którego korzysta się przy ustalaniu porządku uszeregowania transakcji. Pierwsze, co należy zrobić, uogólniając ten scentralizowany schemat na wariant rozproszony, to opracowanie metody wytwarzania niepowtarzalnych znaczników czasu. Po opracowaniu takiej metody nasze poprzednie protokoły można będzie w środowisku bez zwielokrotnień zastosować w postaci niezmienionej.

18.4.2.1 Wytwarzanie niepowtarzalnych znaczników czasu

Są dwie podstawowe metody generowania niepowtarzalnych znaczników czasu – scentralizowana i rozprosiona. W schemacie scentralizowanym wybiera się jedno stanowisko jako dystrybutora znaczników czasu. Stanowisko może w tym celu używać licznika logicznego lub własnego, lokalnego zegara.

W schemacie rozproszonym każde stanowisko wytwarza niepowtarzalny, lokalny znacznik czasu za pomocą licznika logicznego albo lokalnego zegara. Globalnie jednoznaczny znacznik czasu otrzymuje się przez skonkatenowanie niepowtarzanego, lokalnego znacznika czasu i identyfikatora stanowiska, który też musi być jednoznaczny (rys. 18.2). Porządek konkatenowania jest ważny! Identyfikator stanowiska jest umieszczany na mniej znaczącej pozycji, aby zapewnić, że globalne znaczniki czasu wytwarzane na jednym stanowisku nie zawsze będą większe niż znaczniki wygenerowane na innym.



Rys. 18.2 Wytwarzanie niepowtarzalnych znaczników czasu

Warto porównać ten sposób wytwarzania niepowtarzalnych znaczników czasu ze sposobem ich wytwarzania, który przedstawiliśmy w p. 18.1.2.

Jeśli jedno ze stanowisk generuje lokalne znaczniki czasu z większą częstością niż inne stanowiska, to nasze trudności się nie kończą. W takiej sytuacji licznik logiczny szybkiego stanowiska będzie przyjmował wartości większe niż wygenerowane na innych stanowiskach. Jest potrzebny mechanizm zapewniania, że lokalne znaczniki czasu będą wytwarzane sprawiedliwie w obrębie całego systemu. Aby umożliwić sprawiedliwe wytwarzanie znaczników czasu, na każdym stanowisku S_i definiujemy *zegar logiczny ZL_i*, który generuje lokalnie jednoznaczne znaczniki czasu. Zegar logiczny można zrealizować w postaci licznika, który jest zwiększany po wytworzeniu nowego, lokalnego znacznika czasu. W celu zagwarantowania, że różne zegary logiczne będą ze sobą zsynchronizowane, żądamy, aby stanowisko S_i przesuwało do przodu swój zegar logiczny zawsze wtedy, kiedy transakcja T_i o znaczniku czasu $\langle x, y \rangle$ odwiedza dane stanowisko i wartość x jest większa niż bieżąca wartość ZL_i. W takim przypadku stanowisko S_i przesuwa swój zegar logiczny do wartości $x + 1$.

Jeżeli do generowania znaczników czasu używa się zegara systemowego, to znaczniki czasu są przydzielane sprawiedliwie pod warunkiem, że zegar systemowy żadnego ze stanowisk nie spieszy się ani nie spóźnia. Ponieważ zegary mogą nie być wystarczająco dokładne, aby zapewnić, że żaden zegar nie będzie wyprzedzał ani pozostawał w tyle za innym zegarem, jest konieczne użycie metody podobnej do stosowanej w przypadku zegarów logicznych.

18.4.2.2 Metoda porządkowania według znaczników czasu

Podstawowy schemat działania ze znacznikami czasu, wprowadzony w p. 6.9, można w prosty sposób rozszerzyć na system rozproszony. Podobnie jak w odmianie skoncentrowanej, brak zabezpieczeń przed czytaniem przez transakcję nie zatwierdzonych obiektów danych może doprowadzić do wycofań kaskadowych. Aby wyeliminować wycofania kaskadowe, możemy połączyć elementarną metodę znaczników czasu, poznawaną w p. 6.9, z protokołem 2PC, opisany w p. 18.3, otrzymując protokół, który zapewnia szeregowalność bez wycofań kaskadowych. Opracowanie takiego algorytmu pozostawiamy czytelnikowi.

Opisany, elementarny schemat postępowania ze znacznikami czasu ma niepcządaną cechę: konflikty między transakcjami są usuwane przez wycofania, a nie przez organizowanie czekania. Aby złagodzić ten problem, można buforować operacje czytania i pisania (tzn. powodować ich opóźnianie) do momentów, w których nabieramy pewności, że można je wykonać bez ryzyka zaniechań. Operacja *czytaj(x)* wykonywana przez transakcję T_i , musi być opóźniana, jeśli istnieje transakcja T_j , która ma jeszcze nie wykonaną operację

$\text{pisz}(x)$ i $ZC(T_i) < ZC(T_j)$ *. Podobnie, operacja $\text{pisz}(x)$ w transakcji T_i musi być opóźniana, jeżeli istnieje transakcja T_j , która ma do wykonania operację $\text{czytaj}(x)$ albo $\text{pisz}(x)$ i $ZC(T_j) < ZC(T_i)$. Istnieją różne metody zapewniania tej właściwości. W jednej z nich – zwanej *zachowawczym schematem porządkowania według znaczników czasu* (ang. *conservative timestamp-ordering scheme*) – wymaga się, aby każde stanowisko utrzymywało kolejki czytania i pisania, złożone odpowiednio ze wszystkich zamówień czytania i pisania, które mają być na danym stanowisku wykonane, a które należy opóźniać, aby dotrzymać wyżej określonych warunków. Nie przytaczamy tutaj tego schematu, proponując czytelnikowi, żeby opracował go samodzielnie.

18.5 ■ Postępowanie z zakleszczeniami

Algorytmy zapobiegania zakleszczeniom, unikania zakleszczeń oraz ich wykrywania, które przedstawiliśmy w rozdz. 7, można rozszerzyć tak, aby nadawały się do stosowania w systemie rozproszonym. Poniżej opisujemy kilka takich algorytmów rozproszonych.

18.5.1 Zapobieganie zakleszczeniom

Algorytmy zapobiegania zakleszczeniom i ich unikania, przedstawione w rozdz. 7, mogą znaleźć zastosowanie również w systemie rozproszonym, pod warunkiem wprowadzenia do nich odpowiednich zmian. Można na przykład zastosować technikę zapobiegania zakleszczeniom opartą na uporządkowaniu zasobów, definiując po prostu *globalne* uporządkowanie zasobów systemowych. Oznacza to, że wszystkie zasoby w całym systemie otrzymują jednoznaczne numery, po czym proces może zamawiać zasób (z dowolnego procesora) oznaczony jednoznaczny numerem i tylko wtedy, gdy nie przetrzymuje zasobu, którego jednoznaczny numer jest większy niż i . Podobnie, jest możliwe uzycie w systemie rozproszonym algorytmu bankiera przez wyznaczenie jednego z procesów w systemie (*bankiera*) jako tego, który będzie utrzymywał informacje niezbędne do wykonywania algorytmu bankiera. Każde zamówienie na zasoby będzie musiało przechodzić przez algorytm bankiera.

Obu tych schematów można użyć w postępowaniu z zakleszczeniami w środowisku rozproszonym. Pierwszy schemat jest prosty do implementacji i wymaga bardzo małych nakładów. Równie łatwo można zrealizować drugi schemat, lecz może okazać się on zbyt kosztowny. Bankier może stać się wąskim gardłem, ponieważ liczba przychodzących i wychodzących od niego

* ZC = znacznik czasu. – Przyp. tłum.

komunikatów może być wielka. Schemat bankiera nie wydaje się więc odpowiedni do praktycznego zastosowania w systemie rozproszonym.

Przedstawimy teraz nowy schemat zapobiegania zakleszczeniu, oparty na porządkowaniu według znaczników czasu, z wywłaszczeniem zasobów. Dla prostoty rozważmy tylko przypadek z pojedynczymi egzemplarzami zasobów każdego typu.

W celu kontroli wywłaszczenia przypiszemy każdemu procesowi jednoznaczny numer priorytetu. Numery te będą decydowały o tym, czy proces P_i powinien czekać na proces P_j . Na przykład możemy pozwolić P_i czekać na P_j , jeśli P_i ma wyższy priorytet niż P_j ; w przeciwnym razie proces P_i zostanie usunięty z pamięci. Schemat ten zapobiega zakleszczeniom, gdyż dla każdej krawędzi $P_i \rightarrow P_j$ w grafie oczekiwania P_i ma wyższy priorytet niż P_j . Zatem nie jest możliwe istnienie cyku.

Wadą tego schematu jest możliwość występowania głodzenia. Pewne procesy z bardzo niskimi priorytetami mogą być stale usuwane z pamięci. Kłopotu tego daje się uniknąć za pomocą znaczników czasu. Każdy proces w systemie z chwilą utworzenia otrzymuje niepowtarzalny znaczek czasu. Zaproponowano dwa wzajemnie uzupełniające się schematy zapobiegania zakleszczeniom z użyciem znaczników czasu:

- **Czekanie albo śmierć (ang. wait-die):** W tej metodzie nie stosuje się wywłaszczeń. Jeśli proces P_i zamawia zasób aktualnie przetrzymywany przez proces P_j , to procesowi P_i pozwala się czekać tylko wtedy, gdy jego znaczek czasu jest mniejszy od znacznika czasu procesu P_j (tzn. P_i jest starszy niż P_j). W przeciwnym razie P_i jest usuwany z pamięci (umiera). Założymy na przykład, że procesy P_1 , P_2 i P_3 mają odpowiednio znaczniki czasu 5, 10 i 15. Jeśli P_1 zamówi zasób przetrzymywany przez P_2 , to P_1 będzie czekać. Jeśli P_3 zamówi zasób przetrzymywany przez P_2 , to P_3 zostanie usunięty z pamięci.
- **Zranienie albo czekanie (ang. wound-wait):** To podejście opiera się na technice wywłaszczeniowej i jest kontrapozycją dla schematu „czekanie albo śmierć”. Jeśli proces P_i zamówi zasób aktualnie przetrzymywany przez proces P_j , to P_i może czekać tylko wtedy, kiedy ma większy znaczek czasu niż P_j (tzn. P_i jest młodszy od P_j). W przeciwnym razie P_i jest usuwany z pamięci (P_i zostaje zraniony przez P_j). Powracając do naszego poprzedniego przykładu z procesami P_1 , P_2 i P_3 , jeśli P_1 zamówi zasób przetrzymywany przez P_2 , to zasób ten zostanie odebrany procesowi P_2 i P_2 zostanie wycofany. Jeśli P_3 zamówi zasób przetrzymywany przez P_2 , to P_3 czeka.

W obu schematach unika się głodzenia pod warunkiem, że proces usunięty z pamięci nic otrzyma nowego znacznika czasu. Ponieważ znaczniki

czasu stale rosną, proces, który został usunięty z pamięci, będzie w końcu miał najmniejszy znacznik czasu. Nie zostanie zatem wycofany ponownie. W sposobie działania obu schematów istnieją jednak istotne różnice.

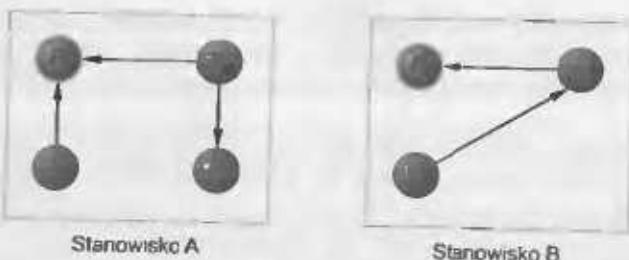
- W schemacie „czekanie albo śmierć” starszy proces musi czekać, aż młodszy zwolni zasoby. Zatem im proces jest starszy, tym większa jest jego tendencja do czekania. Dla porównania, w schemacie „zranienie albo czekanie” starszy proces nigdy nie czeka na młodszego procesu.
- Gdy w schemacie „czekanie albo śmierć” proces P_1 umiera i zostaje wysłany z pamięci, gdyż zapotrzebowal zasób przetrzymywany przez proces P_1 , wówczas P_1 z chwilą wznowienia może wydać ponownie ten sam ciąg zamówień. Jeśli z danego zasobu wciąż korzysta proces P_1 , to proces P_1 umiera ponownie. W ten sposób proces P_1 może umierać wielokrotnie, zanim uzyska potrzebny zasób. Skonfrontujmy ten ciąg zdarzeń z tym, co nastąpi w schemacie „zranienie albo czekanie”. Proces P_1 zostaje zraniony i usunięty z pamięci, ponieważ zasób, którym właśnie rozporządzał, zamówił proces P_1 . Kiedy P_1 zostanie wznowiony i zamówi zasób obecnie zajęty przez proces P_1 , wtedy P_1 czeka. Zatem w schemacie „zranienie albo czekanie” jest mniej wycofań procesów.

Istotnym problemem w obu schematach jest możliwość występowania niepotrzebnych usunięć procesów z pamięci.

18.5.2 Wykrywanie zakleszczenia

Algorytm zapobiegania zakleszczeniu może dokonywać wywłaszczeń zasobów nawet wtedy, gdy nie ma żadnego zakleszczenia. Aby zapobiec niepotrzebnym wywłaszczeniom, można używać algorytmu wykrywania zakleszczenia. Konstruuje się graf oczekiwania, opisujący stan przydziału zasobów. Ponieważ zakładamy tylko pojedyncze reprezentacje poszczególnych typów zasobów, cykl w grafie oczekiwania oznacza zakleszczenie.

Główym problemem w systemie rozproszonym jest decydowanie, jak postępować z grafem oczekiwania. Przybliżymy ten problem, opisując kilka popularnych sposobów jego rozwiązywania. Schematy te wymagają, aby każde stanowisko utrzymywało lokalny graf oczekiwania. Węzły grafu odpowiadają wszystkim tym procesom (zarówno lokalnym, jak i nielokalnym), które bieżąco utrzymują lub zamawiają dowolny zasób lokalny na danym stanowisku. Na przykład na rys. 18.3 mamy system złożony z dwu stanowisk, z których każde utrzymuje lokalny graf oczekiwania. Zauważmy, że procesy P_2 i P_3 występują w obu grafach, co wskazuje, że zamówili one zasoby na obu stanowiskach.



Rys. 18.3 Dwa lokalne grafy oczekiwania

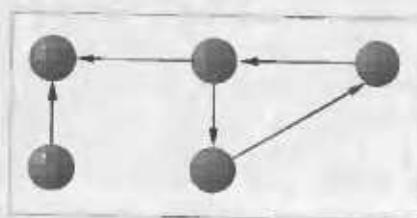
Lokalne grafy oczekiwania lokalnych procesów i zasobów są konstruowane w zwykły sposób. Gdy proces P_i na stanowisku A potrzebuje zasobu użytkowanego przez proces P_j na stanowisku B , wówczas P_i wysyła do stanowiska B komunikat z zamówieniem. Powoduje to dodanie do lokalnego grafu na stanowisku B krawędzi $P_i \rightarrow P_j$.

Jest oczywiste, że jeżeli jakikolwiek lokalny graf oczekiwania zawiera cykl, to wystąpiło zakleszczenie. Z drugiej strony fakt, że w żadnym lokalnym grafie oczekiwania nie ma cyklu, nie oznacza, że nie ma zakleszczeń. Aby zobrazować ten problem, rozważymy system przedstawiony na rys. 18.3. Każdy graf oczekiwania jest grafem bez cykli, niemniej jednak w systemie tym istnieje zakleszczenie. Aby udowodnić, że nie ma zakleszczenia, musimy wykazać, że suma obu grafów oczekiwania jest grafem bez cyklu. Graf (pokazany na rys. 18.4) otrzymany z sumowania obu grafów oczekiwania z rys. 18.3 rzeczywiście zawiera cykl, z czego wynika, że system jest w stanie zakleszczenia.

Istnieje kilka różnych metod organizacji grafu oczekiwania w systemie rozproszonym. Opisemy niektóre powszechnie stosowane schematy.

18.5.2.1 Podejście scentralizowane

W podejściu scentralizowanym z sumy wszystkich lokalnych grafów oczekiwania konstruuje się globalny graf oczekiwania. Jest on utrzymywany w jednym procesie – koordynatorze wykrywania zakleszczeń (ang. deadlock-



Rys. 18.4 Globalny graf oczekiwania do rys. 18.3

-detection coordinator). Ponieważ w systemie istnieje opóźnienie w komunikacji, należy rozróżnić dwa rodzaje grafów oczekiwania. Graf rzeczywisty opisuje faktyczny, lecz nieznany stan systemu w każdej chwili – taki, jakim widziałby go wszechwiedzący obserwator. Graf konstruowany jest przybliżeniem generowanym przez koordynatora podczas wykonywania jego algorytmu. Oczywiście, graf konstruowany musi być tak generowany, aby po każdym wywołaniu algorytmu wykrywania zakleszczenia raportowane wyniki były poprawne w tym sensie, że jeśli istnieje zakleszczenie, to oznajmia się o nim właściwie, a jeśli oznajmia się o zakleszczeniu, to system naprawdę jest w stanie zakleszczenia. Pokazemy, że skonstruowanie takich poprawnych algorytmów nie jest łatwe.

Są trzy różne możliwości (punkty czasowe), kiedy graf oczekiwania może być konstruowany:

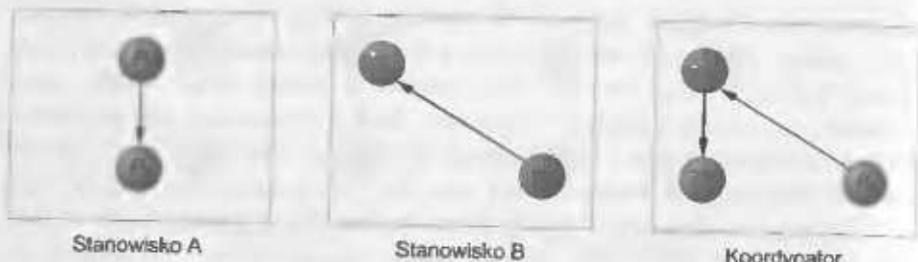
1. Zawsze, ilekroć do któregoś z lokalnych grafów oczekiwania wstawa się lub usuwa z niego nową krawędź.
2. Okresowo, gdy uzbiera się pewna liczba zmian w grafie oczekiwania.
3. Każdorazowo, gdy koordynator musi wywołać algorytm wykrywania cyklu.

Przeanalizujmy pierwszą możliwość. Kiedy wstawa się lub usuwa krawędź w lokalnym grafie, wtedy lokalne stanowisko jest także obowiązane wysłać komunikat do koordynatora, aby go powiadomić o tej zmianie. Po otrzymaniu takiego komunikatu koordynator uaktualnia graf globalny. Inną możliwością jest okresowe wysyłanie w jednym komunikacie informacji o kilku takich zmianach. Powracając do naszego poprzedniego przykładu, proces koordynujący będzie utrzymywał graf jak na rys. 18.4. Gdy stanowisko B wstawa do swojego lokalnego grafu oczekiwania krawędź $P_3 \rightarrow P_4$, wówczas wysyła także komunikat do koordynatora. Podobnie, gdy stanowisko A usuwa krawędź $P_3 \rightarrow P_1$, ponieważ P_1 zwolnił zasób, który był zamawiany przez proces P_3 , wówczas wysyła odpowiedni komunikat do koordynatora.

Gdy zostanie wywołany algorytm wykrywania zakleszczenia, wówczas koordynator przegląda swój graf globalny. W przypadku znalezienia cyklu, typuje się ofiarę do usunięcia z pamięci. Koordynator musi zawiadomić wszystkie stanowiska, że pewien konkretny proces został wybrany jako ofiara. W odpowiedzi na to stanowiska wycofują proces będący ofiarą.

Zauważmy, że w tym schemacie (możliwość 1) może dojść do zbędnego wycofywania procesów w wyniku dwu sytuacji:

- W globalnym grafie oczekiwania mogą występować *falszywe cykle* (ang. *false cycles*). Aby to zobrazować, rozpatrzmy migawkę systemu przed-



Rys. 18.5 Lokalne i globalne grafy oczekiwania

stawioną na rys. 18.5. Założymy, że P_1 zwalnia zasób, z którego korzystał na stanowisku A . Wskutek tego w A następuje usunięcie krawędzi $P_1 \rightarrow P_2$. Proces P_2 zamawia następnie zasób przetrzymywany przez P_3 na stanowisku B , co powoduje dodanie krawędzi $P_2 \rightarrow P_3$ w B . Jeśli komunikat *usun* $P_2 \rightarrow P_3$ z B dojdzie przed komunikatem *usuń* $P_1 \rightarrow P_2$ z A , to koordynator może wykryć fałszywy cykl $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$ po wstawieniu (lecz przed usunięciem). Mogą zostać podjęte czynności związane z usuwaniem zakleszczenia, chociaż do żadnego zakleszczenia nie doszło.

- Niepotrzebne wycofania mogą występować także wówczas, gdy zakleszczenie rzeczywiście wystąpiło i wskazano ofiarę, a jednocześnie w tym samym czasie zaniechano jakiegoś procesu z przyczyn nie związanych z zakleszczeniem (choćby z powodu przekroczenia przez proces przydziału czasu). Założymy na przykład, że stanowisko A z rys. 18.3 postanawia zaniechać procesu P_2 . W tym samym czasie koordynator odkrył występowanie cyklu i wyznaczył P_3 do roli ofiary. Zarówno P_2 , jak i P_3 będą wówczas wycofane na dysk, chociaż powinno się usunąć z pamięci tylko proces P_2 .

Zauważmy, że te same problemy odnoszą się do rozwiązań stosowanych w dwu pozostałych przypadkach (2 i 3).

Zaprezentujemy teraz decentralizowany algorytm wykrywania zakleszczenia, zastosowany do przypadku 3. Algorytm wykrywa wszystkie bieżące zakleszczenia i nie wykrywa zakleszczeń fałszywych. W celu uniknięcia zgłoszenia fałszywych zakleszczeń ząda się, aby do zamówień z różnych stanowisk były dodawane jednoznaczne identyfikatory (znaczniki czasu). Gdy proces P_i na stanowisku A zamawia zasób od P_j ze stanowiska B , wówczas jest posyłany komunikat zamawiający ze znacznikiem czasu ZC . Do lokalnego grafu oczekiwania w A wstawia się krawędź $P_i \rightarrow P_j$ z etykietą ZC . Krawędź ta jest wstawiana do lokalnego grafu oczekiwania w B tylko wtedy, gdy stanowisko B otrzymało komunikat z zamówieniem i nie może natychmiast udostępnić zamawianego zasobu. Zamówienie od P_i do P_j na tym samym

stanowisku jest traktowane w zwykły sposób; krawędzi $P_i \rightarrow P_j$ nie przypisuje się żadnego znacznika czasu. Algorytm wykrywania przedstawia się wtedy następująco:

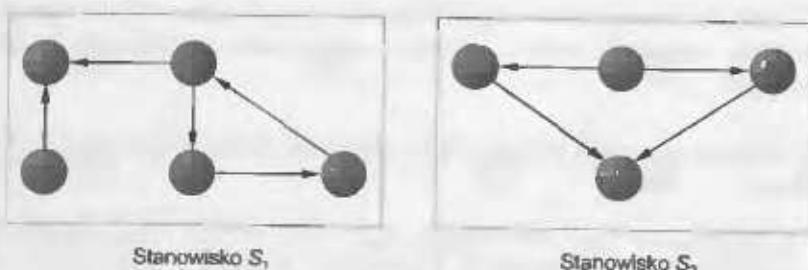
1. Koordynator wysyla komunikat inicjujący do każdego stanowiska w systemie.
2. Po otrzymaniu tego komunikatu stanowisko wysyla swój lokalny graf oczekiwania do koordynatora. Zauważmy, że każdy z tych grafów oczekiwania zawiera całą lokalną informację, którą stanowisko ma o stanie grafu rzeczywistego. Graf odzwierciedla stan stanowiska z danej chwili, lecz nie jest on zsynchronizowany z żadnym innym stanowiskiem.
3. Gdy koordynator otrzyma odpowiedzi od wszystkich stanowisk, wówczas konstruuje graf w sposób następujący:
 - (a) Konstruowany graf zawiera wierzchołek dla każdego procesu w systemie.
 - (b) Graf ma krawędź $P_i \rightarrow P_j$, wtedy i tylko wtedy, gdy (1) krawędź $P_i \rightarrow P_j$ istnieje w jednym z grafów oczekiwania lub (2) krawędź $P_i \rightarrow P_j$ z jakąś etykietą ZC występuje w więcej niż jednym grafie oczekiwania.

Przyjmujemy, że jeśli w konstruowanym grafie występuje cykl, to system jest w stanie zakleszczenia. Jeśli nie ma cyklu w konstruowanym grafie, to w chwili gdy algorytm wykrywania został wywołany za pomocą inicjujących komunikatów wysłanych przez koordynatora (w kroku 1), system nie znajdował się w stanie zakleszczenia.

18.5.2.2 Podejście w pełni rozproszone

W algorytmie wykrywania zakleszczenia w pełni rozproszonym wszystkie procesy nadzorcze (ang. *controllers*) dzielą po równo odpowiedzialność za wykrywanie zakleszczeń. W tym schemacie każde stanowisko konstruuje graf oczekiwania, który reprezentuje część grafu globalnego, w zależności od dynamicznego zachowania systemu. Idea polega na tym, że jeśli istnieje zakleszczenie, to (przynajmniej) w jednym z częściowych grafów pojawi się cykl. Przedstawiamy taki algorytm, zawierający konstruowanie częściowych grafów na każdym stanowisku.

Każde stanowisko utrzymuje własny, lokalny graf oczekiwania. Lokalny graf oczekiwania w tej metodzie różni się od wcześniej opisanego tym, że dodaje się do niego jeden uzupełniający węzeł P_{es} . W grafie istnieje łuk $P_i \rightarrow P_{\text{es}}$, jeśli proces P_i czeka na obiekt danych z innego stanowiska, przetrzymy-



Rys. 18.6 Rozszerzone lokalne grafy oczekiwania z rys. 18.3

wany przez dowolny proces. Podobnie, łuk $P_{ei} \rightarrow P_i$ istnieje w grafie wtedy, kiedy na innym stanowisku istnieje proces czekający na uzyskanie dostępu do zasobu, będącego aktualnie w dyspozycji procesu P_i na danym, lokalnym stanowisku.

W celu zilustrowania tej sytuacji przeanalizujemy dwa lokalne grafy oczekiwania z rys. 18.3. Dodanie węzła P_{ei} w obu grafach daje w wyniku lokalne grafy oczekiwania przedstawione na rys. 18.6.

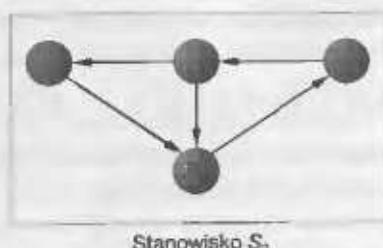
Jeśli lokalny graf oczekiwania ma cykl, który nie zawiera węzła P_{ei} , to system jest w stanie zakleszczenia. Jeśli jednak istnieje cykl zawierający P_{ei} , to wynika z tego, że istnieje możliwość zakleszczenia. Aby przekonać się, czy zakleszczenie istnieje, należy wywołać rozproszony algorytm wykrywania zakleszczenia.

Załóżmy, że na stanowisku S_1 lokalny graf oczekiwania zawiera cykl z węzłem P_{ei} . Cykl ten musi mieć postać

$$P_{ei} \rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \dots \rightarrow P_{k_n} \rightarrow P_{ei},$$

która wskazuje, że transakcja P_{k_n} na stanowisku S_1 czeka na uzyskanie dostępu do obiektu danych znajdującego się na innym stanowisku, powiedzmy – S_j . Po wykryciu tego cyklu stanowisko S_1 wysyła do stanowiska S_j komunikat o rozpoznaniu zakleszczenia, zawierający informację o tym cyklu.

Gdy stanowisko S_j otrzyma komunikat o wykryciu zakleszczenia, wówczas aktualizuje swój lokalny graf oczekiwania za pomocą nowo otrzymanej informacji. Następnie przegląda nowo skonstruowany graf oczekiwania w poszukiwaniu cyklu nie zawierającego P_{ei} . Jeśli taki cykl istnieje, to zakleszczenie zostało znalezione i podejmuje się stosowne działania ratunkowe. Jeśli odnaleziono cykl zawierający P_{ei} , to stanowisko S_j przekazuje komunikat o wykryciu zakleszczenia do stanowiska – powiedzmy – S_k . Stanowisko S_k w reakcji na to powtarza opisaną procedurę. W ten sposób po skończonej liczbie powtórzeń albo zostaje wykryte zakleszczenie, albo procedura wykrywania zakleszczenia się kończy.



Rys. 18.7 Rozszerzony lokalny graf oczekiwania na stanowisku S_2 z rys. 18.6

W celu zobrazowania tej procedury przeanalizujmy lokalny graf oczekiwania z rys. 18.6. Założymy, że stanowisko S_1 odkrywa cykl

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Ponieważ proces P_3 czeka na uzyskanie obiektu danych ze stanowiska S_2 , więc ze stanowiska S_1 do stanowiska S_2 następuje wysłanie komunikatu o wykryciu zakleszczenia, opisującego ten cykl. Gdy S_2 otrzyma ten komunikat, wówczas uaktualni swój lokalny graf oczekiwania, otrzymując taki graf oczekiwania jak na rys. 18.7. Graf ten zawiera cykl

$$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2,$$

w którym nie ma węzła P_{ex} . Wynika z tego, że system jest w stanie zakleszczenia i powinny być podjęte działania zmierzające do jego likwidacji.

Zauważmy, że wynik byłby taki sam, gdyby stanowisko S_2 pierwsze wykryło cykl w swoim lokalnym grafie oczekiwania i wysłało komunikat o wykryciu zakleszczenia do stanowiska S_1 . W najgorszym przypadku oba stanowiska wykryją cykl mniej więcej w tym samym czasie i zostaną wysłane dwa komunikaty o wykryciu zakleszczenia: jeden przez S_1 do S_2 oraz drugi przez S_2 do S_1 . Sytuacja ta spowoduje niepotrzebne przesyłanie komunikatów i nakład na uaktualnianie dwóch lokalnych grafów oczekiwania i poszukiwanie cykli w obu grafach.

Aby zmniejszyć ruch komunikatów, przypiszemy każdej transakcji P_i niepowtarzalny identyfikator, który oznaczmy przez $ID(P_i)$. Gdy stanowisko S_k wykryje, że jego lokalny graf oczekiwania ma cykl w postaci

$$P_{ex} \rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \dots \rightarrow P_{k_n} \rightarrow P_{ex},$$

zawierający węzel P_{ex} , to wyśle komunikat o wykryciu zakleszczenia do innego stanowiska tylko wtedy, gdy

$$ID(P_{k_n}) < ID(P_{k_1}).$$

W przeciwnym razie stanowisko S_k kontynuuje normalne działanie, pozostawiając inicjowanie algorytmu wykrywania zakleszczenia innemu stanowisku.

Aby zilustrować ten schemat, rozważmy znów grafy oczekiwania z rys. 18.6, utrzymywane na stanowiskach S_1 i S_2 . Założymy, że

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4).$$

Przymijmy, że oba stanowiska odkrywają lokalne cykle mniej więcej w tym samym czasie. Cykl na stanowisku S_1 ma postać

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Ponieważ $ID(P_3) > ID(P_2)$, stanowisko S_1 nie wysyła komunikatu o wykryciu zakleszczenia do stanowiska S_2 .

Cykl na stanowisku S_2 ma postać

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}.$$

Ponieważ $ID(P_2) < ID(P_1)$, stanowisko S_2 wysyła komunikat o wykryciu zakleszczenia do stanowiska S_1 , które – po jego otrzymaniu – zaktualizuje swój lokalny graf oczekiwania. Stanowisko S_1 poszuka następnie cyklu w tym grafie i wykryje, że system jest w stanie zakleszczenia.

18.6 ■ Algorytmy elekcji

Pokazaliśmy w punkcie 18.3, że w wielu algorytmach rozproszonych stosuje się proces koordynujący, który wykonuje funkcje potrzebne innym procesom w systemie. Wśród funkcji tych znajdują się takie, jak: wymuszanie wzajemnego wykluczania, utrzymywanie globalnego grafu oczekiwania w celu wykrywania zakleszczenia, zastępowanie utraconych żetonów czy kontrolowanie urządzeń wejścia lub wyjścia w systemie. Jeśli proces koordynujący przestaje działać z powodu awarii stanowiska, na którym przebywa, to w celu dalszej pracy system musi uruchomić nową kopię koordynatora na jakimś innym stanowisku. Algorytmy rozstrzygające, gdzie powinno nastąpić wznowienie kopii koordynatora, są nazywane *algorytmami elekcji* (ang. *election*).

W algorytmach elekcji zakłada się, że z każdym aktywnym procesem w systemie jest związany jednoznaczny numer priorytetu. W celu uproszczenia notacji założymy, że i jest numerem priorytetu procesu P_i . Dla prostoty przyjmiemy też, że istnieje wzajemnie jednoznaczna odpowiedniość między procesami i stanowiskami, zatem do stanowisk będziemy się odwoływać tak jak do procesów. Koordynatorem jest zawsze proces z najwyższym numerem priorytetu. Jeśli więc koordynator ulegnie awarii, to algorytm musi wybrać ten spośród aktywnych procesów, który ma największy numer priorytetu. Numer ten musi zostać wysłany do każdego aktywnego procesu w systemie. Oprócz

tego algorytmu musi zawierać mechanizm umożliwiający zidentyfikowanie bieżącego koordynatora przez proces przywracany do działania.

Przedstawimy poradę dwa interesujące przykłady algorytmów elekcji dla dwóch różnych konfiguracji systemów rozproszonych. Pierwszy algorytm ma zastosowanie w tych systemach, w których każdy proces może wysłać komunikat do dowolnego procesu w systemie. Drugi algorytm znajduje zastosowanie w systemach zorganizowanych w postaci pierścienia (logicznego lub fizycznego). W obu algorytmach na wykonanie elekcji potrzeba n komunikatów, przy czym n jest liczbą procesów w systemie. Zakładamy, że proces, który uległ uszkodzeniu, ma o tym informację podczas przywracania go do działania; podejmuje on zatem odpowiednie czynności (opisane dalej), aby dołączyć do zbioru procesów aktywnych.

18.6.1 Algorytm tyrana

Przypuśćmy, że proces P_i wysyła zamówienie, które pozostaje bez odpowiedzi ze strony koordynatora przez czas T . W tej sytuacji zakłada się, że koordynator uległ awarii, a proces P_i próbuje sam siebie obrac nowym koordynatorem, posługując się następującym algorytmem.

Proces P_i wysyła komunikat o elekcji do wszystkich procesów z wyższymi numerami priorytetów. Następnie czeka przez czas T na nadanie odpowiedzi od któregokolwiek z tych procesów.

Jeśli w czasie T nie nadaje się żadna odpowiedź, to proces P_i zakłada, że wszystkie procesy z numerami priorytetów większymi od i uległy awarii i wybiera siebie na nowego koordynatora. Proces P_i wznowia kopię koordynatora i wysyla komunikat informujący wszystkie procesy o numerach priorytetów mniejszych od i , że P_i został nowym koordynatorem.

Jeżeli jednak odpowiedź nadaje się, to proces P_i rozpoczęta czekanie przez czas T' , aby otrzymać komunikat, że koordynatorem został obrany jakiś proces o wyższym numerze priorytetu. (Jakiś inny proces obrą się koordynatorem i powinien powiadomić o tym w czasie T'). Jeśli przez czas T' nie nadaje się żaden komunikat, to przyjmuje się, że proces o wyższym numerze uległ awarii i proces P_i powinien wznowić wykonywanie algorytmu.

Jeśli proces P_i nie jest koordynatorem, to w każdej chwili swojego działania może otrzymać jeden z następujących dwóch komunikatów od procesu P_j :

1. P_j jest nowym koordynatorem ($j > i$). Proces P_i odnotowuje tę informację.
2. P_j rozpoczął elekcję ($j < i$). Proces P_i wysyła odpowiedź do P_j i sam rozpoczęta wykonywać algorytm elekcji, jeśli jeszcze go nie rozpoczął.

Proces, który zakończy swój algorytm, ma najwyższy numer, więc zostaje koordynatorem. Wysłał on swój numer do wszystkich aktywnych procesów z mniejszymi numerami. Proces, który po awarii powraca do działania, natychmiast rozpoczyna wykonywanie tego samego algorytmu. Jeśli wśród aktywnych procesów nie ma procesów z wyższymi numerami, to wznowiający działanie proces zmusza wszystkie procesy z niższymi numerami do uznania go koordynatorem nawet wówczas, gdy istnieje wśród nich koordynator z niższym numerem. Z tego powodu algorytm ten nazwano algorytmem *tyrana*.

Zademonstrujmy działanie algorytmu na prostym przykładzie systemu złożonego z czterech procesów: P_1, \dots, P_4 . Wykoranie algorytmu przebiega następująco:

1. Wszystkie procesy są aktywne; P_4 jest koordynatorem.
2. Procesy P_1 i P_4 ulegają awarii. Proces P_2 stwierdza, że P_4 jest uszkodzony, ponieważ na zamówienie, które do niego wysłał, odpowiedź nie nadeszła w czasie T . Proces P_2 rozpoczyna algorytm elekcji, wysyłając komunikat do P_3 .
3. Proces P_3 otrzymuje wiadomość, odpowiada procesowi P_2 i sam rozpoczyna algorytm elekcji, wysyłając informację o tym do P_4 .
4. Proces P_2 otrzymuje odpowiedź procesu P_3 i rozpoczyna czekanie przez czas T' .
5. Proces P_4 nie odpowiada przez czas T , więc P_3 obiera się nowym koordynatorem i wysyła komunikat z numerem 3 do P_1 i P_2 (którego P_1 nie odbiera, bo jest uszkodzony).
6. Po pewnym czasie proces P_1 , podejmując na nowo działanie, wysyła komunikat o elekcji do procesów P_2, P_3 i P_4 .
7. Procesy P_2 i P_3 odpowiadają procesowi P_1 i rozpoczynają wykonywać własne algorytmy elekcji. Powtórnie zostaje wybrany proces P_3 na podstawie tego samego ciągu zdarzeń co uprzednio.
8. Kiedy wreszcie proces P_1 powraca do pracy, wtedy powiadamia procesy P_1, P_2 i P_3 , że teraz znów on jest koordynatorem. (P_4 nie zwolnia elekcji, gdyż jest on procesem o najwyższym numerze w systemie).

18.6.2 Algorytm pierścieniowy

W *algorytmie pierścieniowym* (ang. *ring algorithm*) zakłada się, że łącza są jednokierunkowe oraz że procesy wysyłają komunikaty do swoich sąsiadów po prawej stronie. Główną strukturą danych używaną przez algorytm jest *lista*

aktywna (ang. *active list*), zawierająca numery priorytetów wszystkich procesów aktywnych w systemie w chwili zakończenia algorytmu. Każdy proces utrzymuje własną listę aktywną. Przebieg algorytmu jest następujący:

1. Jeśli proces P_i wykrywa awarię koordynatora, to tworzy nową listę aktywną, która początkowo jest pusta. Następnie wysyła komunikat *elekcja(i)* do swojego sąsiada po prawej stronie i dodaje numer i do swojej listy aktywnej.
2. Jeśli P_j otrzyma komunikat *elekcja(j)* od procesu po lewej, to musi odpowiedzieć w jeden z trzech sposobów:
 - (a) Jeśli jest to pierwszy komunikat *elekcja*, z którym ma do czynienia lub który wysłał, to P_j tworzy nową listę aktywną z numerami i i j . Wysyła wówczas komunikat *elekcja(i)*, a po nim komunikat *elekcja(j)*.
 - (b) Jeśli $i \neq j$ (tj. otrzymany komunikat nie zawiera numeru procesu P_j), to P_j dodaje j do swojej listy aktywnej i przekazuje ten komunikat dalej, do swojego sąsiada po prawej stronie.
 - (c) Jeśli $i = j$ (tzn. P_j otrzymuje komunikat *elekcja(i)*), to lista aktywna procesu P_j zawiera już numery wszystkich aktywnych procesów w systemie. Proces P_j może teraz określić największy numer na liście aktywnej, aby wskazać nowy proces koordynujący.

W algorytmie tym nic sprecyzowano, w jaki sposób proces przywracany do działania ma określić numer bieżącego koordynatora. Jednym z rozwiązań mogłyby być wymaganie, aby proces reaktywowany wysłał komunikat z zapytaniem. Komunikat taki, przekazywany naokoło pierścienia, dotarłby do bieżącego koordynatora, który w odpowiedzi wysłałby wiadomość o swoim numerze.

18.7 ■ Osiąganie porozumienia

Aby system mógł być niezawodny, jest potrzebny mechanizm, który pozwala zbiorowi procesów uzgadniać pewną wspólną „wartość”. Istnieje kilka przyczyn, które mogą stanąć na przeszkodzie w osiągnięciu takiego porozumienia. Po pierwsze, środki komunikacji mogą ulegać uszkodzeniom, powodując utratę lub zniekształcanie komunikatów. Po drugie, same procesy mogą być wadliwe i zachowywać się w sposób nieokreślony. Najlepsze, czego można oczekiwac w takim przypadku, to czysty przebieg awarii procesów, prawa-

dzący do zatrzymania procesów bez zbaczania z normalnego toru ich wykonywania. W najgorszym przypadku procesy mogą wysyłać zniekształcone komunikaty do innych procesów lub nawet współpracować z innymi, wadliwymi procesami, dając do dezintegracji systemu.

Zagadnienie to opisano obrazowo jako *problem bizantyjskich generalów* (ang. *Byzantine generals problem*). Pewna liczba dywizji armii bizantyjskiej, z których każdą dowodzi generał, otacza obóz nieprzyjaciela. Bizantyjscy generalowie muszą osiągnąć porozumienie co do tego, czy zaatakować nieprzyjaciela o święcie. Porozumienie wszystkich generalów ma znaczenie zasadnicze, ponieważ atak wykonany tylko przez część dywizji mogliby zakończyć się klęską. Poszczególne dywizje są rozmieszczone w terenie i wodzowie mogą komunikować się ze sobą tylko przez posłanców, którzy biegają od obozu do obozu. Są co najmniej dwie poważne przyczyny tego, że generalowie mogą nie dojść do porozumienia:

- Posłanci mogą zostać pochwyceni przez nieprzyjaciela i nie dostarczą komunikatów. Ta sytuacja odpowiada zawodnym łączom komunikacyjnym w systemie komputerowym i będzie omówiona w p. 18.7.1.
- Wśród generalów mogą być *zdrajcy* usiłujący przeszkodzić *lojalnym* generałom w osiągnięciu porozumienia. Ta sytuacja odpowiada wadliwym procesom w systemie komputerowym i będzie omówiona w p. 18.7.2.

18.7.1 Zawodna komunikacja

Przyjmijmy, że jeśli proces ulega awarii, to robi to w sposób czysty oraz że środki komunikacji są zawodne*. Przypuśćmy, że proces P_i wysłał ze stanowiska A komunikat do procesu P_j na stanowisku B i chce się dowiedzieć, czy P_j odebrał ten komunikat, aby móc podjąć decyzję co dalszych swoich obliczeń. Na przykład P_i ma wykonać funkcję S , jeśli proces P_j otrzymał jego komunikat lub funkcję F , jeśli P_j komunikatu nie odebrał (z powodu jakiejś awarii sprzętu).

Do wykrywania awarii można posłużyć się schematem *odliczania czasu* (ang. *timeout*), podobnym do tego, który opisano w p. 16.4.1. Gdy proces P_i wysyła komunikat, wówczas określa również czas, przez który jest gotów czekać na komunikat potwierdzający od procesu P_j . Kiedy proces P_i odbierze komunikat, wtedy natychmiast wysyła potwierdzenie do P_j . Jeśli proces P_j odbierze komunikat potwierdzający w określonym czasie, to może bezpiecznie uznać, że proces P_i odebrał komunikat. Jeśli jednak wystąpi przekroczenie czasu, to P_i będzie zmuszony powtórnie wysłać komunikat i czekać na po-

* Ten wariant problemu określa się również jako problem dwu armii (ang. *two-army problem*). – Przyp. tłum.

twierdzenie. Procedurę tę powtarza się dopóty, dopóki proces P_i nie otrzyma potwierdzenia lub nie zostanie powiadomiony przez system, że stanowisko B nie działa. W pierwszym przypadku proces wykona funkcję S , w drugim – funkcję F . Zauważmy, że jeżeli do wyboru były tylko te dwie możliwości, to proces P_i musi czekać, aż zostanie powiadomiony o wystąpieniu jednej z dwóch omówionych sytuacji.

Załóżmy teraz, że również proces P_i chce wiedzieć, czy P_i otrzymał jego komunikat potwierdzający, aby móc wybrać sposób kontynuowania obliczeń. Na przykład proces P_i może wykonać obliczenie S tylko pod warunkiem, że jest pewny, że P_i otrzymał jego potwierdzenie. Innymi słowy, procesy P_i i P_j wykonają funkcję S wtedy i tylko wtedy, gdy oba porozumięły się co do tego. Okazuje się, że przy możliwości awarii wypełnienie tego zadania jest niewykonalne. Mówiąc precyzyjniej – w środowisku rozproszonym nie jest możliwe, aby procesy P_i i P_j uzgodniły do końca swoje stany.

Udowodnimy tę tezę. Przypuśćmy, że istnieje minimalny ciąg przesyłań komunikatów, po dostarczeniu których oba procesy uzgadniają wykonanie funkcji S . Niech k' będzie ostatnim komunikatem wysłanym przez P_i do P_j . Ponieważ proces P_i nie wie, czy jego komunikat dotarł do procesu P_j (komunikat mógł zaginąć wskutek awarii), więc P_j wykona funkcję S niezależnie od tego, jak zakończyło się dostarczanie komunikatu. Skoro tak, to komunikat k' można usunąć z ciągu bez wpływu na podejmowaną decyzję. Zatem przyjęty na wstępie ciąg nie był minimalny, co przeczy naszemu założeniu i pokazuje, że ciągu takiego nie ma. Procesy nigdy nie mogą być pewne, że oba wykonają funkcję S .

18.7.2 Wadliwe procesy

Załóżmy, że środki komunikacji są niezawodne, lecz procesy mogą ulegać uszkodzeniom powodującym ich nieokreślone zachowanie. Rozważmy system n procesów, w którym nie więcej niż m jest wadliwych. Przyjmijmy, że każdy proces P_i ma pewną prywatną wartość V_i . Chcemy wymyślić algorytm, który pozwoli każdemu poprawnemu procesowi P_i zbudować wektor $X_i = (A_{i,1}, A_{i,2}, \dots, A_{i,n})$ taki, że:

1. Jeśli P_i jest procesem poprawnym, to $A_{i,j} = V_j$.
2. Jeżeli P_i oraz P_j są oba procesami poprawnymi, to $X_i = X_j$.

Istnieje wiele rozwiązań tego problemu*. Wszystkie one mają następujące cechy:

* Znanego pod nazwą „porozumienia bizantyjskiego” (ang. *Byzantine agreement*) – Przyp. tłum.

1. Poprawny algorytm można podać tylko wtedy, gdy $n \geq 3 \times m + 1$.
2. Najgorsze opóźnienie w dochodzeniu do porozumienia jest proporcjonalne do $m + 1$ opóźnień wynikających z przekazywania komunikatów.
3. Liczba komunikatów wymaganych do osiągnięcia porozumienia jest wielka. Zaden proces z osobra nie jest godny zaufania, więc wszystkie procesy muszą zbierać wszystkie informacje i podejmować własne decyzje.

Zamiast podawać ogólne rozwiązanie, które byłoby dość skomplikowane, przedstawimy algorytm dla prostego przypadku, w którym $m = 1$ i $n = 4$. Algorytm ten wymaga dwóch obiegów wymiany informacji:

1. Każdy proces wysyła prywatną wartość do pozostałych trzech procesów.
2. Każdy proces wysyła do wszystkich innych procesów informacje, które uzyskał w pierwszym obiegu.

Wadliwy proces oczywiście może odmówić wysłania komunikatu. W tym przypadku proces poprawny może wybrać jakąś dowolną wartość i udac, że została ona wysłana przez tamten proces.

Po zakończeniu obu obiegów, poprawny proces P_i może skonstruować swój wektor $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ w następujący sposób:

1. $A_{i,i} = V_i$.
2. Dla $j \neq i$, jeśli co najmniej dwie z trzech wartości przekazanych procesowi P_j (w obu rundach wymiany) zgadzają się, to wartość występująca częściej będzie użyta do określenia wartości $A_{i,j}$. W przeciwnym razie do określenia wartości $A_{i,j}$ będzie użyta wartość zastępcza, powiedzmy – nil.

18.8 ■ Podsumowanie

W systemie rozproszonym, nie mającym pamięci dzielonej ani wspólnego zegara, określenie dokładnej kolejności wystąpienia dwóch zdarzeń jest czasami niemożliwe. Relacja uprzedniości zdarzeń porządkuje tylko częściowo zdarzenia w systemie rozproszonym. Do spójnego uporządkowania zdarzeń w systemie rozproszonym można zastosować znaczniki czasu.

Wzajemne wykluczanie w systemie rozproszonym można implementować na kilka sposobów. W podejściu centralizowanym jeden z procesów systemu jest wybierany do koordynowania wejścia do sekcji krytycznej. W podejściu w pełni rozproszonym podejmowanie decyzji jest rozłożone w całym systemie. Algorytm rozproszony, przydatny w sieciach pierścieniowych, polega na przekazywaniu żetonu.

Zapewnianie niepodzielności pociąga za sobą konieczność uzgadniania końcowego wyniku transakcji T przez wszystkie stanowiska biorące udział w jej wykonaniu. Transakcja T zostaje zatwierdzona na wszystkich stanowiskach albo na wszystkich stanowiskach ulega zaniechaniu. Aby zagwarantować tę właściwość, koordynator transakcji T musi wykonać protokół *zatwierdzania*. Najpopularniejszym protokołem zatwierdzania jest protokół 2PC (zatwierdzanie dwufazowe).

Rozmaite schematy sterowania współbieżnością, stosowane w systemach scentralizowanych, mogą być dostosowane do użytku w środowisku rozproszonym. W przypadku protokołu blokowania jedną konieczną do wprowadzenia zmianą jest sposób realizacji zarządcy blokowania. Schematy ze znacznikami czasu oraz schematy uprawomocniania wymagają tylko zmian w mechanizmie generowania *globalnie* niepowtarzalnych znaczników czasu. Mechanizm taki może powodować konkatenowanie lokalnego znacznika czasu z identyfikatorem stanowiska albo przesuwać zegary lokalne, gdy tylko pojawią się komunikaty z większymi znacznikami czasu.

Podstawowa metoda postępowania z zakleszczeniami w środowisku rozproszonym polega na wykrywaniu zakleszczenia. Główną trudnością jest rozstrzyganie co do sposobu utrzymywania grafu oczekiwania. Istnieją różne metody organizacji grafu oczekiwania, w tym scentralizowane i w pełni rozproszone.

W niektórych algorytmach rozproszonych jest konieczne zastosowanie koordynatora. Jeśli koordynator przestaje działać wskutek awarii stanowiska, na którym przebywa, to system może kontynuować pracę tylko po uruchomieniu nowej kopii koordynatora na jakimś innym stanowisku. Dokonuje się tego przez utrzymywanie zapasowego koordynatora, gotowego do przejęcia obowiązków w przypadku awarii. Inna metoda polega na wyborze nowego koordynatora po awarii dotychczasowego koordynatora. Algorytm przesyłający o miejscu podjęcia działania przez nowego koordynatora nazywa się *elekcją*. Do wyboru nowego koordynatora w przypadku awarii można zastosować dwa algorytmy elekcji – algorytm tyrania oraz algorytm pierścieniowy.

Ćwiczenia

- 18.1** Omów zalety i wady dwóch przedstawionych przez nas metod tworzenia globalnych, niepowtarzalnych znaczników czasu.
- 18.2** Twoja firma buduje sieć komputerową i poproszono Cię o napisanie algorytmu zapewniającego wzajemne wykluczanie w warunkach rozproszenia. Którego schematu warto by było użyć? Uzasadnij swój wybór.

- 18.3** Dlaczego wykrywanie zakleszczenia jest bardziej kosztowne w środowisku rozproszonym niż w środowisku scentralizowanym?
- 18.4** Twoja firma buduje sieć komputerową i poproszono Cię o opracowanie schematu postępowania z problemem zakleszczenia.
- Czy użyjesz schematu wykrywania zakleszczenia, czy też schematu zapobiegania zakleszczeniu?
 - Jeśli decydujesz się zastosować schemat zapobiegania zakleszczeniu, to który wybierzesz? Uzasadnij swój wybór.
 - Jeśli decydujesz się użyć schematu wykrywania zakleszczenia, to który zastosujesz? Uzasadnij swój wybór.
- 18.5** Rozważ następujący, *hierarchiczny algorytm wykrywania zakleszczenia*, w którym globalny graf oczekiwania jest rozłożony między pewną liczbę *kontrolerów* zorganizowanych w drzewo. Każdy kontroler nie będący liściem drzewa utrzymuje graf oczekiwania zawierający niezbędne informacje z grafów kontrolerów w poddrzewie położonym niżej. Niech w szczególności S_A , S_B i S_C będą kontrolerami takimi, że S_C jest najniższym wspólnym przodkiem S_A i S_B (S_C musi być jednoznaczny, ponieważ działały na drzewie). Założmy, że węzeł T , występuje w lokalnych grafach oczekiwania kontrolerów S_A i S_B . Wówczas węzeł T , musi również pojawić się w lokalnym grafie oczekiwania:
- kontrolera S_C ;
 - każdego kontrolera na drodze od S_C do S_A ;
 - każdego kontrolera na drodze od S_C do S_B .
- Ponadto, jeśli węzły T_i i T_j występują w grafie oczekiwania kontrolera S_D i w grafie oczekiwania jednego z potomków S_D istnieje droga z T_i do T_j , to w grafie oczekiwania S_D musi być krawędź $T_i \rightarrow T_j$.
- Wykaż, że jeśli w dowolnym z grafów oczekiwania istnieje cykl, to system jest zakleszczony.
- 18.6** Opracuj algorytm elekcji dla pierścieni dwukierunkowych wydajniejszy niż opisany w tym rozdziale. Ile komunikatów będzie potrzebnych przy n procesach?
- 18.7** Rozważ awarię występującą podeczas dwufazowego zatwierdzania transakcji. W odniesieniu do każdego możliwego rodzaju awarii wyjaśnij, w jaki sposób zatwierdzanie dwufazowe zapewnia niepodzielność transakcji pomimo awarii.

Uwagi bibliograficzne

Rozproszony algorytm rozszerzający relację uprzedniości na spójne, całkowite uporządkowanie wszystkich zdarzeń w systemie (zob. p. 18.1) został opracowany przez Lamporta [228].

Od Lamporta pochodzi też [228] pierwszy ogólny algorytm implementowania wzajemnego wykluczania w środowisku rozproszonym. Schemat opracowany przez Lamporta wymaga $3 \times (n - 1)$ komunikatów na sekcję krytyczną. W ślad za tym Ricart i Agrawala [350] zaproponowali algorytm wymagający tylko $2 \times (n - 1)$ komunikatów. Ich algorytm przedstawiliśmy w p. 18.2.2. Maciąka w artykule [267] przedstawił algorytm pierwiastka kwadratowego służący do osiągania rozproszonego wzajemnego wykluczania. Algorytm przekazywania żetonu w systemach o strukturze pierścieniowej, omówiony w p. 18.2.3, opracował Le Lann [247]. Carvalho i Roucairoł zawarli w artykule [62] rozważania na temat wzajemnego wykluczania w sieciach komputerowych. Wydajne i tolerujące awarie rozwiązanie problemu rozproszonego wzajemnego wykluczania zaprezentowali Agrawal i El Abbadi [4]. Prostą taksonomię algorytmów rozproszonego wzajemnego wykluczania przedstawił Raynal [345].

Zagadnienie synchronizacji rozproszonej omówili: Reed i Kanodia [348] (środowisko pamięci dzielonej), Lamport [228] i [229] oraz Schneider [379] (procesy całkowicie niezależne). Rozproszone rozwiązanie problemu obiadujących filozofów zaprezentował Chang [67].

Protokół 2PC opracowali Lampson i Sturgis [238] oraz Gray [154]. Mohan i Lindsay w materiałach [295] omówili dwie wersje protokołu 2PC – z przewidywaniem zatwierdzenia i z przewidywaniem zaniechania – zmniejszające jego koszt przez przyjmowanie domyślnych założeń co do losu transakcji.

Gray [156], Traiger i in. [428] oraz Spector i Schwarz [399] przedstawili w swoich artykułach problemy implementowania koncepcji transakcji w rozproszonej bazie danych. Ogólne omówienie dotyczące sterowania współbieżnością w środowisku rozproszonym zawiera książka Bernsteina i in. [33].

Rosenkrantz i in. [358] przedstawili algorytm zapobiegania zakleszczeniom w środowisku rozproszonym, korzystający ze znaczników czasu. W pełni rozproszony schemat wykrywania zakleszczenia, omówiony w p. 18.5.2, opracował Obermarck [314]. Metodę hierarchicznego wykrywania zakleszczeń, przedstawioną w ćwicz. 18.3, zaproponowali Menasce i Muntz [283]. Przegląd metod wykrywania zakleszczeń w systemach rozproszonych zaoferowali Knapp [216] i Singhal [393].

Problem bizantyjskich generalów omówili Lamport i in. [233] oraz Pease i in. [327]. Algorytm tyrana został zaprezentowany przez Garcia-Molinę [145]. Algorytm elekcji w systemie o strukturze pierścieniowej napisał Le Lann [247].



Część 6

OCHRONA I BEZPIECZEŃSTWO

Mechanizmy ochrony umożliwiają kontrolowanie dostępu udzielanego użytkownikom w odniesieniu do plików przez ograniczanie jego rodzajów. Poza nadzorowaniem plików ochrona ma również zapewnić, że z segmentów pamięci, procesora i innych zasobów będą mogły korzystać tylko te procesy, któretrzymały odpowiednie pełnomocnictwa od systemu operacyjnego.

Ochrona jest realizowana za pomocą mechanizmu, który nadzoruje dostęp programów, procesów lub użytkowników do zasobów zdefiniowanych w systemie komputerowym. Mechanizm ten powinien umożliwiać określanie zasad nadzoru, jak również pozwalać na ich egzekwowanie.

System bezpieczeństwa zapobiega nieupoważnionemu dostępowi do systemu i uniemożliwia złośliwe uszkadzanie lub zmienianie danych. Organizacja bezpieczeństwa zapewnia środki sprawdzania tożsamości użytkowników systemu w celu ochrony integralności przechowywanych w nim informacji (zarówno danych, jak i kodu) oraz zasobów fizycznych systemu komputerowego.

11072625217540 (65038127)

Rozdział 19

OCHRONA

Różne procesy w systemie operacyjnym muszą być chronione przed wzajemnym oddziaływaniem. Istnieją w tym celu różnorodne mechanizmy, którymi można się posłużyć do zagwarantowania, że pliki, segmenty pamięci, procesor i inne zasoby będą używane tylko przez te procesy, które otrzymały odpowiednie pełnomocnictwa od systemu operacyjnego.

Termin *ochrona* (ang. *protection*) odnosi się do mechanizmu służącego do kontrolowania dostępu programów, procesów lub użytkowników do zasobów zdefiniowanych przez system komputerowy. Mechanizm taki musi zawierać środki pozwalające specyfikować rodzaje wymaganej kontroli oraz pewien zasób środków ich wymuszania. Rozróżniamy dwa pojęcia: ochronę i *bezpieczeństwo* (ang. *security*), które jest miarą zaufania, że system i jego dane pozostaną nienaruszone. Zapewnianie bezpieczeństwa jest znacznie szerszym zagadnieniem niż ochrona; zajmujemy się nim w rozdz. 20.

W niniejszym rozdziale przeanalizujemy zagadnienie ochrony bardziej szczegółowo i opracujemy ujednolicony model implementowania ochrony.

19.1 ■ Cele ochrony

W miarę jak powstawały coraz doskonalsze systemy komputerowe i rozszerzały się ich zastosowania, wzrastało zapotrzebowanie na ochronę ich nienaruszalności. Potrzeba ochrony pojawiła się najpierw ubocznie w wieloprogramowych systemach operacyjnych; chodziło o to, aby zapewnić bezpieczne dzielenie wspólnej przestrzeni nazw logicznych, w rodzaju katalogu plików, lub wspólnej przestrzeni obiektów fizycznych, takich jak pamięć, przez użytko-

kowników, do których nie można się było odnosić z zaufaniem. W nowoczesnych koncepcjach ochrony wzięto pod uwagę podwyższanie niezawodności dowolnego złożonego systemu, w którym korzysta się z zasobów dzielonych.

Istnieje kilka powodów do wprowadzania ochrony. Najbardziej oczywista jest potrzeba zapobiegania złośliwym, zamierzonym naruszeniom ograniczeń dostępu przez użytkowników. Ważniejszym i ogólniejszym powodem jest potrzeba zapewnienia, aby każdy wykonywany w systemie proces używał zasobów systemowych tylko w sposób zgodny z polityką przewidzianą dla użytkowania danych zasobów. Jest to nieodzowne ze względu na niezawodność systemu.

Ochrona może przyczyniać się do zwiększenia niezawodności dzięki wykrywaniu błędów, które kryją interfejsy między składowymi podsystemami. Wczesne wykrycie błędu w interfejsie może często zapobiec szkodliwemu oddziaływaniu na sprawny podsystem ze strony podsystemu pracującego właściwie. Zasobu nie objętego ochroną nie można ustrzec przed użyciem (lub nadużyciem) przez nieuprawnionego lub niekompetentnego użytkownika. W systemach chronionych istnieją środki pozwalające odróżnić użycie uprawnione od nieuprawnionego.

Rola ochrony w systemie komputerowym jest dostarczenie *mechanizmu* (ang. *mechanism*) do wymuszania *polityki* (ang. *policy*) rządzącej sposobem użycia zasobu. Politykę można określać różnorodnie. Część zasad ochrony ustala się w projekcie systemu, inne są wprowadzane przez jego kierownictwo. Jeszcze inną politykę określają indywidualni użytkownicy systemu w celu ochrony ich własnych danych i programów. System ochrony musi być tak elastyczny, aby można było za jego pomocą wymuszać przestrzeganie każdej zasady, którą da się w nim zadeklarować.

Zasady korzystania z zasobu mogą się zmieniać w zależności od zastosowania i upływu czasu. Z tych powodów ochrony nie można rozpatrywać jako sprawy interesującej wyłącznie projektanta systemu operacyjnego. Powinna ona również służyć programistom aplikacji do strzeżenia przed nadużyciem zasobów tworzonych i dostarczanych przez podsystem użytkowy. W tym rozdziale opisujemy mechanizmy, które powinny się znajdować w systemie operacyjnym, aby projektant aplikacji mógł z nich skorzystać przy projektowaniu własnego oprogramowania ochronnego.

Podstawową zasadą jest oddzielenie *polityki* od *mechanizmu*. Mechanizmy określają, jak coś zrobić. W przeciwnieństwie do nich polityka decyduje, co ma być zrobione. Oddzielenie polityki od mechanizmu jest ważne ze względu na elastyczność systemu. Politykę zwykło się zmieniać w zależności od miejsca i czasu. W najgorszym przypadku każda zmiana w polityce może wymagać zmiany w służącym do jej wdrożenia mechanizmie. Bardziej pożąданie są mechanizmy ogólne, gdyż zmiana polityki wymaga w ich wypadku jedynie modyfikacji pewnych systemowych parametrów lub tablic.

19.2 ■ Domeny ochrony

System komputerowy jest zbiorem procesów i obiektów. Przez *obiekty* (ang. *objects*) rozumiemy zarówno elementy sprzętu (takie jak procesor, segmenty pamięci, drukarki, dyski i przewijaki taśmy), jak i elementy oprogramowania (pliki, programy i semafory). Każdy obiekt ma jednoznaczną nazwę, która wyróżnia go spośród wszystkich innych obiektów w systemie, a dostęp do każdego obiektu odbywa się tylko za pomocą dobrze określonych, sensownych operacji. Obiekty są w istocie *abstrakcyjnymi typami danych* (ang. *abstract data types*).

Rodzaj wykonywanych operacji może zależeć od obiektu. Na przykład procesor może tylko wykonywać rozkazy. Segmente pamięci mogą być czytane i zapisywane, a czytnik kart może być użyty tylko do czytania. Przewijaki taśmy mogą służyć do czytania, pisania i przewijania. Pliki danych mogą być tworzone, otwierane, czytane, zapisywane, zamazywane i usuwane; pliki zawierające programy mogą być czytane, zapisywane, wykonywane i usuwane.

Rzecz jasna, proces powinien mieć dostęp tylko do tych zasobów, do których został uprawniony. Co więcej, w każdej chwili powinien mieć możliwość kontaktu tylko z tymi zasobami, których aktualnie potrzebuje do zakończenia zadania. Ten wymóg, nazywany potocznie zasadą *wiedzy koniecznej* (ang. *need-to-know*), przydaje się do ograniczania zakresu uszkodzeń, które błędny proces może spowodować w systemie. Jeśli na przykład proces p wywoła procedurę A , to procedura ta powinna mieć dostęp tylko do swoich zmiennych i przekazanych do niej parametrów formalnych; nie powinna mieć dostępu do wszystkich zmiennych procesu p . Podobnie w przypadku, gdy proces p wywołuje kompilator w celu skompilowania konkretnego pliku: kompilator nie powinien mieć dostępu do dowolnego pliku, lecz tylko do dobrze określonego podzbioru plików (plik źródłowy, protokół z tłumaczenia itd.) związanego z plikiem, który należy skompilować. I na odwrót, kompilator może mieć prywatne pliki, używane do rozliczania i optymalizacji, do których nie powinien mieć dostępu proces p .

19.2.1 Struktura domenowa

W celu wdrożenia tego schematu wprowadzamy pojęcie *domeny ochrony* (ang. *protection domain*). Proces działa w domenie ochrony, która określa dostępne dla niego zasoby. Każda domena definiuje zbiór obiektów i rodzaje operacji, które można wykonywać dla każdego obiektu. Możliwość wykonywania operacji na obiekcie jest *prawem dostępu* (ang. *access right*). Domena jest zbiorem praw dostępu do obiektów, z których każde ma postać pary uporządkowanej: \langle nazwa-obiektu, zbiór-praw \rangle . Jeśli na przykład w domenie D

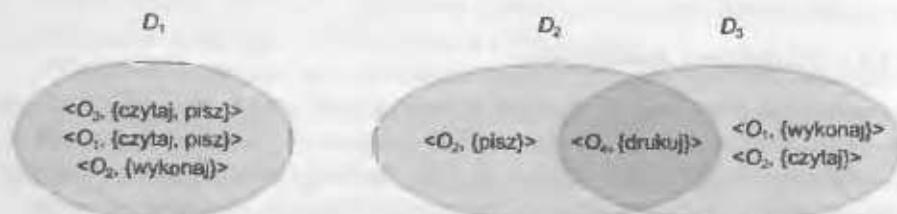
istnieje prawo dostępu $\langle \text{plik } F, \{\text{czytaj, pisz}\} \rangle$, to proces działający w domenie D może zarówno czytać plik F , jak i do niego pisać; inne operacje na tym obiekcie są dla procesu zakazane.

Domeny nie muszą być rozłączne – mogą dzielić prawa dostępu. Na przykład na rys. 19.1 mamy trzy domeny: D_1 , D_2 i D_3 . Prawo dostępu $\langle O_4, \{\text{drukuj}\} \rangle$ jest dzielone przez D_2 i D_3 , co oznacza, że proces działający w każdej z tych dwu domen może drukować obiekt O_4 . Zauważmy, że aby czytać i pisać obiekt O_1 , proces musi działać w domenie D_1 . Z kolei tylko w domenie D_3 procesy mogą wykonywać obiekt O_1 .

Związek między procesem a domeną może być statyczny (jeśli zbiór zasobów dostępnych dla procesu jest ustalony podczas jego dalszego działania) lub dynamiczny. Jak można oczekiwąć, zagadnienia związane z ustanawianiem dynamicznych domen ochrony wymagają staranniejszego potraktowania niż prostszy przypadek statyczny.

Jeśli związek między procesem a domeną jest ustalony i chcielibyśmy, aby zasada wiedzy koniecznej była przestrzegana, to musi istnieć mechanizm zmiany zawartości domeny. Wykonanie jakiegoś procesu może mieć dwie różne fazy. W pierwszej fazie procesowi może być na przykład potrzebny dostęp do czytania, a w drugiej – do pisania. Jeśli domena jest statyczna, to należy zdefiniować domenę zawierającą prawa dostępu zarówno do czytania, jak i do pisania. Jednakże takie rozwiązanie zawiera więcej praw, niż jest wymaganych w każdej z dwóch faz, ponieważ proces otrzymuje dostęp do czytania w fazie, w której wystarczyłby mu dostęp do pisania – i na odwrót. Zatem zasada wiedzy koniecznej jest naruszona. Należy pozwolić na dokonywanie zmian w zawartości domeny, tak aby mogła ona zawsze odzwierciedlać minimum niezbędnych praw dostępu.

Jeśli związek między procesem a domeną jest dynamiczny, to uzyskując się mechanizm pozwalający procesowi na przełączanie od jednej domeny do drugiej. Może być również pożądana zmiana zawartości domeny. Jeśli nie można zmienić zawartości domeny, to ten sam efekt można uzyskać przez



Rys. 19.1 System z trzema domenami ochrony (zaczerpnięty z: J. Quarterman, S. Wilhelm: *UNIX, POSIX and Open Systems: The Open Standards Puzzle* (fig. 1.1, p.5) © 1993 Addison Wesley Longman Inc. Reprinted with permission of Addison Wesley Longman)

utworzenie nowej domeny ze zmienioną zawartością i przełączenie się do tej domeny w razie konieczności zmiany zawartości domeny.

Zauważmy, że domenę można zrealizować różnymi sposobami:

- Każdy *użytkownik* może być traktowany jak domena. W tym przypadku zbiór obiektów, do których można mieć dostęp, zależy od identyfikacji użytkownika. Przełączanie domen wykonuje się wraz ze zmianą użytkownika – z zasady wówczas, gdy jeden użytkownik się wyrejestruje, a drugi dokonuje rejestracji.
- Każdy *proces* może być domeną. W tym przypadku zbiór obiektów, do których można mieć dostęp, zależy od identyfikacji procesu. Przełączaniu domen odpowiada wysyłanie przez jakiś proces komunikatu do innego procesu i oczekивание na odpowiedź.
- Domeną może być każda *procedura*. W tym przypadku zbiór obiektów, do których można mieć dostęp, odpowiada lokalnym zmiennym zdefiniowanym w danej procedurze. Przełączanie domen następuje wraz z wywołaniem procedury.

W dalszej części tego rozdziału omówimy przełączanie domen bardziej szczegółowo.

19.2.2 Przykłady

Rozważmy standardowy, działający w dwóch trybach (tryb monitora i użytkownika) model systemu operacyjnego. Gdy proces działa w trybie monitora, może wówczas wykonywać rozkazy uprzywilejowane, sprawując pełną kontrolę nad systemem komputerowym. Natomiast jeśli proces działa w trybie użytkownika, to może wykonywać tylko rozkazy nieuprzywilejowane. W rezultacie może on działać tylko w swoim, zawsze określonym obszarze pamięci. Te dwa tryby chronią system operacyjny (działający w domenie monitora) przed procesami użytkowników (działającymi w domenie użytkownika). W wieloprogramowym systemie operacyjnym nie wystarczą dwie domeny ochrony, ponieważ użytkownicy wymagają także ochrony wzajemnie przed sobą. Dlatego jest potrzebny bardziej skomplikowany schemat. Na przykładzie dwu znaczących systemów operacyjnych UNIX i MULTICS postaramy się ten schemat zilustrować i przeanalizować implementację omawianych koncepcji.

19.2.2.1 UNIX

W systemie operacyjnym UNIX domena jest związana z użytkownikiem. Przełączaniu domen odpowiada czasowa zmiana identyfikacji użytkownika. Zmiana ta jest wykonywana przez system plików w sposób następujący. Z ka-

dym plikiem jest związany identyfikator właściciela i bit domeny, czyli tzw. *bit ustanowienia identyfikatora użytkownika* (ang. *setuid bit*). Gdy użytkownik (z identyfikatorem *user-id = A*) rozpoczyna wykonanie pliku będącego własnością użytkownika *B*, którego bit domeny jest wyzerowany, to identyfikator *user-id* procesu jest określony jako *A*. Jeśli bit domeny jest ustawiony na 1, to identyfikator użytkownika (*user-id*) przybiera postać identyfikatora użytkownika *B*, czyli właściciela pliku. Kiedy proces kończy działanie, wtedy ta chwilowa zmiana identyfikatora użytkownika przestaje obowiązywać.

W systemach operacyjnych, w których domeny definiuje się za pomocą identyfikatorów użytkowników, stosuje się inne, popularne metody zmiania domen, jako że prawie wszystkie systemy wymagają takiego mechanizmu. Mechanizm tego rodzaju jest stosowany wówczas, gdy jakieś wymagające uprzywilejowania udogodnienie ma być przyznawane ogólni użytkownikom. Może być na przykład pożądane zezwolenie użytkownikom na korzystanie z sieci, jednakże bez pozwalań im na pisanie własnych programów współpracy z siecią. W takim przypadku w systemie UNIX bit ustanowienia identyfikatora użytkownika (*setuid*) programu organizującego współpracę z siecią ma wartość 1, powodując zmianę identyfikatora użytkownika na czas pracy tego programu. Identyfikator użytkownika (*user-id*) przyjmuje czasowo wartość identyfikatora użytkownika mającego przywilej współpracy z siecią (np. identyfikatora korzenia drzewa katalogów (ang. *root*), czyli użytkownika o największych możliwościach w systemie). Trudnością w tej metodzie jest to, że jeśli użytkownikowi uda się utworzyć plik z identyfikatorem *user-id* równym „*root*” i bitem domeny *setuid* ustawionym na 1, to użytkownik taki zyskuje władzę użytkownika „*root*” i może zrobić w systemie wszystko i wszędzie. Więcej informacji na temat mechanizmu *setuid* zawiera rozdz. 21.

Odmiana tej metody, używana w innych systemach operacyjnych, polega na umiejscowieniu programów uprzywilejowanych w specjalnym katalogu. System operacyjny powinien być tak skonstruowany, aby zmieniał identyfikatory użytkowników dla każdego programu wykonywanego z tego katalogu – albo na równoważne identyfikatorowi „*root*”, albo na identyfikator właściciela tego katalogu. Unika się tym samym kłopotu z identyfikatorem *setuid* odnoszącym się do tajnych programów, ponieważ wszystkie takie programy są skupione w jednym miejscu. Jednak ta metoda jest mniej elastyczna niż stosowana w systemie UNIX.

Jeszcze bardziej restrykcyjne, a zatem o silniejszej ochronie, są systemy, w których po prostu zakazuje się zmian identyfikatora użytkownika. W takich systemach w celu umożliwienia użytkownikom korzystania z uprzywilejowanych udogodnień stosuje się specjalne techniki. Na przykład podczas rozruchu systemu może być inicjowany proces-demon, działający pod specjalnym identyfikatorem użytkownika. Użytkownicy chcący skorzystać z uprzywile-

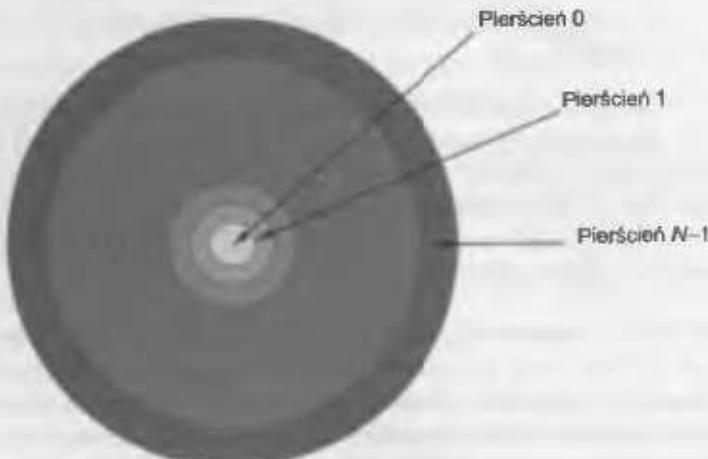
jowanych możliwości wykonują wtedy osobny program, który wysyła zamówienia do tego procesu. Metodę tę zastosowano w systemie operacyjnym TOPS-20.

W każdym z tych systemów pisanie programów uprzywilejowanych wymaga wielkiej pieczy. Kazde przeoczenie lub niestaranność może spowodować całkowitą utratę ochrony systemu. Programy tego rodzaju z reguły pierwsze padają ofiarą ataków ze strony ludzi usiłujących włamać się do systemu. Co gorsza, napastnikom tym często się to udaje. Znane są na przykład liczne przypadki naruszeń bezpieczeństwa w systemach uniksowych z powodu właściwości setuid. Problemami bezpieczeństwa zajmujemy się w rozdz. 20.

19.2.2.2 MULTICS

W systemie MULTICS domeny ochrony są zorganizowane hierarchicznie w strukturę pierścieniową. Każdy pierścień odpowiada pojedynczej domenie (rys. 19.2). Pierścienie są ponumerowane od 0 do 7. Niech D_i i D_j będą dwoma pierścieniami domen. Jeśli $j < i$, to D_i jest podzbiorem D_j . Oznacza to, że proces działający w domenie D_i ma więcej przywilejów niż proces działający w domenie D_j . Proces działający w domenie D_0 jest najbardziej uprzywilejowany. Jeśli istnieją tylko dwa pierścienie, to schemat ten jest równoważny trybom wykonania monitor-użytkownik, przy czym trybowi monitora odpowiada D_0 , a trybowi użytkownika odpowiada D_1 .

System MULTICS ma segmentowaną przestrzeń adresową – każdy segment jest plikiem. Każdy segment jest skojarzony z jednym z pierścieni. Opis segmentu zawiera pozycję identyfikującą numer pierścienia. Ponadto są w nim



Rys. 19.2 Struktura pierścieniowa systemu MULTICS

trzy bity dostępu, służące do kontroli czytania, pisania i wykonywania. Związki między segmentami a pierścieniami należą do sfery polityki, którą nie zajmujemy się w tej książce. Z każdym procesem jest skojarzony *licznik bieżącego numeru pierścienia* określający pierścień, w którym proces jest w danej chwili wykonywany. Gdy proces działa w pierścieniu i , wówczas nie ma dostępu do segmentu przypisanego do pierścienia j , jeśli $j < i$. Ale może kontaktować się z segmentem przypisanym do pierścienia k , jeśli $k \geq i$. Jednak typ dostępu jest ograniczony stosownie do bitów dostępu związanych z danym segmentem.

Przelaczanie domen w systemie MULTICS występuje wtedy, kiedy proces przechodzi z jednego pierścienia do innego wskutek wywołania procedury z innego pierścienia. Oczywiście przełączenie to musi się dokonać w sposób kontrolowany. W przeciwnym razie proces mógłby przejść do działania w pierścieniu 0, zrywając wszelką ochronę. W celu kontroli przełączania domen modyfikuje się pole pierścienia w deskryptorze segmentu tak, aby zawierało:

- **Ograniczniki dostępu:** jest to para liczb całkowitych (b_1, b_2) , takich że $b_1 \leq b_2$.
- **Limit:** jest to liczba całkowita b_3 , taka że $b_3 > b_1$.
- **Wykaz bram:** określa punkty wejścia (bramy), w których można wywoływać segmenty.

Jeśli proces wykonywany w pierścieniu i wywołuje procedurę (segment) z ogranicznikami dostępu (b_1, b_2) , to wywołanie jest dozwolone wówczas, gdy $b_1 \leq i \leq b_2$ oraz bieżący numer pierścienia w procesie pozostaje równy i . W przeciwnym razie powstaje przerwanie, które system operacyjny obsługuje w następujący sposób:

- Jeśli $i < b_1$, to wywołanie zostaje zaakceptowane, gdyż przejście następuje do pierścienia (domeny) mniej uprzywilejowanego. Jeżeli jednak są przekazywane parametry odnoszące się do segmentów w głębszym pierścieniu (tzn. do segmentów niedostępnych dla wywoływanej procedury), to muszą one zostać skopiowane do obszaru, który będzie dostępny dla wywoywanej procedury.
- Jeśli $i > b_2$, to wywołanie jest dozwolone tylko wtedy, gdy b_3 jest mniejsze lub równe i oraz wywołanie jest skierowane do jednego z punktów wejściowych wyznaczonych na wykazie bram. Schemat ten pozwala procesom z ograniczonymi prawami dostępu wywoływać procedury z głębszych pierścieni, mające więcej praw dostępu, przy czym dokonuje się tego pod staranną kontrolą.

Podstawową wadą struktury pierścieniowej (hierarchicznej) jest to, że nie pozwala ona na wymuszanie zasady wiedzy koniecznej. Oto przykład: jeżeli jakiś obiekt ma być dostępny w domenie D_i , lecz niedostępny w domenie D_j , to musi być $j < i$. Jednak to oznacza, że każdy segment dostępny w D_i staje się również dostępny w D_j .

Ochrona w systemie MULTICS jest, ogólnie biorąc, bardziej złożona i mniej wydajna niż rozwiązania przyjęte w dzisiejszych systemach operacyjnych. Jeżeli ochrona koliduje z łatwością użycia systemu lub istotnie obniża jego wydajność, to jej zastosowanie należy starannie rozważyć w świetle zadań systemu. Na przykład można uznać za rozsądne rozporządzanie złożonym systemem ochrony w komputerze uniwersyteckim, stosowanym do przetwarzania studenckich prac końcowych oraz do prac prowadzonych przez studentów na ćwiczeniach. Podobny system ochrony nie byłby odpowiedni dla komputera używanego do wyjątkowo złożonych obliczeń, w którym wydajność wysuwa się na plan pierwszy. Z tego powodu jest dobrze, jeżeli mechanizm ochrony jest odseparowany od zasad jej stosowania, gdyż umożliwia w tym samym systemie stosowanie złożonej lub prostej ochrony w zależności od potrzeb jego użytkowników. Aby móc oddzielać mechanizm od polityki, potrzebujemy ogólniejszych modeli ochrony.

19.3 ■ Macierz dostępów

Nasz model ochrony można rozważyć abstrakcyjnie jako macierz, zwaną *macierzą dostępów* (ang. *access matrix*). Wiersze macierzy dostępów reprezentują domeny, a kolumny – obiekty. Każdy element macierzy zawiera zbiór praw dostępu. Ponieważ obiekty są zdefiniowane jawnie za pomocą nazw kolumn, można pominąć nazwę obiektu w prawie dostępu. Element $\text{dostęp}(i, j)$ określa zbiór operacji, które proces, działający w domenie D_i , może wykonywać na obiekcie O_j .

W celu zilustrowania tych koncepcji rozpatrzymy macierz dostępów pokazaną na rys. 19.3. Zawiera ona cztery domeny i cztery obiekty: trzy pliki (F_1, F_2, F_3) i jedną drukarkę. Gdy proces działa w domenie D_1 , to może czytać pliki F_1 i F_3 . Proces działający w domenie D_4 ma takie same przywileje, jakie należą mu się w domenie D_1 , lecz dodatkowo może on jeszcze pisać do plików F_1 i F_3 . Zauważmy, że drukarka może być dostępna tylko dla procesu działającego w domenie D_2 .

Macierz dostępów dostarcza mechanizmu, który pozwala podejmować decyzje polityczne. Mechanizm ten polega na implementowaniu macierzy dostępów i gwarantowaniu, że określone przez nas własności semantyczne rzeczywiście będą zachowywane. Mówiąc dokładniej, musi być zagwaranto-

Obiekt Domena	F_1	F_2	F_3	Drukarka
D_1	czytaj		czytaj	
D_2				drukuj
D_3		czytaj	wykonaj	
D_4	czytaj pisz		czytaj pisz	

Rys. 19.3 Macierz dostępów

wane, że proces działający w domenie D_i może mieć dostęp tylko do obiektów wyspecyfikowanych w wierszu i , wyłącznie w sposób dozwolony przez wpisy w macierzy dostępów.

Za pomocą macierzy dostępów można realizować decyzje polityczne dotyczące ochrony. Decyzje te dotyczą praw, które powinny znaleźć się w elemencie (i, j) tej macierzy. Należy również określić domenę działania każdego procesu. Pozostaje to na ogół w gestii systemu operacyjnego.

Użytkownicy zazwyczaj decydują o zawartości elementów macierzy dostępów. Gdy użytkownik tworzy nowy obiekt O_i , wówczas do macierzy dostępów jest dodawana kolumna O_i z elementami określonymi zgodnie z życzeniami twórcy obiektu. Stosownie do potrzeb, użytkownik może decydować o umieszczeniu w kolumnie j pewnych praw w jednych elementach i innych praw w innych elementach macierzy.

Mechanizm macierzy dostępów wystarcza do definiowania i implementowania ścisłej kontroli zarówno statycznych, jak i dynamicznych związków między procesami a domenami. Jeśli przełączamy proces z jednej domeny do drugiej, to wykonujemy operację (przełącz) na obiekcie (domenie). Przelaczanie domen można kontrolować przez zaliczenie domen do obiektów macierzy dostępów. Podobnie, zmiana macierzy dostępów oznacza operację na obiekcie, jakim jest macierz dostępów. I znów – zmiany te można nadzorować dzięki ustanowieniu samej macierzy dostępów obiektem macierzy dostępów. Ponieważ każdy element macierzy dostępów można zmieniać z osobna, w rzeczywistości musimy traktować każdy element tej macierzy jako obiekt podlegający ochronie.

Teraz musimy tylko rozważyć operacje, które można wykonywać na tych nowych obiektach (domenach i macierzy dostępów) i postanowić, w jaki sposób będą je mogły wykonywać procesy.

Procesy powinny być w stanie przechodzić z jednej domeny do drugiej. Przelaczanie domeny z domeny D_i do domeny D_j może nastąpić wtedy i tylko

Obiekt		F_1	F_2	F_3	Drukarka	D_1	D_2	D_3	D_4
Domena									
D_1	czytaj			czytaj			przelacz		
D_2					drukuj			przelacz	przelacz
D_3			czyta(j)	wykonaj					
D_4	czytaj pisz			czytaj pisz		przelacz			

Rys. 19.4 Macierz dostępów z rys. 19.3 z domenami w roli obiektów

wtedy, gdy prawo dostępu $przelacz \in \text{dostęp}(i, j)$. Zgodnie zatem z macierzą dostępów pokazaną na rys. 19.4 proces działający w domenie D_1 może przelać się do domeny D_3 albo do domeny D_4 . Proces w domenie D_4 może przelać się do domeny D_1 , a proces w domenie D_1 może przejść do domeny D_2 .

Umożliwienie kontrolowanych zmian zawartości macierzy dostępów wymaga trzech dodatkowych praw: *kopiowania, właściciela i kontroli* (ang. *copy, owner, control*).

Możliwość kopiowania prawa dostępu z jednej domeny (wiersza) w macierzy dostępów do drugiej została oznaczona gwiazdką (*) umieszczoną obok prawa dostępu. Prawo *kopiowania* pozwala na skopiowanie prawa dostępu tylko w obrębie kolumny (tzn. dla obiektu), dla której dane prawo zostało zdefiniowane. Na przykład na rys. 19.5(a) proces działający w domenie D_2 może skopiować operację czytania do dowolnego elementu macierzy związanego z plikiem F_2 . W ten sposób macierz dostępów z rys. 19.5(a) może przyjąć postać przedstawioną na rys. 19.5(b).

Są dwa warianty tego schematu:

1. Prawo jest kopiowane z pola $\text{dostęp}(i, j)$ do pola $\text{dostęp}(k, j)$, a następnie usuwa się je z pozycji $\text{dostęp}(i, j)$. Nie jest to więc kopiowanie, lecz przekazanie prawa dostępu.
2. Przenoszenie prawa kopiowania może być ograniczone. Oznacza to, że jeśli prawo R^* jest kopiowane z pola $\text{dostęp}(i, j)$ do pola $\text{dostęp}(k, j)$, to tworzy się tylko prawo R (a nie R^*). Proces działający w domenie D_k nie może dalej kopiować prawa R .

W systemie może wystąpić tylko jedno z tych trzech praw *kopiowania* lub mogą istnieć trzy odrębne prawa: *kopiowanie, przekazanie i ograniczone kopiowanie*.

Obiekt Domena	F_1	F_2	F_3
D_1	wykonaj		pisz*
D_2	wykonaj	czytaj*	wykonaj
D_3	wykonaj		

(a)

Obiekt Domena	F_1	F_2	F_3
D_1	wykonaj		pisz*
D_2	wykonaj	czytaj*	wykonaj
D_3	wykonaj	czytaj	

(b)

Rys. 19.5 Macierz dostępów z prawami kopiowania

Prawo kopiowania pozwala procesowi na kopowanie niektórych praw z jednego pola do innego w tej samej kolumnie macierzy. Jest potrzebny także mechanizm pozwalający na dodawanie nowych praw i usuwanie pewnych praw. Operacje te wymagają prawa *właściciela*. Jeśli element $\text{dostęp}(i, j)$ zawiera prawo *właściciela*, to proces działający w domenie D_i może dodawać bądź usuwać dowolne prawa do elementów w kolumnie j . Na przykład na rys. 19.6(a) domena D_1 jest właściwicielem pliku F_1 , a zatem w kolumnie F_1 może dodać lub usunąć każde dozwolone prawo. Podobnie, domena D_2 jest właściwicielem F_2 i F_3 , może zatem dodać i usunąć dowolne dozwolone prawo wewnętrz tych kolumn. Tak więc macierz dostępów z rys. 19.6(a) można zmodyfikować do postaci pokazanej na rys. 19.6(b).

Prawa *kopiowania* i *właściciela* pozwalają procesowi zmieniać elementy kolumn. Jest także potrzebny mechanizm zmieniania elementów w wierszu. Prawo *kontroli* jest stosowalne tylko w odniesieniu do obiektów będących domenami. Jeśli pole $\text{dostęp}(i, j)$ zawiera prawo kontroli, to proces pracujący w domenie D_i może usunąć dowolne prawo dostępu z wiersza j . Założymy, że na rys. 19.4 dołączliśmy prawo kontroli do elementu $\text{dostęp}(D_2, D_4)$. Wówczas proces działający w domenie D_2 może zmienić domenę D_4 tak, jak to widać na rys. 19.7.

Obiekt		F_1	F_2	F_3
Domena				
D_1	właściciel wykonaj			pisz
D_2			czytaj* właściciel	czytaj* właściciel pisz*
D_3	wykonaj			

(a)

Obiekt		F_1	F_2	F_3
Domena				
D_1	właściciel wykonaj			
D_2			właściciel czytaj* pisz*	czytaj* właściciel pisz*
D_3			pisz	pisz

(b)

Rys. 19.6 Macierz dostępów z prawami właściciela

Chociaż prawa kopiowania i właściciela tworzą mechanizm umożliwiający ograniczanie rozchodzenia się praw dostępu, to jednak nie są one dostateczne na to, aby zapobiegać rozchodzeniu się informacji (czyli jej ujawnianiu). Zagwarantowanie, że żadna informacja przechowywana na początku w obiekcie nie wydostanie się na zewnątrz jego środowiska wykonania nazywa się *problemem zamknięcia* (ang. *confinement problem*). Zagadnienie to nie jest w ogólności rozwiązywalne (zob. uwagi bibliograficzne).

Obiekt		F_1	F_2	F_3	Drukarka laserowa	D_1	D_2	D_3	D_4
Domena									
D_1	czytaj			czytaj				przelacz	
D_2					drukuj			przelacz	przelacz kontroluj
D_3			czytaj	wykonaj					
D_4	pisz			pisz		przelacz			

Rys. 19.7 Zmodyfikowana macierz dostępów z rys. 19.4

Operacje na domenach i macierzy dostępów nie mają samego szczególnego znaczenia. Ważniejsze jest to, że ilustrują one przydatność modelu macierzy dostępów do implementowania i kontroli ochrony dynamicznej. Nowe obiekty i nowe domeny można tworzyć dynamicznie i włączać do modelu macierzy dostępów. Przedstawiliśmy jednak tylko podstawowy mechanizm; decyzje polityczne rozstrzygające o tym, które domeny mają mieć dostęp do jakich obiektów i w jaki sposób, muszą zapadać w gronie projektantów systemu i jego użytkowników.

19.4 ■ Implementacja macierzy dostępów

Jak efektywnie zaimplementować macierz dostępów? Z zasady będzie to macierz rozrzedzona w tym sensie, że większość jej elementów będzie pusta. Istnieją odpowiednie struktury danych służące do reprezentowania macierzy rozrzedzonych, lecz ich przydatność w omawianym zastosowaniu jest niewielka ze względu na sposób korzystania ze schematu ochrony.

19.4.1 Tablica globalna

Najprostszą realizacją macierzy dostępów jest tablica globalna, składająca się ze zbioru trójk uporządkowanych $\langle \text{domena}, \text{obiekt}, \text{zbiór-praw} \rangle$. Za każdym razem, gdy operacja M ma być wykonana na obiekcie O_i w domenie D_j , w tablicy globalnej szuka się trójki $\langle D_j, O_i, R_k \rangle$, gdzie $M \in R_k$. Jeśli trójka taka zostanie odnaleziona, to operację można wykonywać. W przeciwnym razie powstaje sytuacja wyjątkowa (błąd). Implementacja ta ma kilka wad. Tablica jest zazwyczaj tak wielka, że nie może znajdować się w pamięci operacyjnej, co powoduje konieczność wykonywania dodatkowych operacji wejścia-wyjścia. Do zarządzania taką tablicą stosuje się często techniki pamięci wirtualnej. W dodatku trudno jest korzystać z możliwości specjalnego grupowania obiektów lub domen. Na przykład, jeśli jakiś obiekt ma być dostępny do czytania dla wszystkich, to musi osobno występować w każdej domenie.

19.4.2 Wykazy dostępów do obiektów

Każda kolumna w macierzy dostępów może przybrać postać wykazu dostępów do danego obiektu, jak to opisaliśmy w p. 10.4.2. Rzecz oczywista, można pominąć puste pozycje. W wynikowym wykazie każdemu obiektyowi będą odpowiadać pary uporządkowane $\langle \text{domena}, \text{zbiór-praw} \rangle$, które definiują wszystkie domeny z niepustymi zbiorami praw dostępu do danego obiektu.

Podejście to można łatwo rozszerzyć, dodając do zdefiniowanego wyróżku domyślny zbiór praw dostępu. Kiedy pojawi się próba wykonania operacji M na obiekcie O , w domenie D_i , wtedy w wykazie dostępów szuka się obiektu O , i związanej z nim pozycji $\langle D_i, R_k \rangle$, takiej że $M \in R_k$. Jeśli taka pozycja zostanie odnaleziona, to udziela się zezwolenia na wykonanie operacji. Gdy nie ma odpowiedniej pozycji w wykazie, wówczas przeszukuje się domyślny zbiór praw dostępu. Jeśli operacja M jest w tym zbiorze, to zezwala się na dostęp. W przeciwnym razie następuje odmowa dostępu i powstaje sytuacja wyjątkowa. Zwróćmy uwagę, że aby polepszyć skuteczność, zbiór domyślny można przeglądać w pierwszej kolejności, a dopiero po nim wykaz dostępów.

19.4.3 Wykazy uprawnień do domen

Zamiast kojarzyć kolumny macierzy dostępów z obiektami – jako wykazy dostępów, można każdy wiersz tej macierzy powiązać z jego domeną. Wykaz uprawnień (ang. *capability list*) domeny jest spisem obiektów i operacji dozwolonych do wykonania na tych obiektach. Obiekt jest często reprezentowany przez jego nazwę fizyczną lub adres zwany *uprawnieniem* (ang. *capability*). W celu wykonania operacji M na obiekcie O , proces zaopatruje operację M w parametr określający uprawnienie (wskaźnik) do obiektu O . Aby uzyskać dostęp do obiektu, wystarcza po prostu mieć uprawnienie.

Wykaz uprawnień jest związany z domeną, lecz nigdy nie jest dostępny bezpośrednio dla procesu działającego w tej domenie. W zamian sam wykaz uprawnień jest obiektem chronionym, utrzymywany przez system operacyjny i dostępnym dla użytkownika tylko pośrednio. Ochrona sprawowana na podstawie uprawnień polega na fakcie, że uprawnienia nigdy nie mogą przedostać się do obszaru pamięci dostępnego bezpośrednio dla procesu użytkownika (w którym mogłyby zostać zmienione). Jeśli wszystkie uprawnienia są bezpieczne, to chronione przez nie obiekty są również zabezpieczone przed nieautoryzowanym dostępem.

Pierwotnie uprawnienia miały być pewnego rodzaju chronionymi wskaźnikami. Za ich pomocą chciało zapewnić ochronę zasobów, na którą pojawiło się zapotrzebowanie wraz z nastaniem wieloprogramowych systemów komputerowych. Idea wewnętrznie chronionego wskaźnika (z punktu widzenia użytkownika systemu) jest podstawą ochrony, którą można rozszerzać aż do poziomu aplikacji.

Aby zapewnić wewnętrzną ochronę, należy odróżniać uprawnienia od obiektów innych rodzajów oraz interpretować je za pomocą maszyny abstrakcyjnej, na której są wykonywane programy wyższego poziomu. Uprawnienia są zwykle odróżniane od innych danych w jeden z następujących sposobów:

- Kazdy obiekt ma *znacznik* (ang. *tag*) określający jego typ jako uprawnienia albo udostępniane dane. Sam znacznik nie może być dostępny bezpośrednio dla programów użytkowych. Ograniczenie to może być wymuszone środkami sprzętowymi lub sprzętowo-programowymi. Choć do rozróżnienia między uprawnieniami a innymi obiektem wystarcza tylko 1 bit, używa się często więcej bitów. To rozszerzenie pozwala na oznaczanie typów obiektów przez sprzęt. Tym samym za pomocą znaczników sprzęt może rozróżniać liczby całkowite, zmiennopozycyjne, wskaźniki, wartości boolowskie, znaki, rozkazy, uprawnienia oraz obiekty o nie nadanych wartościach początkowych.
- W rozwiązaniu alternatywnym przestrzeni adresowej związaną z programem można podzielić na dwie części. Jedna część jest dostępna dla programu i zawiera zwykłe dane programowe i instrukcje. Druga część, zawierająca wykaz uprawnień, jest dostępna tylko dla systemu operacyjnego. W realizacji takiego podejścia jest użyteczna pamięć segmentowana (p. 8.6).

Opracowano kilka systemów, w których ochrona polega za stosowaniem uprawnień. Opisujemy je krótko w p. 19.6. Odmię ochrony z wykorzystaniem uprawnień zastosowano również w systemie operacyjnym Mach – jej opis znajduje się w sekcji A.3*.

19.4.4 Mechanizm zamka-klucza

Mechanizm zamka-klucza (ang. *lock-key scheme*) jest czymś pośrednim między wykazami dostępów a wykazami uprawnień. Kazdy obiekt ma wykaz jednoznacznych wzorców binarnych, zwanych *zamkami* (ang. *locks*). Podobnie, każda domena ma wykaz jednoznacznych wzorców binarnych, zwanych *kluczami* (ang. *keys*). Proces działający w domenie może mieć dostęp do obiektu tylko wtedy, gdy dana domena zawiera klucz pasujący do jednego z zamków tego obiektu.

Tak jak w przypadku wykazów uprawnień, wykazy kluczowe domeny muszą być zarządzane przez system operacyjny na zamówienie danej domeny. Użytkownicy nie mają prawa bezpośrednio sprawdzać ani zmieniać zamków (lub kluczów).

19.4.5 Porównanie

Wykazy dostępów odpowiadają bezpośrednio zapotrzebowaniom użytkowników. Gdy użytkownik tworzy obiekt, może wówczas określić, które domeny

* Chodzi o materiały dostępne w sieci Internet. W książce sekcja A.3 nie występuje – Przyp. tłum.

mają mieć dostęp do tego obiektu i za pomocą jakich operacji. Jednakże z uwagi na to, że informacja o prawach dostępu w poszczególnych domenach nie występuje w jednym miejscu, określanie zbioru praw dla każdej domeny jest trudne. Ponadto każdy dostęp do obiektu musi podlegać sprawdzaniu, co wymaga przeszukiwania wykazu dostępów. W wielkich systemach, z długimi wykazami dostępów, przeszukiwanie takie może być czasochłonne.

Wykazy uprawnień nie odpowiadają wprost potrzebom użytkowników. Niemniej jednak są one użyteczne dzięki lokalizowaniu informacji dotyczącej poszczególnych procesów. Proces usiłujący uzyskać dostęp do obiektu musi wykazać, że ma odpowiednie uprawnienia. Zatem system ochrony musi tylko sprawdzić, czy te uprawnienia są ważne. Cofanie uprawnień bywa jednak niewydajne (p. 19.5).

Mechanizm zamka-klucza zajmuje pozycję pośrednią między obydwoma schematami. Mechanizm ten może być zarówno efektywny, jak i elastyczny – zależnie od długości klucza. Klucze można swobodnie przekazywać od domeny do domeny. Ponadto przywileje dostępów można łatwo uaktualniać za pomocą prostej techniki ziniany niektórych kluczy przypisanych do obiektu (p. 19.5).

W większości systemów stosuje się kombinację wykazów dostępów i uprawnień. Kiedy proces po raz pierwszy próbuje uzyskać dostęp do obiektu, wtedy przegląda się wykaz dostępów. Jeśli dostęp jest zabroniony, to powstaje sytuacja wyjątkowa. W przeciwnym razie tworzy się wykaz uprawnień, który dołącza się do procesu. Przy następnych odniesieniach, w celu sprawnego wylegitymowania się swoimi uprawnieniami, proces korzysta z wykazu uprawnień. Po ostatnim dostępie uprawnienia zostają cofnięte. Strategię tę zastosowano w systemach MULTICS oraz CAL – używa się w nich wykazów zarówno dostępów, jak i uprawnień.

Jako przykład rozważmy system plików. Każdemu plikowi jest przyporządkowany wykaz dostępów. Jeśli proces otwiera plik, to przeszukuje się strukturę katalogową w celu znalezienia pliku, sprawdzenia praw dostępu i przydzielenia buforów. Całą tę informację zapisuje się jako nowy element tablicy plików należącej do procesu. Operacja ta daje w wyniku indeks elementu tablicy, dotyczącego nowo otwartego pliku. We wszystkich dalszych operacjach na pliku następuje podanie indeksu do tablicy plików. Wpis w tablicy plików zawiera wskaźnik do pliku i jego buforów. Gdy plik zostaje zamknięty, wówczas odpowiadający mu wpis usuwa się z tablicy plików. Ponieważ na tablicy plików działa system operacyjny, użytkownik nie może jej uszkodzić. Użytkownik ma dostęp jedynie do tych plików, które zostały otwarte. Ochrona pliku jest zatem zapewniona, ponieważ dostęp jest sprawdzany przy otwarciu pliku. Opisaną strategię zastosowano w systemie UNIX.

Zauważmy, że prawo do dostępu musi być sprawdzane przy każdym dostępie, a wpis w tablicy plików zawiera wykaz uprawnień do wykonywania

tylko dozwolonych operacji. Jeżeli plik otwiera się do czytania, to w elemencie tablicy plików umieszcza się uprawnienie do dostępu do czytania. Jeśli usiłowano by wpisać coś do takiego pliku, to naruszenie ochrony zostałoby wykryte wskutek porównania żądanej operacji z wykazem uprawnień w tablicy plików.

19.5 ■ Cofanie praw dostępu

W systemie ochrony dynamicznej może być czasami niezbędne cofanie praw dostępu do obiektów dzielonych przez różnych użytkowników. Sposób cofania praw dostępu nasuwa różne pytania:

- **Natychmiast czy z opóźnieniem:** Czy cofanie praw następuje natychmiast, czy też jest opóźniane? Jeśli cofanie praw jest opóźnione, to czy można przewidzieć, kiedy ono nastąpi?
- **Wybiórczo czy ogólnie:** Gdy dochodzi do cofnięcia prawa dostępu do obiektu, to czy dotyczy ono wszystkich użytkowników, którzy mieli to prawo dostępu do danego obiektu, czy też można określić grupę użytkowników, którym dane prawo powinno być odebrane?
- **Częściowo czy całkowicie:** Czy można unieważnić podzbiór praw związanych z obiektem, czy też należy cofnąć wszystkie prawa dotyczące danego obiektu?
- **Czasowo czy na stałe:** Czy dostęp jest cofany na stałe (tzn. cofniętego prawa dostępu nigdy już nie będzie można osiągnąć), czy też dostęp można unieważnić i uzyskać go później z powrotem?

Jeśli zastosuje się schemat wykazu dostępów, to cofanie praw będzie całkiem proste. Wyszukuje się na wykazie dostępów prawo (lub prawa) do unieważnienia i usuwa je z wykazu. Unieważnienie jest natychmiastowe i może być ogólne lub wybiorcze, całkowite lub częściowe, stałe lub czasowe.

Natomiast w przypadku uprawnień zadanie cofania praw jest o wiele bardziej skomplikowane. Ponieważ uprawnienia są rozproszone po całym systemie, należy je wpierw odnaleźć. Jest kilka różnych schematów realizacji cofania uprawnień. Należą do nich m.in.:

- **Wtórne pozyskiwanie:** Uprawnienia są okresowo usuwane z każdej domeny. Jeżeli proces potrzebuje danego uprawnienia, to wykrywa jego usunięcie i może wtedy spróbować uzyskać je na nowo. Jeżeli uprawnienie zostało cofnięte, to proces nie będzie mógł uzyskać go ponownie.

- **Wskazniki zwrotne:** Z każdym obiektem jest utrzymywany wykaz wskazników do wszystkich związanych z nim uprawnień. Gdy jest wymagane cofnięcie jakiegoś prawa, można wówczas, posługując się tymi wskaznikami, zmieniać uprawnienia stosownie do potrzeb. Schemat ten przyjęto w systemie MULTICS. Jest on całkiem ogólny, jednak jego realizacja jest kosztowna.
- **Adresowanie pośrednie:** Uprawnienia nie wskazują na obiekty wprost, lecz pośrednio. Kazde uprawnienie wskazuje na jednoznaczny wpis w tablicy globalnej, który z kolei wskazuje na obiekt. W celu cofnięcia uprawnienia przeszukuje się tablicę globalną i usuwa z niej odpowiedni element. Gdy pojawia się próba dostępu, wówczas okazuje się, że uprawnienie wskazuje na niedozwolony element. Elementy tablicy mogą być bez kłopotu użyte powtórnie do innych uprawnień, ponieważ zarówno uprawnienie, jak i wpis w tablicy zawiera jednoznaczną nazwę obiektu. Obiekt oznaczony przez uprawnienie oraz odpowiadający mu wpis w tablicy muszą do siebie pasować. Metodę tę zaadaptowano w systemie CAL. Nie pozwala ona na selektywne cofanie praw.
- **Klucze:** Klucz jest jednoznacznym wzorcem binarnym, który można powiązać z wszelkimi uprawnieniami. Klucz taki jest definiowany przy tworzeniu uprawnień, przy czym proces mający dane uprawnienia nie może klucza sprawdzić ani zmienić. *Klucz główny* (ang. *master key*), związany z każdym obiektem, może być zdefiniowany lub zastąpiony za pomocą operacji *ustal klucz* (ang. *set-key*). Tworzonym uprawnieniom przyporządkowuje się bieżącą wartość klucza głównego. Podeczas sprawdzania uprawnień porównuje się ich klucz z kluczem głównym. Jeśli klucze pasują do siebie, to zezwala się na wykonanie operacji. W przeciwnym razie następuje zgłoszenie wyjątku. Cofnięcie uprawnień polega na zastąpieniu klucza głównego nową wartością za pomocą operacji *ustal klucz*, co delegalizuje wszystkie dotychczasowe uprawnienia dotyczące danego obiektu.

Zauważmy, że ta metoda nie pozwala na selektywne cofanie uprawnień, gdyż z każdym obiektem jest związany tylko jeden klucz główny. Jeśli z obiektami skojarzymy listę kluczy, to realizacja wybiórczego cofania uprawnień stanie się możliwa. Można też zgrupować wszystkie klucze w jednej, globalnej tablicy kluczy. Prawomocność uprawnień jest wtedy równoznaczna z dopasowaniem ich klucza do jakiegoś klucza w tablicy globalnej. Cofanie uprawnień polega na usunięciu dopasowanego klucza z tablicy. Za pomocą tego schematu jeden klucz można przypisać do kilku obiektów, co daje maksimum elastyczności.

Operacje definiowania kluczy, umieszczania ich na listach oraz usuwania z list nie powinny być dostępne dla wszystkich użytkowników. Za rozsądne można by w szczególności uznać pozwolenie na ustalanie kluczy tylko właścielowi danego obiektu. Jest to jednak decyzja ze sfery polityki, którą system ochrony może realizować, lecz nie powinien jej określać.

19.6 ■ Systemy działające na zasadzie uprawnień

W tym punkcie dokonujemy krótkiego przeglądu dwóch systemów ochrony działających na zasadzie uprawnień. Systemy te różnią się pod względem złożoności oraz reguł ochrony, które można wdrażać przy ich udziale. Żaden z nich nie jest szeroko używany, jednak są one ciekawym terenem do udowadniania teoretycznych aspektów ochrony.

19.6.1 System Hydra

Hydra jest systemem ochrony działającym na zasadzie uprawnień, charakteryzującym się dość dużą elastycznością. System dostarcza stałego zbioru możliwych praw dostępu, które zna i interpretuje. Wśród praw tych znajdują się takie podstawowe rodzaje dostępu, jak prawo do czytania, pisania lub wykonywania segmentu pamięci. Ponadto system zawiera środki umożliwiające użytkownikowi (korzystającemu z systemu ochrony) deklarowanie dodatkowych praw. Interpretacja praw zdefiniowanych przez użytkownika zależy wyłącznie od programu użytkownika, lecz system umożliwia ochronę dostępu przy korzystaniu z tych praw na takich samych zasadach, jak w stosunku do praw zdefiniowanych w nim samym. Możliwości systemu są dość interesujące i stanowią istotny postęp w technice ochrony.

Operacje na obiektach są zdefiniowane proceduralnie. Procedury, które wykonują te operacje, są też pewnego rodzaju obiektami i dostęp do nich odbywa się pośrednio, za pomocą uprawnień. Jeśli system ochrony ma zajmować się obiektami o typach zdefiniowanych przez użytkownika, to należy mu przedłożyć nazwy procedur zdefiniowanych przez użytkownika. Kiedy Hydra pozna definicję obiektu, wtedy nazwy przewidzianych dla jego typu operacji stają się *prawami uzupełniającymi* (ang. *auxiliary rights*). Prawa uzupełniające można opisać w uprawnieniach dotyczących reprezentanta danego typu. Aby proces mógł wykonać operację na obiekcie danego typu, jego uprawnienia dotyczące tego obiektu powinny zawierać nazwę wywoływanej operacji wśród praw uzupełniających. To zastrzeżenie pozwala na ograniczanie praw dostępu na zasadzie reprezentant reprezentantowi lub proces procesowi.

Inną interesującą koncepcją jest *wzmacnianie praw* (ang. *rights amplification*). Polega ono na upelnomocnieniu procedury jako „zaufanej” (ang. *trustworthy*), aby mogła działać na parametrze formalnym określonego typu w imieniu dowolnego procesu mającego prawo ją wykonywać. Prawa procedury zaufanej są niezależne od praw wywołującego ją procesu i mogą być od nich silniejsze. Jest jednak niezbędne, aby nie uważać takiej procedury za uniwersalnie zaufaną (nie wolno jej na przykład działać na innych typach) oraz nie rozszerzać zaufania na żadne inne procedury lub programy, które mogą być wykonywane przez proces.

Wzmacnianie praw przydaje się przy umożliwianiu procedurom implementacyjnym dostępu do zmiennych reprezentujących abstrakcyjne typy danych. Jeżeli na przykład proces ma uprawnienia do operowania obiektem typu *A*, to uprawnienia te mogą zawierać uzupełniające prawo wywoływania pewnej operacji *P*, lecz nie mogą zawierać żadnego z tzw. praw jądra, takich jak czytanie, pisanie lub wykonywanie w odniesieniu do segmentu reprezentującego *A*. Tego rodzaju uprawnienia dają procesowi środki dostępu pośredniego (przez operację *P*) do reprezentacji obiektu *A*, lecz tylko w określonych celach.

Z drugiej strony, jeśli proces wywołuje operację *P* na obiekcie *A*, to uprawnienia dostępu do *A* mogą zostać wzmacnione w chwili przekazania sterowania do kodu operacji *P*. Może to być niezbędne, aby udzielić *P* prawa dostępu do segmentu pamięci reprezentującego *A* w celu zrealizowania operacji, którą *P* definiuje na abstrakcyjnym typie danych. Kod *P* może uzyskać prawo bezpośredniego czytania lub pisania segmentu *A*, pomimo że wywołujący go proces nie ma takiego prawa. Przy kończeniu operacji *P* przywraca się obiektemu *A* uprawnienia w pierwotnym, nie wzmacnionym stanie. Jest to typowy przypadek, w którym prawa dostępu procesu do chronionego segmentu muszą zmieniać się dynamicznie w zależności od pracy, którą należy wykonać. Dynamicznej regulacji praw dokonuje się w celu zagwarantowania spójności abstrakcyjnych typów danych zdefiniowanych przez programistę. Wzmocnienie praw można jawnie określić w deklaracji typu abstrakcyjnego, przedkładanej systemowi Hydra.

Jeśli użytkownik przekazuje obiekt do procedury jako argument, to może być konieczne zapewnienie, że procedura nie zmieni przekazanego jej obiektu. Ograniczenie tego rodzaju można czytelnie wprowadzić przez przekazanie prawa dostępu nie zawierającego prawa zmieniania (pisania). Jeśli jednak może występować wzmacnianie praw, to prawo zmieniania może zostać nadane wtórnie. Zatem można by było obejść regulę ochrony sformułowane przez użytkownika. Rzecz oczywista, użytkownik może na ogół wierzyć, że procedura naprawdę wykona swoje zadanie poprawnie. Jednak to założenie nie zawsze się spełnia z powodu błędów sprzętowych lub w oprogramowaniu.

W systemie Hydra rozwiązano to zagadnienie przez ograniczanie możliwości wzmacniania praw.

Mechanizm wywołania procedury w systemie Hydra opracowano z myślą o bezpośrednim rozwiązyaniu problemu *wzajemnie podejrzewających się podsystemów* (ang. *mutually suspicious subsystems*). Problem ów definiuje się następująco. Założmy, że mamy do czynienia z programem, który może być wywoływany w celach usługowych przez wielu różnych użytkowników (może to być na przykład procedura sortująca, kompilator lub gra). Gdy użytkownicy wywołują ten program usługowy, podejmują wówczas ryzyko, że program, wskutek błędnego działania, uszkodzi jakieś dane lub pozostawi sobie pewne prawa dostępu do tych danych (bez pozwolenia) do późniejszego użytku. Podobnie, program usługowy może mieć pewne prywatne pliki (np. do celów rozliczeniowych), które nie powinny być dostępne bezpośrednio dla wywołującego go użytkownika. W systemie Hydra istnieją mechanizmy służące do bezpośredniego radzenia sobie z tym problemem.

Podsystem Hydra jest zbudowany powyżej swojego jądra ochrony; jego własne składowe mogą wymagać ochrony. Podsystem współdziała z jądrem za pomocą wywoływanego zdefiniowanych w jądrze elementarnych działań, określających prawa dostępu do zasobów zdefiniowanych przez podsystem. Sposoby korzystania z tych zasobów przez procesy użytkowe mogą zostać określone przez projektanta podsystemu, są one jednak egzekwowane za pomocą standardej ochrony dostępów, osiąganej dzięki systemowi uprawnien.

Programista może bezpośrednio korzystać z systemu ochrony po zaznajomieniu się z jego własnościami w odpowiednim podręczniku. Hydra ma obszerną bibliotekę procedur systemowych, które można wywoływać w programach użytkowych. Użytkownik systemu Hydra ma możliwość jawnego dołączania wywołań tych procedur systemowych do kodu swoich programów lub posługiwania się translatorem programu, który tworzy interfejs z systemem Hydra.

19.6.2 System Cambridge CAP

Odmienne podejście do ochrony opartej na uprawnieniach zastosowano w projekcie systemu Cambridge CAP. System uprawnień CAP jest prostszy i na pierwszy rzut oka o wiele mniej uniwersalny od systemu Hydra. Jednak bliższa analiza wykazuje, że również on może posłużyć do bezpiecznej ochrony obiektów zdefiniowanych przez użytkownika. W systemie CAP są dwa rodzaje uprawnień. Zwykły rodzaj uprawnień odnosi się do danych (ang. *data capability*). Można ich używać do udostępniania obiektów, lecz jedynymi realizowanymi prawami są standardowe czytanie, pisanie lub wykonywanie poszczególnych segmentów pamięci związanych z obiektem. Uprawnienia do danych są interpretowane przez mikroprogram maszyny CAP.

Tak zwane uprawnienia programowe (ang. *software capability*) są chronione przez mikroprogram CAP, ale nie interpretowane. Interpretuje je chroniona (czyli uprzywilejowana) procedura, którą może napisać programista aplikacji jako część jakiegoś podsystemu. Z procedurą chronioną jest związany szczególny rodzaj wzmacniania praw. Podczas wykonywania kodu takiej procedury proces uzyskuje czasowo prawa czytania lub pisania treści samych uprawnień programowych. Ów specyficzny rodzaj wzmacniania praw ma odzwierciedlenie w implementacji elementarnych operacji zapieczętowania i odpieczętowania (ang. *seal* i *unseal*), wykonywanych na uprawnieniach (zob. uwagi bibliograficzne). Rzeczą jasna, przywilej taki jest ciągle poddawany weryfikacji typu w celu zapewnienia, że procedurze chronionej będą przekazywane tylko uprawnienia programowe odpowiednie dla określonego typu abstrakcyjnego. Pełnym zaufaniem nie obdarza się żadnego fragmentu kodu oprócz mikroprogramu maszyny CAP.

Interpretacja uprawnień programowych jest w całości zależna od podsystemu i zawartych w nim procedur chronionych. Schemat taki umożliwia realizowanie różnorodnej polityki ochrony. Wprawdzie programista może zdefiniować własne chronione procedury (z których każda może być nieuprawniona), lecz mimo to bezpieczeństwo całego systemu nie zostanie naruszone. Podstawowy system ochrony nie zezwoli na nie sprawdzony, określony przez użytkownika, dostęp chronionej procedury do segmentów pamięci (lub uprawnień), które nie należą do środowiska ochrony, w którym ona rezyduje. Najpoważniejszą konsekwencją wykonania nie zabezpieczonej procedury chronionej jest naruszenie ochrony podsystemu, za który dana procedura ponosiła odpowiedzialność.

Projektanci systemu CAP spostrzegli, że użycie uprawnień programowych pozwoliło im istotnie zoptymalizować formułowanie i implementowanie polityki ochrony według wymagań abstrakcyjnych zasobów. Jednak projektant podsystemu chcący wykorzystać te właściwości nie może ograniczyć się po prostu do przestudiowania podręcznika, jak w przypadku systemu Hydra. Musi jeszcze wyuczyć się zasad i sposobów ochrony, gdyż system nie udostępnia żadnej biblioteki procedur gotowych do użycia.

19.7 ■ Ochrona na poziomie języka programowania

Ochrona w istniejących systemach komputerowych jest zazwyczaj organizowana na poziomie jądra systemu operacyjnego, które działa niczym agent bezpieczeństwa kontrolujący i legalizujący każdy zamiar dostępu do chronionego zasobu. Ponieważ wszechstronna legalizacja dostępów jest potencjalnie źródłem znacznych nakładów, więc w celu zmniejszenia kosztu uprawomoc-

nania należy wykonywać je sprzętowo lub wypadnie pogodzić się z tym, iż projektant systemu będzie zmuszony nagiąć się do kompromisowego potraktowania zadań ochrony. Spełnienie tych wszystkich celów może okazać się trudne, jeśli elastyczność w realizowaniu różnorodnej polityki ochrony będzie ograniczona przez zastosowane mechanizmy lub jeśli objęte ochroną środowiskowa będą obszerniejsze, niż byłoby to niezbędne do zagwarantowania większej efektywności operacyjnej.

W miarę wzrostu złożoności systemów operacyjnych, a zwłaszcza wskutek zarysowującej się w nich tendencji do dostarczania wysokiego poziomu interfejsów z użytkownikiem, cele ochrony stały się bardziej złożone. W owym skomplikowaniu można dostrzec, że projektanci systemów ochrony ulegają wpływom idei wywodzących się od języków programowania, a szczególnie koncepcji abstrakcyjnych typów danych i obiektów. Systemy ochrony rozwija się obecnie nie tylko pod kątem identyfikacji zasobów i prób dostępu do nich, lecz również z uwagi na funkcjonalną naturę danego dostępu. W najnowszych systemach ochrony kwestią doboru funkcji, którą należy się posłużyć, wykracza poza zbiór działań zdefiniowanych w systemie, takich jak standardowe metody dostępu do pliku, obejmując również funkcje definiowane przez użytkownika.

Sposoby postępowania z zasobami mogą również ulegać zmianom w zależności od zastosowania i upływu czasu. Z tych przyczyn nie można już traktować ochrony jako sprawy, którą zajmuje się wyłącznie projektant systemu operacyjnego. Powinna ona być narzędziem dostępnym również dla projektanta aplikacji, aby zasoby podsystemu użytkowego mogły być strzeżone przed nadużyciem lub skutkami błędu.

W tym miejscu na arenę wkraczają języki programowania. Określenie pożąданiej kontroli dostępu do dzielonego zasobu w systemie sprowadza się do napisania odpowiedniej deklaracji w odniesieniu do danego zasobu. Taką deklarację można dodać do języka przez rozszerzenie jego możliwości operowania typami. Deklarując ochronę przy okazji określania typów danych, projektant każdego podsystemu może określić własne wymagania w stosunku do ochrony, a także swoje potrzeby odnośnie do innych zasobów systemu. Specyfikację taką powinno się podawać bezpośrednio podczas tworzenia programu, w tym samym języku, w którym wyraża się dany program. Podejście takie ma kilka istotnych zalet:

1. Zapotrzebowania na ochronę są po prostu deklarowane, a nie programowane w formie ciągu wywołań procedur systemu operacyjnego.
2. Wymagania dotyczące ochrony można sformułować niezależnie od środków dostarczanych przez konkretny system operacyjny.

3. Projektant podsystemu nie musi dostarczać środków wymuszania ochrony.
4. Notacja deklaratywna jest naturalna, ponieważ przywileje dostępów pozostały w ścisłym związku z lingwistyczną koncepcją typu danych.

Istnieje wiele sposobów, które w implementacji języka programowania można udostępnić w celu wymuszenia ochrony, lecz każdy z nich musi w pewnym stopniu zależeć od środków dostarczanych przez zastosowany sprzęt i system operacyjny. Założymy na przykład, że do generowania kodu mającego pracować w systemie Cambridge CAP zastosowano jakiś język programowania. W systemie tym każde sprzętowe odniesienie do pamięci odbywa się pośrednio, za pośrednictwem uprawnień. Ograniczenie takie zapobiega w każdej chwili wszelkim próbom dostępu procesu do zasobu spoza jego strefy ochronnej. Jednak program może wymusić dowolne ograniczenia na sposób użytkowania zasobu podczas wykonania konkretnego segmentu kodu przez dowolny proces. Ograniczenia takie można zrealizować najczytelniej, korzystając z uprawnień programowych dostarczanych przez system CAP. Implementacja języka może udostępniać standardowe, chronione procedury do interpretacji uprawnień programowych, służących urzeczywistnianiu takich zasad ochrony, jakie da się wyrazić w danym języku. Schemat taki pozwala określanie zasad ochrony decyzjom osób programujących, a jednocześnie uwalnia je od szczegółów związanych z egzekwowaniem tych zasad.

Nawet jeśli system nie dostarcza tak silnego jądra ochrony jak systemy Hydra lub CAP, zawsze zawiera mechanizmy umożliwiające implementowanie reguł ochrony wyrażonych w języku programowania. Podstawowa różnica polega na tym, że uzyskane w ten sposób bezpieczeństwo nie będzie tak duże jak to, które gwarantuje jądro ochrony, ponieważ taki mechanizm musi przyjmować więcej założeń odnośnie do działania systemu. Kompilator może odróżniać odwołania, co do których nie ma wątpliwości, że nie naruszają ochrony, od tych, które grożą takimi konsekwencjami, i traktować je w różny sposób. Bezpieczeństwo tego rodzaju ochrony spoczywa na założeniu, że kod wygenerowany przez kompilator nie zostanie zmieniony ani przed, ani podczas wykonania.

Jakie są wobec tego względne argumenty na rzecz wymuszania ochrony wyłącznie przez jądro systemu w porównaniu z wymuszaniem jej przede wszystkim przez kompilator?

- **Bezpieczeństwo:** Wymuszanie przez jądro daje wyższy stopień bezpieczeństwa samego systemu ochrony aniżeli generowanego przez kompilator kodu sprawdzającego ochronę. W systemie ochrony tworzonym przez kompilator bezpieczeństwo zależy od poprawności translatora, od pewnych elementarnych mechanizmów zarządzania pamięcią, chroniących

segmenty, w których znajdują się skompilowane rozkazy wykonywanego kodu, wreszcie – od bezpieczeństwa plików, z których skompilowany kod jest ładowany do pamięci. Niektóre z tych aspektów odnoszą się także do jądra ochrony wspieranego programowo, lecz w mniejszym stopniu, jako że jądro może rezydować w ustalonych segmentach pamięci fizycznej i może być ładowane tylko ze ścisłe określonego pliku. W systemie uprawnień znaczonych (ang. *tagged capability system*), w którym wszystkie obliczenia adresów są wykonywane przez sprzęt lub przez stały mikroprogram, można osiągnąć jeszcze większe bezpieczeństwo. Ochrona realizowana sprzętowo jest także względnie odporna na naruszenia wynikające ze złego działania sprzętu lub oprogramowania systemowego.

- **Elastyczność:** Jądro ochrony ogranicza wdrażanie polityki zdefiniowanej przez użytkownika, choć może rozporządzać odpowiednimi środkami do egzekwowania zasad ochrony narzuconych przez system. Z pomocą języka programowania polityka ochrony może podlegać deklarowaniu, a jej egzekwowanie urzeczywistnia implementacja danego języka. Jeśli język nie gwarantuje wystarczającej elastyczności, to można go rozszerzyć lub wymienić na inny; powoduje to mniej zaburzeń w systemie niż modyfikowanie jądra systemu operacyjnego.
- **Wydajność:** Najlepszą wydajność uzyskuje się wówczas, gdy egzekwowanie ochrony odbywa się wprost za pomocą sprzętu (lub mikroprogramu). W zależności jednak od stopnia programowej realizacji ochrony jej egzekwowanie środkami języka programowania ma tę zaletę, że kontrola statycznego dostępu może się odbywać podczas komplikacji, a nie działania programu. Istnieje również możliwość unikania ustawicznych kosztów wywołań jądra związanych z ochroną dzięki dopasowaniu mechanizmów jej wymuszania do konkretnych potrzeb przez inteligentny kompilator.

Podsumowując, możemy powiedzieć, że określanie ochrony za pomocą języka programowania pozwala opisywać na wysokim poziomie zasady przydziału i użytkowania zasobów. Implementacja języka może umożliwiać egzekwowanie ochrony tam, gdzie nie istnieje możliwość kontroli sprzętowej. Ponadto może ona interpretować reguły ochrony, generując wywołania zarówno w przypadku sprzętowej realizacji ochrony, jak i wówczas, gdy za ochronę odpowiada system operacyjny.

Jednym ze sposobów udostępnienia ochrony w programie użytkowym jest posłuszenie się *uprawnieniami programowymi*, które mogą stanowić przedmiot obliczeń. Tkwi w tym pomyśle założenie, że pewne części oprogramowania mogą mieć przywileje tworzenia i sprawdzania takich progra-

mowych uprawnień. Program tworzący uprawnienia będzie mógł wykonywać elementarne operacje opieczętowywania struktur danych, powodujące ich niedostępność dla jakichkolwiek elementów oprogramowania, które nie mają przywileju dostępu do opieczętowanych danych lub ich odpieczętowywania. Nieuprzywilejowane programy mogą kopiować chronioną strukturę danych lub przekazywać jej adres do innych części oprogramowania, lecz nie mogą uzyskać dostępu do jej zawartości. Wprowadzenie takich uprawnień programowych ma na celu wbudowanie mechanizmu ochrony do języka programowania. W tym pomyśle jest kłopotliwe jedynie proceduralne podejście do określania ochrony za pomocą operacji *seal* i *unseal* (pieczętowania i odpieczętowywania). Notacja nieproceduralna lub deklaratywna wydaje się bardziej godna polecenia przy udostępnianiu środków ochrony osobom programującym aplikacje.

To, co jest potrzebne, to bezpieczny, dynamiczny mechanizm rozprowadzania między procesy użytkowników uprawnień do zasobów systemowych. Jeśli ma być osiągnięta ogólna niezawodność systemu, to mechanizm kontroli dostępu musi być bezpieczny w użyciu. Aby mechanizm ów mógł być użyteczny w praktyce, powinien być również stosunkowo wydajny. To wymaganie doprowadziło do opracowania pewnej liczby konstrukcji językowych, umożliwiających programistom deklarowanie rozmaitych ograniczeń, stosownie do specyfiki zarządzania zasobem (odpowiednie odsyłacze można znaleźć w uwagach bibliograficznych). Konstrukcje te dostarczają mechanizmów wykonywania trzech funkcji:

1. Bezpiecznego i wydajnego rozprowadzania uprawnień między procesami konsumenckimi; istnieją w szczególności mechanizmy zapewniające, że proces użytkownika skorzysta z powierzonego ich opiece zasobu tylko wtedy, kiedy uzyska do niego uprawnienia.
2. Określania rodzajów operacji, które dany proces może wykonać na przydzielonym zasobie (np. proces, który ma czytać plik, powinien mieć tylko prawo czytania pliku, a proces piszący do pliku powinien móc zarówno czytać, jak i pisać). Nie powinno być konieczne przyznawanie tego samego zbioru praw każdemu procesowi użytkowemu, jak również nie powinno być możliwe, aby proces powiększył swoje prawa dostępu bez akceptacji mechanizmu kontroli dostępu.
3. Określania porządku, w którym konkretny proces może wywoływać różne operacje na zasobie (np. plik przed czytaniem powinien być otwarty). Powinno być możliwe nadanie dwu procesom różnych ograniczeń odnośnie do porządku, w którym mogą one wywoływać operacje na przydzielonym zasobie.

Wdrażanie zagadnień ochrony do języków programowania jako praktycznego narzędzia projektowania systemów jest obecnie we wczesnym stadium rozwoju. Jest prawdopodobne, że ochrona stanie się obiektem wielkiego zainteresowania projektantów nowych systemów o rozproszonych architekturach i zwiększych wymaganiach na bezpieczeństwo danych. Rola odpowiednich notacji językowych służących wyrażaniu reguł ochrony stanie się wówczas szerzej doceniona.

19.8 ■ Podsumowanie

Systemy komputerowe zawierają wiele obiektów. Obiekty te należy chronić przed niewłaściwym użyciem. Mogą to być obiekty sprzętowe (takie jak pamięć, czas procesora lub urządzenia wejścia-wyjścia) lub programowe (np. pliki, programy i abstrakcyjne typy danych). Prawo dostępu jest pozwoleniem na wykonanie operacji na obiekcie. Domena jest zbiorem praw dostępu. Procesy działają w domenach i mogą używać dowolnego z praw dostępu w danej domenie w celu docierania do obiektów i wykonywania na nich operacji.

Macierz dostępów jest ogólnym modelem ochrony. Macierz dostępów tworzy mechanizm ochrony bez narzucania konkretnej polityki ochrony w stosunku do systemu lub jego użytkowników. Oddzielenie polityki od mechanizmu jest ważną zasadą projektowania systemu.

Macierz dostępów jest rozrzedzona. Na ogół implementuje się ją jako wykazy dostępów powiązane z poszczególnymi obiektami albo jako wykazy uprawnień powiązane z każdą domeną. Do modelu macierzy dostępów można dodać ochronę dynamiczną, rozpatrując domeny i samą macierz dostępów jako obiekty.

Rzeczywiste systemy są znacznie bardziej ograniczone. Dostrzega się w nich tendencję do realizowania ochrony wyłącznie w odniesieniu do plików. Reprezentatywnym przykładem jest system UNIX, w którym w odniesieniu do każdego pliku zapewniono ochronę czytania, pisania i wykonywania – z osobna dla właściciela, grupy i ogółu. W systemie MULTICS zastosowano strukturę pierścieniową w uzupełnieniu metod dostępu do plików. Systemy Hydra, Cambridge CAP oraz Mach działają na zasadzie uprawnień i rozszerzają ochronę na obiekty programowe zdefiniowane przez użytkownika.

■ Ćwiczenia

- 19.1 Jakie są główne różnice między wykazami uprawnień a wykazami dostępów?

- 19.2 Plik w systemie Burroughs B7000/B6000 MCP może zostać oznaczony jako dane wrażliwe. W czasie usuwania takiego pliku zajmowane przez niego miejsce w pamięci zostaje zapisane losowym ciągiem bitów. Do jakiego celu schemat taki może okazać się przydatny?
- 19.3 Poziom 0 w systemie ochrony pierścieniowej ma największe prawa dostępu do obiektów, a poziom n (większe od zera) ma mniejsze prawa dostępu. Prawa dostępu programu na konkretnym poziomie w strukturze pierścieniowej są rozpatrywane jako zbiór uprawnień. Jaka jest zależność między uprawnieniami w domenie na poziomie j oraz w domenie na poziomie i w odniesieniu do jakiegoś obiektu (dla $j > i$)?
- 19.4 Rozważmy system, w którym gry komputerowe mogą być używane przez studentów tylko między godziną 22 a 6 rano, przez pracowników uczelni – między 5 po południu a 8 rano, a przez załogę centrum komputerowego – przez cały czas. Zasugeruj schemat efektywnej realizacji takiej polityki.
- 19.5 System RC 4000 (i inne systemy) definiował drzewo procesów (tak właśnie nazywane) w ten sposób, że wszyscy potomkowie procesu otrzymywali zasoby (obiekty) i prawa dostępu wyłącznie po swoich przodkach. Tak więc potomek nigdy nie mógł robić czegoś, czego nie mógł robić jego przodek. Korzeniem drzewa był system operacyjny, który mógł robić wszystko. Założmy, że zbiór praw był reprezentowany przez macierz dostępów A . Element $A(x, y)$ tej macierzy określa prawa procesu x do obiektu y . Jaki jest związek między elementami $A(x, y)$ i $A(z, y)$ dla dowolnego obiektu y , jeśli x jest potomkiem z ?
- 19.6 Jakie cechy sprzętowe są potrzebne do wydajnego manipulowania uprawnieniami? Czy można ich użyć do ochrony pamięci?
- 19.7 Rozważmy środowisko, w którym każdemu procesowi i obiektowi w systemie jest przyporządkowana niepowtarzalna liczba. Założmy, że dostęp procesu n do obiektu m jest dozwolony tylko wtedy, gdy $n > m$. Jaki typ struktury ochrony otrzymujemy?
- 19.8 Jakie kłopoty z ochroną mogą powstać przy użyciu stosu dzielnego do przekazywania parametrów?
- 19.9 Rozważmy środowisko komputerowe, w którym proces ma przywilej tylko n -krotnego dostępu do obiektu. Zaproponuj schemat realizacji tej polityki.
- 19.10 Jeśli wszystkie prawa dostępu do obiektu zostaną usunięte, to dalszy dostęp do obiektu jest niemożliwy. W tej sytuacji obiekt również po-

winno się usunąć, a zajmowaną przez niego przestrzeń zwrócić do systemu. Zaproponuj efektywną implementację tego schematu.

19.11 Co oznacza zasada wiedzy koniecznej? Dlaczego jest ważne, aby w systemie ochrony przestrzegano tej zasady?

19.12 Dlaczego jest ciężko chronić system, w którym użytkownicy mogą sami wykonywać operacje wejścia-wyjścia?

19.13 Wykazy uprawnień są zwykle przechowywane w przestrzeni adresowej użytkownika. W jaki sposób system zapewnia, że użytkownik nie zmieni zawartości tych wykazów?

Uwagi bibliograficzne

Domenowy model ochrony obiektów za pomocą macierzy dostępów opracował Lampson [235 i 236]. Popek [332] oraz Saltzer i Schroeder [366] dokonali znakomitego przeglądu zagadnień ochrony. Harrison i in. [165] użyli formalnej wersji tego modelu do matematycznego dowodu właściwości systemu ochrony.

Koncepcja uprawnień wywodzi się od *słów kodowych* (ang. *codewords*) zaimplementowanych przez Iliffe'a i Jodeita [188] na komputerze w Rice University. Termin „uprawnienia” (ang. *capabilities*) wprowadzili do obiegu Dennis i Van Horn [105].

System Hydra opisali Wulf i in. [445]. System CAP został omówiony przez Needhama i Walkera [307]. Omówienia ochrony pierścieniowej w systemie MULTICS dokonał Organick [317].

Unieważnianie praw omówili Redell i Fabry [346], Cohen i Jefferson [78] oraz Ekanadham i Bernstein [125]. O potrzebie przestrzegania zasady oddzielenia polityki od mechanizmu przekonywali projektanci systemu Hydra (Levin i in. [252]). Problem zamknięcia (ang. *confinement*) został po raz pierwszy poruszony przez Lampsona [237], a potem przebadany przez Lipnera [257].

Zastosowanie języków wysokiego poziomu do określania reguł nadzorowania dostępu zaproponował po raz pierwszy Morris w artykule [296]; sugerował on użycie operacji *seal* i *unseal*, omówionych w p. 19.7. Kieburtz i Silberschatz [211, 212] oraz McGraw i Andrews [274] zaproponowali rzemiące konstrukcje językowe do działań według ogólnych schematów dynamicznego zarządzania zasobami. Jones i Liskov [200] rozważali zagadnienie wdrożenia statycznego schematu sprawdzania dostępu w języku programowania zawierającym możliwość definiowania abstrakcyjnych typów danych.

Rozdział 20

BEZPIECZEŃSTWO

Ochrona – jak to omówiliśmy w rozdz. 19 – jest problemem ścisłe *wewnętrzny*: jak umożliwić kontrolowany dostęp do programów i danych przechowywanych w systemie komputerowym? Natomiast zapewnienie *bezpieczeństwa* (ang. *security*) wymaga, oprócz adekwatnego systemu ochrony, przeanalizowania również środowiska *zewnętrznego*, w którym działa dany system. Ochrona wewnętrzna jest nieskuteczna, jeśli konsola operatora jest wystawiona na działania nieuprawnionego personelu lub jeżeli pliki (przechowywane np. na taśmach i dyskach) mogą być po prostu usunięte z systemu komputerowego i przeniesione do systemu nie mającego żadnej ochrony. Tego rodzaju zagadnienia bezpieczeństwa są w istocie problemami zarządzania, a nie systemu operacyjnego.

Informacje przechowywane w systemie (zarówno dane, jak i kod), a także fizyczne zasoby systemu komputerowego muszą być chronione przed nieupoważnionym dostępem, złośliwymi uszkodzeniami lub zmianami oraz przypadkowymi wystąpieniami niespójności. W tym rozdziale analizujemy sposoby, w jakie może dochodzić do nadużycia informacji lub zamierzonego naruszenia jej spójności. Następnie przedstawiamy mechanizmy, które mają strzec przed takimi zdarzeniami.

20.1 ■ Zagadnienie bezpieczeństwa

W rozdziale 19 omówiliśmy rozmaite mechanizmy, które mogą być dostarczane przez system operacyjny (wraz z odpowiednią pomocą ze strony sprzętu) w celu umożliwiania użytkownikom chronienia ich zasobów (zazwy-

czaj programów i danych). Mechanizmy te działają poprawnie dopóty, dopóki użytkownicy nie próbują obchodzić przewidzianych metod korzystania z zasobów i dostępu do nich. Niestety, taka sytuacja należy do rzadkości. Kiedy ona nie występuje, wtedy pojawia się kwestia bezpieczeństwa. Mówimy, że system jest *bezpieczny* (ang. *secure*), jeżeli dostęp do jego zasobów i sposob ich wykorzystania jest zgodny z założonym w każdych warunkach. Należy z ubolewaniem stwierdzić, że osiągnięcie całkowitego bezpieczeństwa nie jest, ogólnie biorąc, możliwe. Niemniej jednak muszą być dostępne jakieś mechanizmy zapewniające, że naruszenia bezpieczeństwa będą zjawiskami rzadkimi, zamiast stawać się normą.

Naruszenia bezpieczeństwa (nadużycia) systemu można podzielić na rozmyślne (złośliwe) i przypadkowe. Łatwiej jest chronić się przed nadużyciami przypadkowymi niż przed złośliwymi. Do złośliwych form nadużycia należą m.in.:

- czytanie danych bez upoważnienia (kradzież informacji);
- zmienianie danych bez upoważnienia;
- niszczenie danych bez upoważnienia.

Całkowita ochrona systemu przed złośliwymi nadużyciami nie jest możliwa, można jednak uczynić je dla przestępów na tyle kosztownymi, aby powstrzymać większość (jeśli nie wszystkie) usiłowań dostępu do pozostającej w systemie informacji bez właściwego upoważnienia.

Aby chronić system, musimy określić miary bezpieczeństwa na dwu poziomach:

- **fizycznym** – miejsce lub miejsca rozlokowania systemów komputerowych muszą być zabezpieczone fizycznie przed zbrojnym lub potajemnym wtargnięciem intruzów;
- **ludzkim** – użytkownicy muszą być starannie prześwietlani w celu minimalizowania szans upoważnienia użytkownika, który może potem umożliwić dostęp intruzowi (np. w zamian za łapówkę).

W celu zapewnienia bezpieczeństwa systemu operacyjnego należy przestrzegać bezpieczeństwa na obu tych poziomach. Słabe bezpieczeństwo na wysokim poziomie (fizycznym lub ludzkim) umożliwia obchodzenie ścisłych zasad bezpieczeństwa na niskim poziomie (systemu operacyjnego).

W wielu zastosowaniach warto włożyć sporo wysiłku w bezpieczeństwo systemu komputerowego. Wielkie systemy komercyjne zawierające listy płac lub inne dane finansowe są atrakcyjnymi celami dla złodziei. Systemy

zawierające dane o działalności przedsiębiorstw mogą być obiektami zainteresowania pozbawionej skrupułów konkurencji. Należy zdawać sobie sprawę z tego, że utrata tego rodzaju danych, obojętnie – przypadkowa czy wskutek oszustwa, może poważnie osłabić zdolność przedsiębiorstwa do działania.

Z drugiej strony jest konieczne, aby sprzęt systemu realizował ochronę (na zasadach omówionych w rozdz. 19) w celu umożliwienia implementacji reguł bezpieczeństwa. Na przykład w systemach MS-DOS i Macintosh OS poziom bezpieczeństwa jest niski, ponieważ sprzęt, dla którego systemy te były pierwotnie projektowane, nie umożliwiał ochrony pamięci ani ochrony urządzeń wejścia-wyjścia. Dzisiaj, kiedy sprzęt komputerowy jest wystarczająco rozwinięty, żeby umożliwić ochronę, projektanci tych systemów toczą boje o zaopatrzenie ich w środki bezpieczeństwa. Niestety, dodawanie właściwości do działającego systemu jest znacznie bardziej trudnym zadaniem, niż powiedzieć – wyzwaniem, aniżeli zaprojektowanie i zrealizowanie odpowiednich cech, zanim system zostanie zbudowany. Późniejsze systemy operacyjne, takie jak Windows NT, skonstruowano w sposób umożliwiający realizację środków bezpieczeństwa od podstaw.

W pozostałej części tego rozdziału zajmiemy się bezpieczeństwem na poziomie systemu operacyjnego. Bezpieczeństwo na poziomie fizycznym i ludzkim – choć ważne – wykracza znacznie poza zakres tej książki. Bezpieczeństwo w systemie operacyjnym realizuje się na kilku poziomach, począwszy od haseł niezbędnych do dostępu do systemu aż po izolowanie procesów wykonywanych współbieżnie w systemie. W systemie plików również występuje pewien stopień ochrony.

20.2 ■ Uwierzytelnianie

Podstawowym problemem bezpieczeństwa w systemach operacyjnych jest problem *uwierzytelniania** (ang. *authentication*). System ochrony zależy od zdolności identyfikowania działających programów i procesów. Z kolei ta zdolność zależy od naszych możliwości identyfikacji każdego użytkownika systemu. Użytkownik zazwyczaj przedstawia się sam. Jak rozstrzygnąć, czy tożsamość użytkownika jest prawdziwa? W celu sprawdzania tożsamości z reguły wykorzystuje się jedną (lub więcej) z następujących jednostek danych: stan posiadania użytkownika (klucz lub karta), wiedzę użytkownika (nazwa użytkownika i hasło) oraz atrybut użytkownika (odcisk palca, wzorzec siatkówki oka lub podpis).

* Czyli sprawdzania tożsamości. – Przyp.tłum.

20.2.1 Hasła

Najpopularniejszą metodą sprawdzania tożsamości użytkowników jest operowanie hasłami użytkowników. Gdy użytkownik się przedstawia, wówczas jest proszony o *hasło* (ang. *password*). Jeśli podane przez użytkownika hasło jest zgodne z hasłem zapamiętanym w systemie, to system zakłada, że ma do czynienia z pełnoprawnym użytkownikiem.

W razie braku bardziej złożonych schematów ochrony hasła są często używane do ochrony obiektów w systemie komputerowym. Można je uważać za specjalny przypadek kluczy lub uprawnień. Hasło można na przykład kojarzyć z każdym zasobem (takim jak plik). Każdorazowe zamówienie na użycie tego zasobu musi być opatrzone hasłem. Jeśli hasło jest poprawne, to zezwala się na dostęp. Z różnymi prawami dostępu mogą być skojarzone różne hasła. Na przykład przy każdym czytaniu, dopisywaniu lub aktualnianiu pliku mogą być stosowane różne hasła.

20.2.2 Słabości hasłów

Chociaż stosowanie hasłów jest kłopotliwe, hasła są jednak niezwykle rozpoznawcze, gdyż łatwo je zrozumieć i używać. Kłopoty z hasłami dotyczą trudności w utrzymaniu ich w sekrecie. Hasła mogą być lamane przez odgadnięcie, przypadkowe ujawnienie lub nielegalne przekazanie stronie nieuprawnionej przez użytkownika uprawnionego, co pokażemy dalej.

Istnieją dwa popularne sposoby odgadywania hasła. Jeden polega na tym, że intruz (człowiek lub program) zna użytkownika lub posiada o nim pewne informacje. Nazbyt często ludzie używają oczywistych informacji (takich jak imiona ich kotów lub współmałżonków) w charakterze hasłów. Drugi sposób polega na użyciu metody siłowej (ang. *brute force*), czyli wypróbowaniu wszystkich możliwych kombinacji liter, cyfr i znaków przestankowych – aż do znalezienia hasła. Krótkie hasła nie dają dostatecznie dużych możliwości wyboru, zapobiegającego odgadnięciu hasła w wyniku wielokrotnych prób. Na przykład dla hasła złożonego z czterech cyfr dziesiętnych istnieje tylko 10 000 kombinacji. Zatem średnio trzeba wykonac tylko 5000 prób, aby je odgadnąć. Jesliby napisać program sprawdzający kolejne hasło w ciągu 1 ms, to odgadnięcie czterocyfrowego hasła zajęłoby około 5 s. Dłuższe hasła są mniej podatne na odgadnięcie przez wyliczenie, a systemy, w których rozróżnia się litery małe i wielkie oraz dopuszcza występowanie w hasłach cyfr i wszystkich znaków interpunkcyjnych, znacznie utrudniają zadanie odgadnięcia hasła. Rzeczą oczywistą użytkownicy powinni korzystać z zalet większej przestrzeni hasłów i nie powinni stosować na przykład tylko małych liter.

Do ujawnienia hasła może dojść wskutek podglądzania – zwykłego lub elektronicznego. Intruz może patrzeć zza pleców użytkownika („zaglądać przez ramię”) i w chwili gdy użytkownik się rejestruje, przechwycić hasło bez trudu na podstawie obserwacji klawiatury. Z kolei każdy, kto ma dostęp do sieci, w której znajduje się dany komputer, może niepostrzeżenie zamontować w niej monitor sieci i śledzić wszystkie dane przekazywane przez sieć („węszenie”), w tym identyfikatory użytkowników i hasła. Szczególnie poważne jest niebezpieczeństwo ujawnienia, jeśli hasło jest zanotowane, ponieważ może ono być przeczytane lub zgubione. Jak się przekonamy, niektóre systemy zmuszają użytkowników do wybierania haseł trudnych do zapamiętania lub długich. W skrajnym przypadku takie wymaganie może doprowadzić do tego, że użytkownik zapisze hasło, pogarszając bezpieczeństwo znacznie bardziej niż w przypadku systemu, który pozwala na stosowanie łatwych haseł.

Ostatnia metoda ujawniania haseł wynika z właściwości natury ludzkiej. W większości instalacji komputerowych obowiązuje zasada, że użytkownicy nie mają wspólnych kont. Zasadę tę wprowadza się czasem ze względów personalnych, lecz często ma ona służyć poprawianiu bezpieczeństwa. Gdyby na przykład identyfikator pewnego użytkownika był dzielony przez kilku użytkowników i doszło do naruszenia bezpieczeństwa konta o tym identyfikatorze, to nie można by było poznac, kto posługiwał się w danej chwili identyfikatorem tego użytkownika ani sprawdzić, czy wizytująca je osoba miała uprawnienia. Powiązanie z jednym identyfikatorem tylko jednego użytkownika umożliwia bezpośredni wgląd w sposób korzystania z jego konta. Niektóre użytkownicy lamią zasadę rozdzielcości kont, aby pomóc znajomym lub ominąć rozliczanie kont. Takie praktyki mogą spowodować przenikanie do systemu osób nieupoważnionych, być może szkodliwych.

Hasła mogą być generowane przez system lub wybierane przez użytkownika. Hasła wygenerowane przez system mogą być trudniejsze do zapamiętania, więc dość powszechną praktyką może okazać się ich zapisywanie przez użytkowników. Z kolei hasła wybierane przez użytkowników są często łatwe do odgadnięcia (np. imię użytkownika lub ulubiona marka samochodu). W niektórych instalacjach administratorzy okresowo sprawdzają haseła i powiadamiają użytkowników, jeśli ich haseła są za krótkie lub łatwe do odgadnięcia. Pewne systemy stosują również *postarzanie* (ang. *aging*) haseł, zmuszając użytkowników do ich zmieniania w regularnych okresach (np. co trzy miesiące). Metoda ta nie jest jednak skuteczna, gdyż użytkownik może łatwo posługiwać się dwoma hasłami naprzemiennie. Jako rozwiązanie tej trudności w niektórych systemach stosuje się zapisywanie używanych haseł każdego użytkownika. System może na przykład pamiętać N ostatnich haseł i nie pozwolić na ich ponowne użycie.

Można stosować kilka odmian tych prostych schematów ustalania hasłów. Na przykład można często zmieniać hasło. W skrajnym przypadku hasło zmienia się podczas każdej sesji. Na końcu każdej sesji jest wybierane (przez system lub użytkownika) nowe hasło, którego należy użyć przy następnej sesji. Zauważmy, że nawet w przypadku nadużycia hasła, może to zdarzyć się tylko jeden raz, a jego nieuprawnione użycie uniemożliwi pracę prawomocnemu użytkownikowi. W rezultacie prawomocny użytkownik odkryje naruszenie bezpieczeństwa przy następnej sesji, kiedy posłuży się już nieważnym hasłem. Można więc będzie przystąpić do naprawiania złamanej zabezpieczenia.

20.2.3 Hasła szyfrowane

Wadą wszystkich tych metod są kłopoty z przechowywaniem hasłów w sekrécie. W systemie UNIX stosuje się szyfrowanie w celu uniknięcia konieczności utrzymywania w tajemnicy spisu hasłów. Każdy użytkownik ma hasło. System zawiera funkcję, której wartość łatwo obliczyć, lecz niezwykle trudno jest znaleźć dla niej funkcję odwrotną (projektanci mają nadzieję, że jest to wręcz niemożliwe). Oznacza to, że mając daną wartość x , łatwo jest obliczyć wartość funkcji $f(x)$. Natomiast dla danej wartości $f(x)$ wyliczenie x jest niemożliwe*. Funkcja ta służy do kodowania wszystkich hasłów. Przechowuje się tylko hasła zakodowane. Gdy użytkownik podaje hasło, wówczas zostaje ono zakodowane i porównane z zapamiętanym, zakodowanym hasłem. Nawet jeśli przechowywane hasło zakodowane zostanie podpatrzone, to nie można go zdekodować, a więc nie można ustalić hasła. Zatem plik z hasłami nie musi być przechowywany w tajemnicy. Funkcja $f(x)$ jest zazwyczaj algorytmem szyfrowania (ang. *encryption*), który został bardzo starannie zaprojektowany i przetestowany; omawiamy to w p. 20.7.

Slabością tej metody jest brak kontroli systemu nad hasłami. Pomimo że hasła są szyfrowane, każdy, kto ma kopię pliku z hasłami, może zastosować do niego szybkie procedury szyfrowania, szyfrując na przykład każde słowo ze słownika i porównując wyniki z hasłami. Jeśli użytkownik wybrał hasło będące słowem ze słownika, to dojdzie do złamania hasła. Na wystarczająco szybkich komputerach lub nawet z pomocą grup wolniejszych komputerów takie porównanie może trwać zaledwie kilka godzin. Ponieważ w systemach uniksowych stosuje się dobrze znane algorytmy szyfrowania, haker mógłby przechowywać podręcznie pary hasło-szyfr w celu szybkiego odnajdywania hasłów uprzednio złamanych. Z tego powodu w nowych wersjach systemu UNIX wpisy hasłów są ukryte.

* W literaturze systemów operacyjnych funkcje takie nazywa się również jednokierunkowymi (ang. *one-way functions*). – Przyp. tłum.

Inną wadą hasłów używanych w systemach uniksowych jest to, że wiele z nich analizuje tylko pierwszych osiem znaków w hasle, więc właściwe wykorzystanie dostępnej przestrzeni hasel staje się dla użytkowników sprawą niezwykle ważną. Aby uniknąć metody łamania hasłów za pomocą szyfrowania słownika, w niektórych systemach nie pozwala się na użycie jako hasłów słów ze słownika. Dobry sposób polega na utworzeniu hasła z pierwszych liter wyrazów łatwej do zapamiętania frazy, z użyciem zarówno liter małych, jak i wielkich i z dodaniem pewnej liczby znaków przestankowych. Na przykład z frazy „Mojej mamie jest na imię Katarzyna” można otrzymać hasło „MmijnIK!”. Hasło takie jest trudne do złamania, ale konkretny użytkownik może je łatwo zapamiętać.

20.3 ■ Hasła jednorazowe

Aby uniknąć kłopotów z wyważchiwaniem hasłów i zagładaniem przez ramię, można zastosować w systemie zbiór hasłów dobranych parami (ang. *paired passwords*). Na początku sesji system wybiera losowo i przedstawia jedną część pary hasłów; użytkownik musi podać jej drugą część. W takim systemie użytkownik jest wzywany do odpowiedzi i musi udzielić jej poprawnie.

Podejście to można uogólnić, używając algorytmu jako hasła. Algorytm może być na przykład funkcją całkowitoliczbową. System wybiera liczbę losową i przedstawia ją użytkownikowi. Użytkownik, traktując tę liczbę jako argument, oblicza wartość funkcji i przekazuje w odpowiedzi poprawny wynik. System również stosuje tę funkcję. Jeśli oba wyniki są takie same, to dostęp zostaje udzielony.

Przymijmy, że istniałaby metoda operowania hasłem, której nie dałoby się zarzucić, iż grozi ujawnieniem hasła. Użytkownik wpisywałby na przykład hasło, a dowolna jednostka, która by je przechwyciła i próbowała użyć ponownie, musiałaby ponieść porażkę. System taki istnieje naprawdę; korzysta się w nim z hasłów algorytmicznych. W tej odmianie system i użytkownik dzielą się tajemnicą. Tajemnicy nigdy nie przesyła się środkami komunikacji, które grożą jej ujawnieniem. Zamiast tego tajnej informacji używa się jako argumentu funkcji wraz ze wspólnym ziarnem (ang. *seed*). Ziarnem nazywa się liczbę losową lub ciąg alfanumeryczny. Ziarno jest wezwaniem do uwierzytelnienia pochodząącym od komputera. Tajemnica oraz ziarno są używane jako argumenty funkcji *ftajemnica, ziarno*). Wynik tej funkcji przesyła się jako hasło do komputera. Ponieważ docelowy komputer również zna tajemnicę i ziarno, może wykonać te same obliczenia. Jeśli wyniki są zgodne, to użytkownik zostaje uwierzytelniony. Kiedy następnym razem użytkownik będzie wymagał uwierzytelnienia, nastąpi wygenerowanie innego ziarna i wykonanie tych samych kroków. Tym razem hasło będzie już inne.

W tego rodzaju systemie *hasel jednorazowych* (ang. *one-time password*) hasło jest inne za każdym razem. Ktakolwiek przechwyciłby takie hasło podczas jednej sesji i spróbowałby użyć go w czasie innej sesji, skazany byłby na niepowodzenie. Hasła jednorazowe są niemalże jedynym sposobem zapobiegania niewłaściwym uwierzytelnieniom, powodowanym ujawnianiem hasł. Istnieje wiele systemów hasel jednorazowych. W implementacjach komercyjnych, takich jak system SecurID, zastosowano kalkulatory sprzętowe. Większość tych urządzeń przypomina kształtem czytniki kart kredytowych, lecz ma klawiaturę i ekran. W niektórych jako losowych zień używa się bieżącego czasu. Użytkownik za pomocą klawiatury wprowadza tajną, dzieloną informację, nazywaną także *osobistym numerem identyfikacyjnym* (ang. *personal identification number* – PIN). Na ekranie pojawia się wówczas jednorazowe hasło.

Inna odmiana hasel jednorazowych polega na użyciu *książki kodowej* (ang. *code book*), czyli *bloczku jednorazowego* (ang. *one-time pad*), będącego listą hasel jednorazowego użytku. W tej metodzie każde kolejne hasło z listy jest używane jeden raz, po czym skreśla się je i usuwa. W popularnym systemie S/Key^{*} jako źródło hasel jednorazowych stosuje się kalkulator programowy albo książkę kodową zawierającą wyniki takich obliczeń.

20.4 ■ Zagrożenia programowe

W środowisku, w którym program napisany przez jednego użytkownika może być używany przez innego użytkownika, istnieje możliwość popełniania nadużyć mogących powodować nieoczekiwane skutki. W punktach 20.4.1 i 20.4.2 opisujemy dwie typowe metody stosowane w tego rodzaju sytuacji. Są to: konie trojańskie i boczne wejścia.

20.4.1 Koń trojański

W wielu systemach istnieje mechanizm pozwalający użytkownikom wykonywać programy napisane przez innych użytkowników. Jeśli te programy są wykonywane w domenie praw dostępu należącej do żądającego ich wykonania użytkownika, to mogą nadużyć tych praw. Na przykład wewnątrz edytora tekstu może znajdować się kod, który służy do wyszukiwania w redagowanym pliku pewnych słów kluczowych. W razie wykrycia któregokolwiek z takich słów cały plik może być skopiowany do specjalnego obszaru dostęp-

^{*} Zob. np. artykuł N. M. Hallera pt. *The S/Key (TM) One-Time Password System*, dostępny w sieci Internet (<ftp://ftp.bellcore.com/pub/nmh/docs/ISOC/smp.ps>) — Przyp. tłum.

nego dla twórcy edytora tekstu. Segment kodu nadużywający swojego środowiska jest nazywany *koniem trojańskim* (ang. *trojan horse*). Zagrożenie koniem trojańskim pogarszają długie ścieżki wyszukiwania (typowe w systemach uniksowych). Ścieżki wyszukiwania zawierają wykazy katalogów, które należy przeszukać, jeśli zostanie napotkana niejasna nazwa programu. W tych katalogach poszukuje się pliku o danej nazwie, po czym następuje jego wykonanie. Wszystkie katalogi na ścieżce wyszukiwania muszą być zabezpieczone, w przeciwnym razie koń trojański mógłby się włiznąć na ścieżkę użytkownika i zostać przypadkowo wykonany.

Rozważmy na przykład używanie znaku „..” w ścieżce wyszukiwania. Znak „..” sygnalizuje powłoce systemu, że należy uwzględnić w poszukiwaniach katalog bieżący. Jeśli więc użytkownik ma „..” w swojej ścieżce wyszukiwania oraz wybierze jako bieżący katalog znajomego i wpisze nazwę zwykłego polecenia systemowego, to zamiast tego polecenia może zostać wykonany program z katalogu tego znajomego*. Program taki działałby w domenie naszego użytkownika, co pozwoliłoby mu wykonać wszystko to, co wolno wykonywać danemu użytkownikowi, łącznie z – dajmy na to – usuwaniem plików tego użytkownika.

Odmianą konia trojańskiego mógłby być program, który naśladuje program rejestracyjny (ang. *login*). Niczego nie podejrzewający użytkownik rozpoczyna rejestrację na terminalu i odnosi wrażenie, że widocznie wpisał błędne hasło. Za drugim razem wszystko jest już dobrze. W rzeczywistości jego klucz identyfikacyjny oraz hasło zostały ukradzione przez symulator programu rejestracyjnego, który złodziej pozostawił aktywny na terminalu. Symulator przechował hasło w innym miejscu, wydrukował sygnał o błędzie rejestracji i zakończył działanie. Dopiero wtedy użytkownik otrzymał zaproszenie do pisania od prawdziwego programu rejestracyjnego. Ten rodzaj ataku można odeprzeć, jeśli system operacyjny będzie drukować instrukcję użycia (terminalu) na końcu sesji interakcyjnej lub za pomocą nieprzechwytywalnych operacji klawiszowych, takich jak **Control-Alt-Delete** w systemie Windows NT.

20.4.2 Boczne wejście

Projektant programu lub systemu może zostawić w oprogramowaniu lukę, którą tylko on potrafi wykorzystać. Ten rodzaj naruszenia bezpieczeństwa pokazano w filmie „Gry wojenne”. Może to być na przykład kod sprawdzający specyficzną nazwę użytkownika lub hasło i obchodzący normalne procedury bezpieczeństwa. Znane są przypadki aresztowania programistów za sprze-

* Aby do tego doszło, kropka na ścieżce musi znajdować się przed nazwami katalogów systemowych. – Przyp. tłum.

niewierzenia bankowe polegające na manipulowaniu zaokrąglaniem kwot w programach i przelewaniu od czasu do czasu po pół centa na własne konto. Takie przelewy mogły urastać do wielkich sum pieniędzy, zważywszy liczbę transakcji wykonywanych przez wielki bank.

Bardzo sprytne boczne wejście można zainstalować w kompilatorze. Kompilator może generować standardowy kod wynikowy oraz boczne wejście – niezależnie od kodu źródłowego, który kompiluje. Jest to działalność szczególnie perfidna, gdyż przeglądanie programu źródłowego nie uwiadomi żadnych niedociągnięć. Informację zawiera tylko kod źródłowy kompilatora. Boczne wejścia przysparzają dużych problemów, ponieważ ich wykrycie wymaga przeanalizowania kodu źródłowego wszystkich części systemu. Zważywszy że systemy oprogramowania mogą liczyć miliony wierszy kodu, analizy takiej nie wykonuje się często.

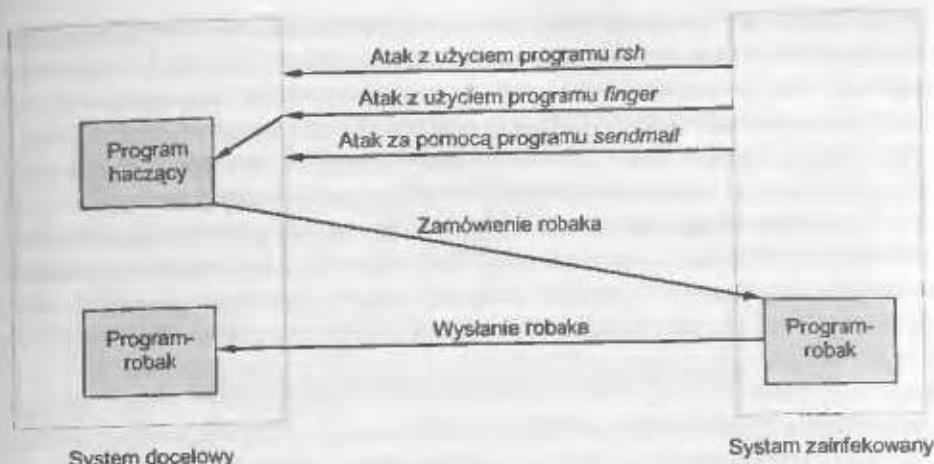
20.5 ■ Zagrożenia systemowe

Większość systemów operacyjnych pozwala na rozmnażanie procesów. W takim środowisku istnieje możliwość nadużywania zasobów systemu operacyjnego i plików użytkownika. Dwie najpopularniejsze metody umożliwiające takie nadużycia polegają na użyciu programów zwanych robakami i wirusami.

20.5.1 Robaki

Robakiem (ang. *worm*) nazywa się proces używający mechanizmu rozmnażania do paraliżowania działania systemu. Robak rodzi własne kopie, zużywając zasoby systemowe i na ogół blokuje innym procesom możliwości korzystania z systemu. Robaki wykazują szczególną vitalność w sieciach komputerowych, ponieważ mogą się reprodukować między systemami, doprowadzając do awarii całej sieci. Zdarzyło się to w 1988 r. w systemach uniksowych światowej sieci Internet i spowodowało straty czasu pracy systemów i programistów warte miliony dolarów.

Sieć Internet łączy w skali międzynarodowej tysiące komputerów instytucji rządowych, akademickich, badawczych, a także zamontowanych w przedsiębiorstwach, i służy jako infrastruktura do elektronicznej wymiany informacji naukowej. Pod koniec roboczego dnia 2 listopada 1988 r. Robert Tappan Morris, Jr., student pierwszego roku studiów w Cornell University utracił kontrolę nad programem-robakiem na co najmniej jednym z komputerów podłączonych do sieci Internet. Biorąc za cel stacje robocze Sun 3 firmy Sun Microsystems oraz komputery VAX działające pod nadzorem odmian wersji 4 systemu UNIX BSD, robak szybko rozprzestrzenił się na duże odległości. Po pięciu godzin-



Rys. 20.1 Robak internetowy Morrisa

nach od rozpoczęcia jego działalności zasoby systemowe były już pozajmowane do tego stopnia, że spowodowało to zawieszenie pracy zainfekowanych maszyn.

Chociaż samopowielający się program do błyskawicznej reprodukcji i rozprzestrzeniania był dziełem Roberta Morrisa, rozhodzenie się robaka w systemie umożliwiły niektóre właściwości środowiska sieci uniksowych. Jako początkowy obiekt zainfekowania Morris wybrał zapewne komputer, który pozostawiono w sieci Internet otwarty i dostępny dla użytkowników z zewnątrz. Stamąd program-robak począł wykorzystywać luki w procedurach bezpieczeństwa systemu operacyjnego UNIX i ułatwienia w dzieleniu zasobów sieci lokalnej w celu uzyskiwania bezprawnego dostępu do tysięcy innych podłączonych do sieci stanowisk. Dalej przedstawiamy zastosowaną przez Morrisa metodę ataku.

Robak składał się z dwóch części: programu haczącego (ang. *grappling hook*)^{*}, nazywanego też programem wciągającym (ang. *bootstrap*) lub wektorem oraz z programu głównego. Program haczący, czyli plik *II.c* składał się z 99 wierszy kodu w języku C i działał na każdej maszynie, do której uzyskał dostęp. Po zainstalowaniu w atakowanym systemie program haczący łączył się z maszyną, z której pochodził i sprawdzał z niej do „zahaczonego” systemu kopię robaka głównego (rys. 20.1). Program główny rozpoczynał wówczas poszukiwanie następnych maszyn, z którymi nowo zainfekowany system mógłby łatwo nawiązać kontakt. Do tego celu Morris użył sieciowego programu *rsh* systemu UNIX, umożliwiającego łatwe wykonywanie zadań zdalnych. Okre-

* Dosłownie: *bosak*, czyli *osęka* – drzewce ze stalowym zadziorem. – Przyp. tłum.

ślając specjalne pliki z wykazami par (nazwa komputera, nazwa rejestracyjna), użytkownicy mogą omijać wprowadzanie hasła przy każdym kontakcie ze zdalnym kontem, umieszczonym na takim wykazie. Robak przeglądał te pliki specjalne w poszukiwaniu nazw stanowisk, na których wolno się było rejestrować bez podawania hasła. Po uruchomieniu zdalnych powłok systemowych program-robak był znów sprawdzany i rozpoczynał działanie na nowo.

Atak odbywający się przez dostęp zdalny był jedną z trzech metod infekowania wbudowanych do robaka. W dwu pozostałych metodach skorzystano z błędów w programach *finger* i *sendmail* systemu operacyjnego UNIX. Program *finger* działa jak elektroniczna książka telefoniczna. Polecenie

finger nazwa-użytkownika@nazwa-komputera

przekazuje w odpowiedzi nazwisko osoby oraz jej nazwę rejestracyjną wraz z innymi informacjami ewentualnie dostarczonymi przez użytkownika, takimi jak adres biurowy i domowy, numer telefonu, przedmiot zainteresowań dawczych lub błyskotliwy cytat. Proces *finger* działa w tle (jako demon) na każdym stanowisku zaopatrzonym w system BSD i odpowiada na pytania zadawane za pośrednictwem sieci Internet. Czułym punktem złośliwej ingrencji było czytanie danych wejściowych bez sprawdzania występowania nadmiaru. Program Morrisa kierował do procesu *finger* pytanie w postaci napisu o długości 536 bajtów, tak chytrze skonstruowanego, aby przepiąć bufor przydzielony na wejściu i zapisać obszar stosu. Zamiast wracać do miejsca w programie głównym, z którego wywołał go Morris, demon *finger* był kierowany do procedury umieszczonej w owym zaborczym, 536-bajtowym napisie, położonym obecnie na stosie. W tej procedurze wykonywano polecenie */bin/sh*, które – jeśli kończyło się sukcesem – udostępniało robakowi zdalną powłokę systemu na atakowanej maszynie.

Błąd, który wykorzystano w programie *sendmail*, również polegał na zastosowaniu procesu-demonu do podstępnie spreparowanych danych wejściowych. Demon *sendmail* kieruje pocztę elektroniczną w środowisko sieci. Kod uruchomieniowy tego programu umożliwia osobom testującym sprawdzanie i wyświetlanie stanu systemu pocztowego. Wariant uruchomieniowy przydaje się administratorom systemu i często jest włączony jako proces drugoplanowy. Do arsenalu swoich środków ataku Morris dodał wywołanie programu *debug*, który zamiast określenia adresu użytkownika – co wystąpiłoby w normalnym testowaniu – wydawał zestaw poleceń wysyłających kopię programu haczącego i rozpoczęjących jego wykonanie.

Znalazłszy się na miejscu, robak główny podejmował systematyczne próby poznania hasel użytkowników. Zaczynał od sprawdzania prostych wariantów bez hasła lub z hasłem zbudowanym jako przedstawienie nazwy konta użytkownika, następnie dokonywał porównań z wewnętrznym słownikiem

zawierającym wybór 432 ulubionych haseł, po czym przechodził do fazy końcowej, tj. do wyprowadzania w charakterze potencjalnego hasła każdego słowa ze standardowego, podręcznego słownika przechowywanego w systemie UNIX. Ten drobiazgowy i wydajny, trzyfazowy algorytm łamania haseł umożliwił robakowi zyskiwanie kolejnych dostępów do kont innych użytkowników zarządzanego systemu. Wówczas robak przeszukiwał na kontach, do których właśnie udało się mu włamać, pliki danych dla programu *rsh*. Sprawdzeniu podlegał każdy wpis *rsh* określający możliwość zdalnego rejestrowania, wskutek czego – jak już opisaliśmy wcześniej – robak mógł uzyskiwać dostęp do kont użytkowników w systemach zdalnych.

Przy każdym nowym dostępie program-robak poszukiwał już aktywnych swoich kopii. W przypadku ich znalezienia pozostawał tylko co siódma kopię – inne zaprzestawały działania. Gdyby robak kończył działanie po zauważeniu każdej kopii, być może pozostałby nie wykryty. Pozwolenie na pozostawianie co siódmej kopii (mające zapewne na celu pokrzyżowanie wysiłków zmierzających do powstrzymania jego rozprzestrzeniania przez podkładanie na przynętę „fałszywych” robaków) doprowadziło do masowego zarobaczenia systemów Sun i VAX w sieci Internet.

Te same właściwości środowiska sieci uniksowej, które pomagały robakowi w jego rozprzestrzenianiu się, okazały się pomocne w zahamowaniu jego pochodu. Łatwość komunikacji elektronicznej, mechanizmy kopiowania plików źródłowych i binarnych do odległych maszyn, a także dostęp do kodu źródłowego, jak i do fachowych ekspertyz umożliwiały połączenie wysiłków na rzecz szybkiego znalezienia rozwiązania. Wieczorem dnia następnego, tj. 3 listopada, w internetowym obiegu między administratorami pojawiły się metody powstrzymywania agresywnego programu. Po kilku dniach opracowano stosowne pakiety naprawcze, usuwające lukę w użytkowym systemie bezpieczeństwa.

Jedną z naturalnych reakcji było pytanie o motywy, którymi kierował się Morris, uwalniając robaka. Działanie to oceniono zarówno jako nieszkodliwy figiel, który wymknął się spod kontroli, jak i w kategoriach poważnego przestępstwa kryminalnego. Zważywszy na złożoność przygotowań do ataku trudno było uznać, że wypuszczenie robaka lub zakres jego oddziaływania były niezamiczrzone. Program-robak podejmował starannie wymierzone kroki, aby zacierać po sobie ślady i odpierać próby powstrzymania jego ekspansji. Niemniej jednak program nie zawierał żadnego kodu, który miałby uszkadzać lubniszczyć systemy, w których działał. Jest oczywiste, że autor miał kwalifikacje, aby dołączyć tego rodzaju polecenia. W istocie, w kodzie wciążającym występowaly struktury danych, które mogły być użyte do przenoszenia konia trojańskiego lub programów wirusowych (p. 20.5.2). Rzeczywiście zachowanie programu mogło dostarczyć interesujących obserwacji, lecz nie

dało poważnych podstaw do wzięcia pod uwagę motywu szpiegowskiego. Nie pozostawiają natomiast miejsca na spekulacje skutki prawne tego czynu: sąd federalny skazał Morrisa i ogłosił wyrok obejmujący trzy lata w zawieszeniu, 400 godzin prac na cele publiczne oraz 10 000 USD grzywny. Koszty procesu Morrisa przekroczyły prawdopodobnie 100 000 USD.

20.5.2 Wirusy

Inną formą ataku na zasoby komputera jest *wirus* (ang. *virus*). Podobnie jak robaki, wirusy są tak skonstruowane, aby rozchodziły się po innych programach i siąły spustoszenie w systemie, zmieniając lub niszcząc pliki i doprowadzając go do załamania lub złego działania programów. Podeczas gdy robak ma postać kompletnego, wolnostojącego programu, wirus jest fragmentem kodu osadzonym w poprawnym programie. Wirusy są poważnym problemem dla użytkowników komputerów, zwłaszcza w systemach mikrokomputerowych. Komputery z systemami dla wielu użytkowników są na ogół niepodatne na wirusy, ponieważ programy wykonywalne są chronione przez system operacyjny przed wpisywaniem do nich czegokolwiek. Jeżeli nawet wirus zainfekuje program, to jego siły są ograniczone, ponieważ inne elementy systemu są chronione. Systemy z jednym użytkownikiem nie mają takiej ochrony i wskutek tego wirus ma wolne pole do działania.

Wirusy są zazwyczaj roznoszone przez użytkowników sprawdzających zawiirusowane programy z powszechnie dostępnych stanowisk nowości lub wymieniających między sobą zainfekowane dyskietki. Zdarzenie, do którego doszło w 1992 r. za sprawą dwu studentów z Cornell University, może posłużyć za ilustrację. Studenci ci opracowali trzy gry na komputer Macintosh, osadzając w nich wirusa, po czym rozpowszechnili je siecią Internet między światowe archiwa cprogramowania. Wirusa odkryto wtedy, kiedy pewien profesor matematyki z Walii sprowadził te gry i programy antywirusowe, zamontowane w jego systemie, wszczęły alarm. Około 200 innych użytkowników również sprawdziło te gry. Choć wirus nie niszczył danych, mógł się roznosić wraz z plikami aplikacji i powodować takie trudności, jak długotrwałe opóźnienia i złe działanie programów. Odnalezienie autorów było rzeczą łatwą, gdyż gry były wysyłane elektronicznie z konta uniwersytetu Cornell. Władze stanu Nowy Jork zaarrestowały studentów pod zarzutem wykroczenia, oskarżając ich o ingerencję w dobra komputerowe i inne czyny.

W innym przypadku programista z Kalifornii, rzucający się z żoną, podarował jej dysk do załadowania w komputerze będącym przedmiotem sporu. Dysk zawierał wirusa, który wykasował wszystkie pliki z systemu. Mąż został zaarrestowany i oskarżony o zniszczenie mienia.

Od czasu do czasu zbliżające się ataki wirusowe są naglaśniane przez środki masowego przekazu. Było tak z wirusem Michał Anioł, który został zaplanowany na usuwanie plików z zainfekowanych dysków w dniu 6 marca 1992 r., w 517 rocznicę narodzin renesansowego artysty. Ze względu na szeroki rozgłos wokół tego wirusa większość stanowisk komputerowych w USA zlokalizowało go i zniszczyło, zanim zdążył się uaktywnić, toteż wyrządzone przez niego szkody były znikome lub obeszły się bez żadnych szkód. Przypadki takie alarmują jednak opinię publiczną o zagrożeniu wirusowym. Programy antywirusowe sprzedają się obecnie znakomicie. Ich praca polega na wyszukiwaniu we wszystkich programach w systemie specyficznych szablonów rozkazów tworzących kod wirusa. Gdy szablon taki zostanie wykryty, wówczas jego rozkazy usuwa się, „odkażając” program. Katalogi tego rodzaju pakietów oprogramowania znajdujących się na rynku zawierają setki wirusów branych pod uwagę podczas szukania.

Najlepszą ochroną przed wirusami komputerowymi jest profilaktyka, czyli praktykowanie bezpiecznej pracy z komputerem (ang. *safe computing*). Dokonywanie zakupów nie rozpakowanego oprogramowania od dostawców i unikanie zaopatrywania się w kopie darmowe lub pirackie w źródłach publicznych oraz wymiany dyskietek jest najbezpieczniejszym sposobem zapobiegania infekcji. Jednak nawet nowe kopie legalnych aplikacji programowych nie są odporne na zakażenie wirusem. Zdarzały się przypadki zarażania przez niezdolnych pracowników firmy komputerowej głównych kopii oprogramowania, aby narazić sprzedającą to oprogramowanie firmę na straty finansowe.

Inna metoda zabezpieczania, choć nie zapobiega infekcji, pozwala ją wcześniej wykryć. Użytkownik musi rozpocząć od całkowitego sformatowania dysku twardego, zwłaszcza sektora rozruchowego, który jest częstym celem ataków wirusowych. Następnie ładuje się tylko bezpieczne kopie oprogramowania i oblicza sumę kontrolną każdego pliku. Wykaz sum kontrolnych musi być przechowyany w sposób nie zagrożony nieupoważnionym dostępem. Podczas każdego rozruchu systemu można ponownie obliczyć za pomocą programu sumy kontrolne i porównać je z pierwotnym wykazem. Każdaauważona różnica będzie służyć jako ostrzeżenie o możliwym zainfekowaniu.

Ze względu na działanie między systemami robaki i wirusy zalicza się z reguły do zagadnień bezpieczeństwa, a nie ochrony.

20.6 ■ Nadzorowanie zagrożeń

Bezpieczeństwo systemu można poprawić za pomocą dwóch sposobów zarządzania. Jednym z nich jest *nadzorowanie zagrożeń* (ang. *threat monitoring*). System może śledzić podejrzane zachowania i próbować wykrywać

naruszenia bezpieczeństwa. Typowym przykładem tego schematu jest zliczanie przez system z podziałem czasu niewłaściwych haseł podanych przy rejestraniu się użytkownika. Więcej niż kilka złych prób może być sygnałem usiłowania odgadnięcia hasła.

Innym powszechnie stosowanym sposobem jest prowadzenie *dziennika kontroli* (ang. *audit log*). W dzienniku tym zapisuje się po prostu czas, nazwę użytkownika i rodzaje wszystkich dostępów do obiektu. Po naruszeniu bezpieczeństwa można skorzystać z dziennika kontroli w celu ustalenia, jak i kiedy do tego doszło oraz – być może – zakresu spowodowanych uszkodzeń. Informacja taka może być pozytywna przy usuwaniu skutków ingerencji, a niewykluczone, że może się też przydać do opracowania lepszego oszacowania stanu bezpieczeństwa w celu unikania przyszłych problemów. Niestety, dzienniki takie mogą stawać się bardzo duże, a ich utrzymywanie zajmuje zasoby systemowe, których wskutek tego nie można przydzielić użytkownikom.

Zamiast więc rejestrować czynności systemu, można analizować go określowo pod kątem wyszukiwania luk w bezpieczeństwie. Analizę taką można wykonywać w chwilach względnie małego ruchu w komputerze, dzięki czemu mniej absorbuje ona system. Analizie może podlegać wiele aspektów systemu:

- krótkie lub łatwe do odgadnięcia hasła;
- nieuprawnione programy manipulujące przywilejem *set-uid*, jeżeli w systemie występuje ten mechanizm;
- działalność nieupoważnionych programów w katalogach systemowych;
- nieoczekiwane długo wykonywane procesy;
- nieodpowiednia ochrona katalogów, zarówno należących do użytkowników, jak i systemowych;
- nieodpowiednia ochrona plików z danymi systemowymi, takimi jak plik haseł, moduły sterujące urządzeń lub nawet samo jądro systemu operacyjnego;
- niebezpieczne wpisy na ścieżkach wyszukiwania programów (np. koń trojański, omówiony w p. 20.4.1);
- zmiany w programach systemowych wykryte za pomocą sum kontrolnych.

Każde uchybienie wykryte wskutek analizy bezpieczeństwa może być skorygowane automatycznie lub zameldowane administratorowi systemu.

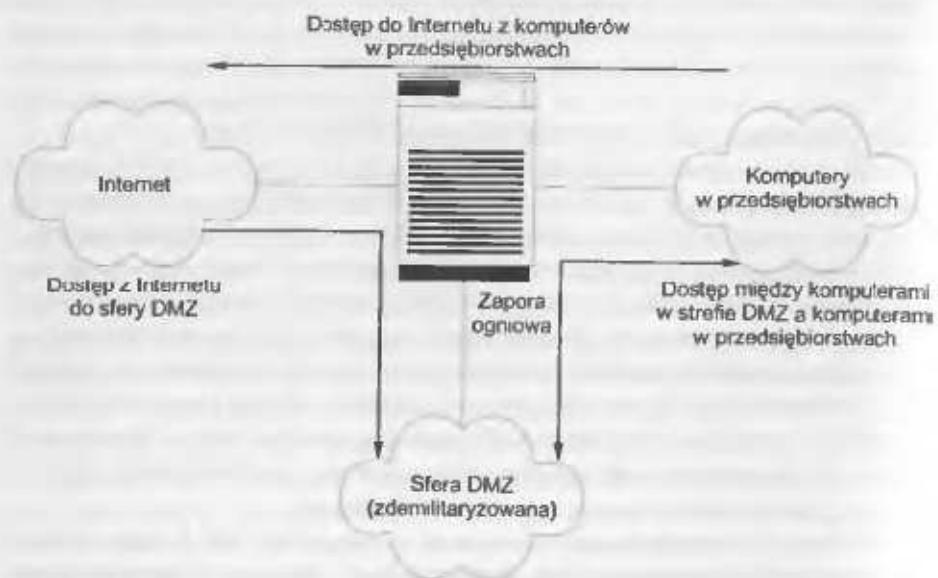
Bezpieczeństwo komputerów pracujących w sieciach jest o wiele bardziej zagrożone niż systemów autonomicznych. Zamiast z ryzykiem ataku ze zna-

nego zbioru punktów dostępu, w rodzaju bezpośrednio podłączonych terminali, mamy do czynienia z atakami pochodząymi z nieznanego i bardzo wielkiego zbioru punktów dostępu – jest to potencjalnie bardzo poważny problem bezpieczeństwa. Narażone są również, choć w mniejszym stopniu, systemy przyłączone za pomocą modemów do linii telefonicznych.

W rzeczywistości rząd federalny USA uważa systemy za bezpieczne tylko w takim stopniu, w jakim są bezpieczne ich najodleglejsze połączenia. Na przykład system ściśle tajny powinien być dostępny tylko z wnętrza budynku, który również jest w wysokim stopniu zabezpieczony. System traci kwalifikację ściśle tajnego, jeśli dochodzi do jakiegokolwiek rodzaju komunikacji z otoczeniem zewnętrznym. W niektórych organach rządowych podejmuje się szczególne środki bezpieczeństwa. Gniazda służące do podłączania terminali w celu komunikacji z zabezpieczonym komputerem są na czas nieużywania terminalu zamkane w sejfie w pomieszczeniu urzędu. Osoba chcącą mieć dostęp do komputera musi znać kombinację zamka mechanicznego oraz informację uwierzytelniającą ją w samym komputerze.

Niestety, administratorzy systemowi i specjalisci od bezpieczeństwa komputerowego często nie są w stanie zamknąć maszyny w pomieszczeniu i zakazać do niej jakiegokolwiek dostępu z zewnątrz. Na przykład sieć Internet łączy obecnie miliony komputerów. Dostęp do Internetu zaczyna mieć dla wielu przedsiębiorstw i osób fizycznych znaczenie kluczowe i staje się nieodzowny. Jak w każdym klubie zrzeszającym miliony członków, pośród wielu przyzwoitych osób znajdzie się pewna ich liczba o nagannym zachowaniu. Zły członkowie mają wiele narzędzi, których mogą używać w sieci Internet w celu uzyskiwania dostępu do połączonych nią komputerów, choćby tak jak to zrobił internetowy robak Morrisa.

Powstaje następujące pytanie: jak bezpiecznie łączyć godne zaufania komputery przez niepewną sieć? Jedno z rozwiązań polega na zastosowaniu *zapory ogniodżowej* (ang. *firewall*) do oddzielania systemów zaufanych od niegodnych zaufania. Zapora ogniodżowa nazywa się komputer lub ruter występujący między częścią zaufaną i niepewną. Ogranicza on dostęp sieciowy między dwiema *domenami bezpieczeństwa* (ang. *security domains*) oraz nadzoruje i rejestruje wszystkie połączenia. Serwery usług WWW (ang. *web servers*) stosują na przykład protokół *http* do komunikacji z przeglądarkami stron WWW. Zapora ogniodżowa może więc przepuszczać komunikaty tego protokołu. Robak internetowy Morrisa korzystał z protokołu *finger*, aby włamywać się do komputerów, zatem można nie zezwolić na przechodzenie informacji z protokołu *finger*. W rzeczywistości zapora ogniodżowa może wydzielić w sieci wiele domen. W typowej implementacji wyróżnia się w sieci internetowej domenę niepewną, drugą domenę częściowo zabezpieczonej sieci o ograniczonym zaufaniu – nazywaną strefą zdemilitaryzowaną (ang. *demilitarized*



Rys. 20.2 Bezpieczeństwo w sieci uzyskiwanie przez wydzielenie domen odseparowanych zaporą ogniotrwałą

zone – DMZ), oraz komputery zamontowane w przedsiębiorstwach jako trzecią domenę (rys. 20.2). Zezwala się na połączenia bieżące z Internetu do komputerów w strefie zdemilitaryzowanej oraz z komputerów w przedsiębiorstwach do Internetu. Nie pozwala się natomiast na połączenia wiodące z Internetu lub komputerów ze strefy DMZ do komputerów w przedsiębiorstwach. Dzięki temu wszystkie dostępły są ograniczone i żadne systemy DMZ, do których włamanie nastąpiło za pomocą protokołów przepuszczanych przez zaporę ogniotrwałą, nie będą mimo to w stanie uzyskać dostępu do komputerów w przedsiębiorstwach.

20.7 ■ Szyfrowanie

Ochrona, jaką tworzą rozmaite udogodnienia w systemie operacyjnym w celu uwierzytelniania, może okazać się zbyt mała w przypadku szczególnie wrażliwych danych. Ponadto wraz ze wzrostem popularności sieci komputerowych coraz więcej wrażliwej (klasyfikowanej) informacji przesyła się kanałami, w których jest możliwy podsłuch i przechwytywanie komunikatów. Aby zachować w tajemnicy takie delikatne informacje, są potrzebne mechanizmy pozwalające użytkownikowi chronić dane przesypane przez sieć.

Szyfrowanie jest powszechnie stosowaną metodą ochrony informacji przesyłanej przez niepewne łącza. Podstawowy mechanizm działa następująco:

1. Informacja (tekst) zostaje *zaszyfrowana* (ang. *encrypted*), czyli zakodowana z jej początkowej, czytelnej postaci (nazywanej *tekstem czystym*^{*} na postać wewnętrzną (zwaną *tekstem zaszyfrowanym*; ang. *cipher text*). Ta wewnętrzna postać tekstu – choć można ją czytać – jest pozbawiona jakiegokolwiek sensu.
2. Tekst zaszyfrowany można zapamiętać w pliku otwartym do czytania lub przesłać niechronionymi kanałami.
3. Aby nadać sens tekowi zaszyfrowanemu, odbiorca musi go *odszafrować* (ang. *decrypt*), czyli zdekodować z powrotem do postaci tekstu czystego.

Nawet jeśli zaszyfrowana informacja dostanie się do rąk osoby nieuprawnionej, to pozostałe bezużyteczna, chyba że zostanie odszyfrowana. Głównym wyzwaniem w zastosowaniu tego podejścia jest opracowanie schematów szyfrowania niemożliwych (lub przynajmniej bardzo trudnych) do złamania.

Istnieje wiele metod spełniających to wymaganie. W najpopularniejszych stosuje się ogólny algorytm szyfrowania E , ogólny algorytm deszyfrowania D i tajny klucz (lub klucze) dostarczany wraz z każdą aplikacją. Niech E_k i D_k oznaczają odpowiednio algorytmy szyfrowania i deszyfrowania do użytku w konkretnej aplikacji z kluczem k . Wówczas algorytm szyfrowania musi mieć następujące właściwości dla każdego komunikatu m :

1. $D_k(E_k(m)) = m$.
2. Zarówno obliczenie E_k , jak i D_k można wykonać efektywnie.
3. Bezpieczeństwo systemu zależy tylko od tajności klucza, a nie od tajności algorytmów E lub D .

Schemat tego rodzaju, nazwany *standardem szyfrowania danych* (ang. *data-encryption standard* – DES), został przyjęty przez National Bureau of Standards^{**}. Kłopot w tym schemacie sprawia *dystribucja klucza* (ang. *key distribution*)^{***}. Zanim dojdzie do komunikacji, klucze tajne muszą zostać bezpiecznie przesłane zarówno do nadawcy, jak i do odbiorcy. Zadania tego nie można wykonać skutecznie w środowisku sieci komunikacyjnej. Trudność

* Z ang. *clear text*. Inne spotykane określenie angielskie to *plain text*, czyli tekst jawni lub otwarty. – Przyp. tłum.

** Narodowe Biuro Standardów (USA). – Przyp. tłum.

*** W 1997 r. przybył temu standardowi nowy kłopot – połączonymi siłami komputerów sieciowych wykazano, że jego klucz, nawet w wersji 56-bitowej, można złamać w czasie budzącym realne zagrożenie bezpieczeństwa. – Przyp. tłum.

tę można usuwać przez zastosowanie schematu *szyfrowania z kluczem jawnym*^{*} (ang. *public-key encryption*). Kazdy użytkownik ma zarówno klucz jawnego, jak i prywatny, a dwu użytkowników może się komunikować, znając tylko wzajemnie swoje klucze jawnego.

Oto jak wygląda algorytm oparty na tym pomysle. Uważa się, że algorytm tego prawie nie można złamać. Klucz jawnego szyfrowania stanowi para (e, n) , a klucz prywatny para (d, n) , przy czym $e, d \in n$ są dodatnimi liczbami całkowitymi. Każdy komunikat jest reprezentowany jako liczba całkowita z przedziału od 0 do $n - 1$. (Długi komunikat dzieli się na ciąg krótszych komunikatów, z których każdy da się przedstawić w postaci takiej liczby). Funkcje E i D są zdefiniowane następująco:

$$\begin{aligned} E(m) &= m^e \bmod n = C, \\ D(C) &= C^d \bmod n. \end{aligned}$$

Głównym problemem jest dobór kluczy szyfrowania i deszyfrowania. Liczba całkowita n jest obliczona jako iloczyn dwóch wielkich (100 lub więcej cyfrowych), losowo wybranych liczb pierwszych p i q :

$$n = p \times q.$$

Wartość d wybiera się tak, aby była wielką, losową liczbą całkowitą względnie pierwszą z $(p - 1) \times (q - 1)$. To znaczy d spełnia równość

$$\text{największy wspólny dzielnik } [d, (p - 1) \times (q - 1)] = 1.$$

Wreszcie liczba całkowita e jest obliczona na podstawie p, q i d jako multiplikatywna odwrotność d modulo $(p - 1) \times (q - 1)$. To znaczy e spełnia równość

$$e \times d \bmod (p - 1) \times (q - 1) = 1.$$

Zauważmy, że chociaż n jest jawnie, p i q nie są powszechnie znane. Warunek ten jest do przyjęcia na podstawie dobrze znanego faktu, że rozłożenie liczby n na czynniki jest bardzo trudne. W konsekwencji również liczb d i e nie można łatwo odgadnąć.

Zilustrujmy ten schemat na przykładzie. Niech $p = 5$ i $q = 7$. Wtedy $n = 35$ i $(p - 1) \times (q - 1) = 24$. Ponieważ 11 jest względnie pierwsze z 24, możemy wybrać $d = 11$ i $e = 11$, gdyż $11 \times 11 \bmod 24 = 121 \bmod 24 = 1$. Załączmy teraz, że $m = 3$. Wtedy

$$C = m^e \bmod n = 3^{11} \bmod 35 = 12$$

$$C^d \bmod n = 12^{11} \bmod 35 = 3 = m.$$

Zatem, jeśli zaszyfrujemy m przy użyciu e , to możemy odszyfrować m przy użyciu d .

* Lub inaczej: *publicznym*. – Przyp. red

20.8 ■ Klasyfikacja poziomów bezpieczeństwa komputerowego

W dokumencie zatytułowanym „Kryteria oceny zaufanego systemu komputerowego” opracowanym przez Ministerstwo Obrony USA* wyróżnione cztery działy bezpieczeństwa systemów: A, B, C i D. Najniżej zakwalifikowano dział D, czyli ochronę minimalną. Dział D obejmuje tylko jedną klasę, odnoszoną do systemów, które po poddaniu ocenie nie spełniły wymagań zadnej z innych klas bezpieczeństwa. W dziale D znajdują się na przykład systemy MS-DOS oraz Windows 3.1.

Dział C – następny stopień bezpieczeństwa – określa dowolne reguły ochrony i odpowiedzialności użytkowników oraz ich działań przez umożliwianie oglądu w ich poczynania (ang. *audit*). Dział C ma dwa poziomy: C1 i C2. System klasy C1 zawiera środki kontroli, które umożliwiają użytkownikom ochronę informacji i nie pozwalały innym użytkownikom na przypadkowe czytanie lub niszczenie ich danych. W środowisku klasy C1 współpracujący użytkownicy mają dostęp do danych na tych samych poziomach uwrażliwienia. Większość wersji systemu UNIX jest klasy C1.

Zespół wszystkich systemów ochronnych w systemie komputerowym (sprzęt, oprogramowanie, oprogramowanie układowe), które we właściwy sposób wymuszają zasady bezpieczeństwa, nazywa się *bazą bezpieczeństwa komputerowego* (ang. *Trusted Computer Base* – TCB). W systemie klasy C1 baza bezpieczeństwa sprawuje nadzór nad dostępem między użytkownikami i plikami na zasadzie umożliwiania użytkownikowi określania i kontrolowania reguł dzielenia obiektów z nazwanymi osobami lub zdefiniowanymi grupami użytkowników. Ponadto baza TCB wymaga, aby użytkownik przedstawiał się przed rozpoczęciem jakichkolwiek działań, w których baza TCB pośredniczy. Prezentacji tej dokonuje się za pomocą chronionego mechanizmu lub hasła. Dane uwierzytelniające są chronione w bazie TCB, aby były niedostępne dla nieuprawnionych użytkowników.

W systemie klasy C2 do wymagań systemu klasy C1 dodaje się indywidualny poziom kontroli dostępu. Na przykład prawa dostępu do pliku można określić w odniesieniu do poszczególnych osób. Ponadto administrator systemu może śledzić wybiórczo działania dowolnego użytkownika lub grupy, korzystając z indywidualnej identyfikacji. Baza TCB również ochrania samą siebie przed zmienianiem jej kodu lub struktur danych. Ponadto zadane informacje wytworzzone przez wcześniejszego użytkownika nie zostaną udostępnione później użytkownikowi, który zwraca się do obiektu pamięci już prze-

* Nazwa oryginalna: *U.S. Department of Defence Trusted Computer System Evaluation Criteria*. – Przyp. tłum.

kazanego z powrotem do systemu. Niektóre specjalnie zabezpieczone wersje systemu UNIX mają certyfikaty poziomu C2.

Systemy z ochroną obowiązkową, kwalifikowane do działu B, mają wszystkie właściwości systemów klasy C2, a ponadto dołącza się w nich do każdego obiektu kategorię jego uwrażliwienia. W bazie bezpieczeństwa klasy B1 dla każdego obiektu w systemie określa się kategorię bezpieczeństwa; korzysta się z niej przy podejmowaniu decyzji w sprawie obowiązkowej kontroli dostępu. Na przykład użytkownikowi z poziomu poufnego nie wolno sięgać do pliku na wyższym poziomie bezpieczeństwa. W bazie TCB określa się również ramy poziomów uwrażliwienia dla każdej strony wyników zdanych do czytania przez człowieka. Oprócz kontrolowania zwykłych informacji uwierzytelniających, tj. nazwy użytkownika i hasła, baza TCB zajmuje się sprawdzaniem rzetelności (ang. *clearance*) i upoważnianiem poszczególnych użytkowników i dostarcza co najmniej dwu poziomów bezpieczeństwa. Są to poziomy hierarchiczne, tak więc użytkownik ma prawo dostępu do obiektów, których kategorie wrażliwości są równe lub mniejsze niż jego poziom zaufania (ang. *security clearance*). Na przykład użytkownik z poziomu tajnego mógłby mieć dostęp do pliku na poziomie poufnym mimo braku innych uregulowań dostępu. Procesy również podlegają izolacji dzięki stosowaniu rozłącznych przestrzeni adresowych.

W systemie klasy B2 kategorie uwrażliwienia są przypisywane wszystkim zasobom, na przykład obiektem pamięci. Urządzeniom fizycznym przypisuje się minimalne i maksymalne poziomy bezpieczeństwa, egzekwowane przez system w celu wymuszenia ograniczeń wynikających z fizycznych środowisk, w których dane urządzenia się znajdują. Ponadto system B2 udostępnia tajne kanały oraz umożliwia śledzenie zdarzeń, które mogą prowadzić do eksploatacji kanału tajnego.

W systemie klasy B3 można tworzyć wykazy kontroli dostępów określające użytkowników i grupy, którym zabrania się dostępu do obiektu o danej nazwie. W bazie TCB istnieje także mechanizm nadzorowania zdarzeń mogących wskazywać na naruszenie przyjętych zasad bezpieczeństwa. Mechanizm ten powiadamia administratora odpowiedzialnego za bezpieczeństwo i, w razie konieczności, zamyka łańcuch zdarzeń w sposób powodujący najmniejsze straty.

Najwyżej w klasyfikacji znajduje się dział A. System klasy A1 jest funkcjonalnie równoważny systemowi klasy B3, jednak stosuje się w nim formalną specyfikację projektu oraz techniki weryfikacji gwarantujące wysoki poziom pewności, że baza bezpieczeństwa komputerowego (TCB) zostanie zaimplementowana poprawnie. System wykraczający poza klasę A1 może być projektowany i realizowany w warunkach zapewniających bezpieczeństwo i przez zaufany personel.

¹ Dosłownie: etykietę bezpieczeństwa (ang. *security label*). – Przyp. tłum.

Zauważmy, że użycie bazy TCB gwarantuje tylko zdolność egzekwowania przez system zakładanej polityki bezpieczeństwa. Baza TCB nie określa, jaką politykę należy przyjąć. Na ogół w danym środowisku komputerowym wypracowuje się zasady bezpieczeństwa w celu uzyskania certyfikatu oraz rozporządza się planem uznawanym (ang. *accredited*) przez agencję bezpieczeństwa w rodzaju National Computer Security Center*. Pewne środowiska komputerowe mogą wymagać innych certyfikatów, na przykład wydanych przez TEMPEST, które strzegą przed podsluchem elektronicznym. Terminaly systemu z certyfikatem TEMPEST są osłonięte ekranami zapobiegającymi rozchodzeniu się fal elektromagnetycznych. Ekranowanie takie uniemożliwia wykrywanie informacji wyświetlanego na terminalu za pomocą aparatury umieszczonej na zewnątrz pokoju lub budynku, w którym znajduje się terminal.

20.9 ■ Przykład modelu bezpieczeństwa – system Windows NT

Oprogramowanie Windows NT firmy Microsoft jest stosunkowo nowym systemem operacyjnym, w którego projekcie przewidziano różnorodne środki bezpieczeństwa z wyróżnieniem jego poziomów – od zabezpieczeń minimalnych do bezpieczeństwa spełniającego wymogi amerykańskiej, państwowej normy C2. Poziom bezpieczeństwa przyjmowany w systemie NT na zasadzie domyślności można określić jako minimalny, lecz administrator systemu może łatwo dokonać jego stosownego podwyższenia. Program narzędziowy *C2config.exe* umożliwia administratorowi wybranie żądanego reguły bezpieczeństwa. W tym punkcie dokonamy przeglądu właściwości, które w systemie Windows NT znajdują zastosowanie do organizowania zabezpieczeń.Więcej informacji na temat systemu Windows NT i jego podstaw zawiera rozdz. 23.

Podstawą modelu bezpieczeństwa NT jest pojęcie *konta użytkownika* (ang. *user account*). System NT umożliwia założenie dowolnej liczby kont użytkowników, które można grupować w dowolny sposób. Dostęp do obiektów systemowych może być potem udzielany i cofany stosownie do potrzeb. Użytkownicy są rozpoznawani w systemie za pomocą jednoznacznego identyfikatora bezpieczeństwa (ang. *unique security ID*). Podczas rejestrowania użytkownika system tworzy dla niego żeton kontrolowanego dostępu (ang. *security access token*), który zawiera identyfikator bezpieczeństwa użytkownika, identyfikatory bezpieczeństwa każdej z grup, do której dany użytkownik należy, oraz wykaz wszelkich specjalnych przywilejów posiadanych przez

* Tzn. Narodowego Centrum Bezpieczeństwa Komputerowego. Chodzi o instytucję amerykańską – Przyp. tłum.

użytkownika. Przykładami specjalnych przywilejów mogą być: prawo tworzenia zapasowych kopii plików i katalogów, prawo wyłączania komputera, wykonywanie interakcyjnych rejestracji lub ustawianie zegara systemowego. Każdy proces wykonywany przez system NT na zamówienie użytkownika otrzymuje kopię żetonu dostępu. System korzysta z identyfikatorów bezpieczeństwa w żetonach dostępu, aby udzielać lub zakazywać dostępu do obiektów systemowych, ilekroć użytkownik lub działający w jego imieniu proces ubiega się o dostęp do obiektu. Uwierzytelnienie użytkownika konta jest zazwyczaj dokonywane za pomocą nazwy użytkownika i hasła, aczkolwiek modularna konstrukcja systemu NT pozwala na zastosowanie indywidualnych pakietów uwierzytelniania. W celu upewniania się, że użytkownik jest tym, za kogo się podaje, można na przykład zastosować skaner siatkówki oka.

W systemie NT w celu zapewniania, że program wykonywany przez użytkownika nie uzyska większych praw do systemu, niż na to pozwalały uprawnienia użytkownika, korzysta się z koncepcji *dziedziny* (ang. *subject*). Dziedzina zarządzania pozwoleniami i ich kontrolowania odnosi się do każdego programu wykonywanego przez użytkownika; w jej skład wchodzi żeton dostępu użytkownika oraz program wykonywany w imieniu użytkownika. Ponieważ system NT działa na zasadzie modelu klient-serwer, do nadzorowania dostępu używa się dwu klas dziedzin. Przykładem *prostej dziedziny* (ang. *simple subject*) jest typowy program użytkowy, który użytkownik wykonuje po zarejestrowaniu się w systemie. Prostej dziedzinie przyporządkowuje się po prostu *kontekst bezpieczeństwa* (ang. *security context*) wynikający z żetonu kontrolowanego dostępu przysługującego użytkownikowi. *Dziedzinę serwera* (ang. *server subject*) tworzy proces zrealizowany w postaci chronionego serwera, który – działając na zamówienie klienta – posługuje się kontekstem bezpieczeństwa danego klienta. Metoda umożliwiająca posługивание się przez pewien proces atrybutami bezpieczeństwa innego procesu nosi nazwę *perso-nifikacji* (ang. *impersonation*)*.

Jak zauważyliśmy w p. 20.6, śledzenie poczynan jest użyteczną techniką zabezpieczania. Możliwość śledzenia jest wbudowana w system NT i umożliwia obserwowanie wielu typowych zagrożeń bezpieczeństwa. Przykładami śledzenia mogącego się przydać do wykrywania zagrożeń jest odnotowywanie błędów występujących podczas prób rejestracji i wyrejestrowywania, co umożliwia wykrywanie prób losowego złamania hasła, zwracanie uwagi na udane rejestracje i wyrejestrowania w dziwnych godzinach, doglądanie udanych i nieudanych prób zapisywania plików wykonywalnych w celu wykry-

* Czego nie należy mylić z negatywnym „podsywaniem się”, jednym ze sposobów ataku na system, na którego określenie można spotkać w literaturze angielskiej ten sam termin. – Przyp. tłum.

wania ataku wirusów oraz udanych i nieudanych prób dostępu do plików w celu wykrywania dostępu do wrażliwych plików.

Atrybuty bezpieczeństwa obiektu w systemie NT są opisane przez *deskryptor bezpieczeństwa* (ang. *security descriptor*). Deskryptor bezpieczeństwa zawiera identyfikator bezpieczeństwa właściciela obiektu (który może zmieniać prawa dostępu), grupowy identyfikator bezpieczeństwa – używany tylko w podsystemie POSIX, dowolny wykaz kontroli dostępów, określający użytkowników lub grupy, którym wolno lub nie wolno korzystać z obiektu, oraz systemowy wykaz kontroli dostępów, który określa, jakie komunikaty śledzenia mają być generowane przez system. Na przykład w deskryptorze bezpieczeństwa pliku *foo.bar* mógłby być wpisany właściciel *avi* i następujący, dowolny wykaz kontroli dostępów:

- *avi* – pełny dostęp;
- grupa *cs* – dostęp do czytania i pisania;
- użytkownik *cliff* – zakaz wszelkiego dostępu.

Ponadto plik ten mógłby mieć w systemowym wykazie kontroli dostępów zaznaczoną konieczność śledzenia wszelkich operacji pisania. Wykaz kontroli dostępów składa się z pozycji zawierających identyfikator bezpieczeństwa danej jednostki oraz maskę definiującą wszystkie możliwe działania na obiekcie z wartościami AccessAllowed (dostęp dozwolony) lub AccessDenied (zakaz dostępu) odniesionymi do poszczególnych działań. Do plików w systemie NT można stosować następujące rodzaje dostępu: ReadData (czytanie danych), WriteData (pisanie danych), AppendData (dolaczanie danych), Execute (wykonywanie), ReadExtendedAttribute (czytanie atrybutu rozszerzonego), WriteExtendedAttribute (pisanie atrybutu rozszerzonego), ReadAttributes (czytanie atrybutów) i WriteAttributes (pisanie atrybutów). Widać, że umożliwia to z dużą precyzją kontrolowanie dostępu do obiektów.

W systemie NT obiekty są klasyfikowane jako *pojemniki* (ang. *containers*) lub *obiekty nie zamknięte* (ang. *noncontainer objects*). Obiekty-pojemniki, takie jak katalogi, mogą zawierać w sobie logicznie inne obiekty. Kiedy obiekt jest tworzony wewnątrz innego obiektu, wtedy na zasadzie domyślności nowy obiekt dziedziczy prawa po swoim przodku. Dzieje się tak również wtedy, kiedy użytkownik kopiuje plik z jednego katalogu do drugiego – plik dziedziczy wówczas prawa katalogu docelowego. Jeśli jednak pozwolenia dostępu do katalogu ulegają zmianie, to nie dotyczą one automatycznie istniejących plików ani podkatalogów. Użytkownik może tych pozwoleń udzielić jawnie, jeśli będzie tego potrzebował. Podobnie, podczas przemieszczania pliku przez użytkownika do innego katalogu, bieżące prawa dostępu są przenoszone wraz z plikiem.

Administrator systemu może również zakazać drukowania na drukarce przez cały dzień lub jego część, może też skorzystać z monitora wydajności systemu NT, aby ułatwić dostrzeżenie nadchodzących trudności. Ogólnie biorąc, można powiedzieć, że w systemie NT zrobiono sporo dobrego, aby zaopatrzyć go we właściwości pomagające zapewnić bezpieczne środowisko obliczeniowe. Jednak wiele z tych właściwości nie jest udostępnionych domyślnie, aby upodobić dostarczane środowisko do tego, do którego przywykł typowy użytkownik komputera osobistego. W celu otrzymania prawdziwego środowiska z wieloma użytkownikami administrator systemu powinien określić plan zabezpieczenia i wdrożyć go, korzystając z możliwości zawartych w systemie NT.

20.10 ■ Podsumowanie

Ochrona jest problemem wewnętrznym systemu. Przy zapewnianiu bezpieczeństwa należy uwzględniać zarówno system komputerowy, jak i otoczenie (ludzie, budynki, interesy, centrum obiektów i zagrożenia), w którym dany system jest eksploatowany.

Dane przechowywane w komputerze muszą być chronione przed nieupoważnionym dostępem, złośliwym zniszczeniem lub zmianą oraz przed przypadkowym wprowadzeniem niespójności. Ochrona przed losową utratą spójności danych jest łatwiejsza niż ochrona przed złośliwym dostępem do danych. Calkowita ochrona informacji przechowywanej w komputerach przed złośliwymi nadużyciami nie jest możliwa, jednak można utrudnić działania przestępco w takim stopniu, aby pohamować większość (jeśli nie wszystkie) zamiarów sięgania po informacje bez odpowiedniego upoważnienia.

Rozmaite metody upoważniania w systemie komputerowym mogą nie zapewniać wystarczającej ochrony bardzo cennych danych. W takich przypadkach można dane szyfrować. Zaszyfrowanych danych nie da się przeczytać, jeżeli czytający nie będzie znał sposobu ich odszyfrowania.

■ Ćwiczenia

- 20.1 Hasło może stać się wiadomością dla innych użytkowników na wiele różnych sposobów. Czy istnieje prosta metoda wykrycia, że wystąpiło takie zdarzenie? Odpowiedź uzasadnij.
- 20.2 Wykaz wszystkich haseł jest przechowywany w systemie operacyjnym. Jeśli zatem użytkownik zdola przeczytać ten wykaz, to ochrona haseł przestaje istnieć. Zaproponuj schemat pozwalający uniknąć tego pro-

blemu. (Wskazówka: użyj różnych reprezentacji wewnętrznych i zewnętrznych).

- 20.3 Eksperymentalne uzupełnienie systemu UNIX umożliwia użytkownikowi przyłączenie programu „stróża” (ang. *watchdog*) do pliku w ten sposób, że stróż jest wywoływany zawsze wtedy, kiedy jakiś program chce uzyskać dostęp do danego pliku. Stróż udziela wówczas lub odmawia dostępu do pliku. Przedstaw dwa argumenty na rzecz używania stróża w celu zabezpieczania systemu i dwa argumenty przeciw ich stosowaniu.
- 20.4 Uniksowy program COPS analizuje dany system pod kątem występowania w nim luk w bezpieczeństwie i alarmuje użytkownika o ewentualnych problemach. Jakie dwa potencjalne niebezpieczeństwa wiążą się z użyciem takiego systemu do zabezpieczania? Jak można te zagrożenia zmniejszyć lub wyeliminować?
- 20.5 Omów środki, za pomocą których zarządcy systemów podłączonych do sieci Internet mogą ograniczyć lub wyeliminować w swoich systemach bezpieczeństwo uszkodzeń spowodowanych przez robaka. Wskaz wady wynikające ze zmian, które zaproponujesz.
- 20.6 Uzasadnij lub podważ wyrok sądowy orzeczony przeciwko Robertowi Morrisowi juniorowi za to, że opracował i wdrożył robaka w sieci Internet.
- 20.7 Utwórz listę sześciu spraw dotyczących bezpieczeństwa bankowego systemu komputerowego. Zaznacz dla każdej pozycji na Twojej liście, czy dotyczy ona bezpieczeństwa w rozumieniu fizycznym, czy w odniesieniu do czynnika ludzkiego, czy też bezpieczeństwa systemu operacyjnego.
- 20.8 Jakie są dwie zalety szyfrowania danych przechowywanych w systemie komputerowym?

Uwagi bibliograficzne

Ogólne omówienie problemów bezpieczeństwa podają Hsiao i in. [183], Landwehr [239], Denning [101], Pfleeger [330] oraz Russell i Gangemi [365]. Ogólnie kwestie bezpieczeństwa podejmuje też Lobel w książce [262].

Problemy projektowania i weryfikacji systemów bezpiecznych omówili: Rushby [363] oraz Silverman [390]. Jądro bezpieczeństwa dla mikrokomputera wieloprocesorowego opisał Schell [377]. Rozproszony system bezpieczeństwa omówili Rushby i Randell [364].

Morris i Thompson w artykule [298] zajęli się bezpieczeństwem haseł. Morshedian [299] przedstawił metody zwalczania piratów hasłowych. Uwierzytelnianie hasła przy użyciu nie zabezpieczonych środków łączności jest przedmiotem artykułu Lamporta [230]. Zagadnienia łamania haseł omawia Seely [386]. Wlamania do komputerów zostały omówione przez Lehmana [246] i przez Reida [349].

Rozważania na temat bezpieczeństwa w systemie UNIX zaproponowali: Grampp i Morris [153], Wood i Kochan [440], Farrow [129 i 130], Filipski i Hanko [134], Hecht i in. [167], Kramer [224] oraz Garsinkel i Spafford [146]. Bershad i Pinkerton w pracy [34] zaprezentowali rozszerzenie systemu UNIX BSD o koncepcję stróży (ang. *watchdogs*). Pakiet COPS służący do догlądania bezpieczeństwa napisał Farmer z Purdue University. Oprogramowanie to jest dostępne dla użytkowników sieci Internet za pośrednictwem programu *ftp* z katalogu */pub/security/cops komputera ftp.uu.net*.

Spafford [398] dokonał szczegółowego omówienia technicznych aspektów robaka internetowego. Artykuł Spafforda ukazał się wraz z trzema innymi artykułami w specjalnej sekcji poświęconej robakowi internetowemu w czasopiśmie *Communications of the ACM* (1989, 32, 6).

Diffie i Hellman byli pierwszymi badaczami, którzy w artykułach [107] i [108] zaproponowali użycie schematu szyfrowania z kluczem jawnym. Algorytm zaprezentowany w p. 20.7 opiera się na schemacie szyfrowania z kluczem jawnym: został on opracowany przez Rivesta i in. [355]. Lempel [248], Simmons [391], Gifford [150], Denning [101] oraz Ahituv i in. [5] skupili się na zastosowaniu kriptografii w systemach komputerowych. Rozważania dotyczące ochrony podpisów cyfrowych zaoferowali Akl [6], Davies [93] i Denning [102 i 103].

Rząd federalny USA jest oczywiście zainteresowany bezpieczeństwem. W publikacji *Department of Defence Trusted Computer System Evaluation Criteria* [106], znanej też pod nazwą Pomarańczowej księgi, przedstawiono zbiór poziomów bezpieczeństwa i cechy, które musi mieć system operacyjny, aby zakwalifikować się do każdej kategorii bezpieczeństwa. Lektura tej książki jest dobrym punktem wyjścia do zrozumienia zagadnień bezpieczeństwa. W publikacji *Microsoft Windows NT Workstation Resource Kit* [292] zawarto opis modelu bezpieczeństwa systemu Windows NT oraz sposobów jego używania.

Część 7

PRZYKŁADY KONKRETYCH SYSTEMÓW

Możemy obecnie zestawić różne koncepcje zaprezentowane w tej książce, opisując rzeczywiste systemy operacyjne. Omówimy szczegółowo trzy systemy: wersję systemu UNIX 4.3BSD z Berkeley, system Linux oraz system Microsoft Windows NT. System 4.3BSD z Berkeley oraz system Linux wybraliśmy dlatego, że UNIX był w swoim czasie wystarczająco mały, aby można go było zrozumieć, a jednocześnie nie był to „zabawkowy” system operacyjny. Większość jego wewnętrznych algorytmów dobrano z uwagi na prostotę, a nie szybkość lub wyrafinowanie. Oba systemy – 4.3BSD i Linux – są łatwo osiągalne na wydziałach informatyki, toteż wielu studentów ma do nich dostęp*.

Opisujemy również szczegółowo system Windows NT. Ten nowy system operacyjny rodem z firmy Microsoft zyskuje na popularności nie tylko na rynku komputerów autonomicznych, lecz także na rynku serwerów dla grup roboczych. Nasz wybór systemu Windows NT wynika z tego, że daje on nam możliwość zapoznania się z nowoczesnym systemem operacyjnym, którego projekt i wykonanie zasadniczo różnią się od systemów uniksowych**.

W uzupełnieniu dc tych trzech systemów operacyjnych omawiamy krótko kilka innych znaczących systemów. Kolejność prezentacji została wybrana tak, aby uwypuklić podobieństwa i różnice systemów; nie jest ona ściśle chronologiczna i nie odzwierciedla względnej ważności systemów.

* Należący do rodziny uniksowej system Linux jest dostępny po cenie nośnika danych i upowszechniany w sieci Internet oraz przez popularne czasopisma informatyczne (zob. też rozdz. 22). – Przyp. tłum.

** Choć i on dziedziczy wiele po systemie UNIX (por. p. 21.1). – Przyp. tłum.

Z omówieniem systemu Mach, który jest nowoczesnym systemem operacyjnym zgodnym z systemem 4.3BSD, można zapoznać się na stronach WWW (URL:<http://www.bell-labs.com/topic/books/os-book/Mach.ps>)^{*}.

Przegląd systemu Nachos jest również dostępny na stronach WWW (URL:<http://www.bell-labs.com/topic/books/os-book/Nachos.ps>).

^{*} Zob. też A. Silberschatz, J.L. Peterson, P.B. Galvin: *Podstawy systemów operacyjnych*, WNT 1993, 1996; rozdz. 16, s. 570-599.

Rozdział 21

SYSTEM UNIX

Chociaż pojęcia systemów operacyjnych można rozważyć czysto teoretycznie, często warto zobaczyć, jak są one zrealizowane w praktyce. Przedstawiona w tym rozdziale pogłębiona analiza systemu operacyjnego 4.3BSD, będącego wersją systemu UNIX, służy jako ilustracja różnych koncepcji zaprezentowanych w tej książce. Analizując pełny, rzeczywisty system, możemy się przekonać, w jaki sposób liczne pojęcia omówione w tej książce wiążą się ze sobą w praktyce. Zajmiemy się najpierw krótką historią systemu UNIX i przedstawimy jego interfejsy z użytkownikiem i programistą. Następnie omówimy wewnętrzne struktury danych i algorytmy, których jądro systemu UNIX używa do obsługi interfejsu użytkownika i programisty.

21.1 ■ Historia

Pierwsza wersja systemu UNIX została opracowana w 1969 r. przez Kena Thompsona z zespołu badawczego^{*} Bell Laboratories, aby spożytkować bezczynną maszynę PDP-7. Do Thompsona dołączył wkrótce Dennis Ritchie. Thompson, Ritchie i inni członkowie tego zespołu opracowali wczesne wersje systemu UNIX.

Ritchie pracował uprzednio nad projektem MULTICS i ten właśnie projekt wywarł silny wpływ na nowy system operacyjny. Nawet nazwa „UNIX” jest kalamburem określenia MULTICS. Podstawowa organizacja systemu plików, pomysł potraktowania interpretera poleceń (powłoki – ang. *shell*) jako

* O nazwie Research Group. Przyp. tłum.

procesu użytkownika, uzycie oddzielnego procesu do kazdego polecenia, oryginalne znaki redagowania wiersza (# do kasowania ostatniego znaku i @ do kasowania całego wiersza) oraz liczne inne cechy pochodzące wprost z systemu MULTICS. Wykorzystano również pomysły z różnych innych systemów operacyjnych, jak C1SS z MIT oraz XDS-940.

Ritchie i Thompson pracowali nad systemem UNIX bez rozgłosu przez wiele lat. Praca nad pierwszą wersją systemu pozwoliła im na zaimplementowanie go na komputerze PDP-11/20 w postaci wersji drugiej. Trzecia wersja powstała w wyniku przepisania większości systemu operacyjnego na język programowania systemowego o nazwie C, użyty zamiast stosowanego do tej pory asemblera. Język C opracowano w Bell Laboratories* w celu zaprogramowania systemu UNIX. UNIX został także przeniesiony na większe modele komputera PDP-11, takie jak 11/45 i 11/70. Podczas przepisywania systemu UNIX na język C oraz w związku z przenoszeniem go na komputery, które oferowały sprzętowe wsparcie wieloprogramowości (takie jak PDP-11/45), rozszerzono go o wieloprogramowość i inne ulepszenia.

W miarę rozwoju systemu UNIX zaczęto go używać na szeroką skalę w Bell Laboratories, skąd rozpoczęło się jego stopniowe przenikanie na uniwersytety. Pierwsza wersja szeroko dostępna poza Bell Laboratories miała oznaczenie *Version 6* i pochodziła z 1976 r. (Numery wersji wczesnych systemów UNIX odpowiadają numerom wydań dokumentacji pod nazwą *UNIX Programmer's Manual*, rozpowszechnianej wraz z systemem: kod i podręczniki były weryfikowane niezależnie).

W 1978 r. do obiegu weszło wydanie systemu oznaczone jako *Version 7*. Ta wersja systemu UNIX pracowała na komputerach PDP-11/70 oraz Interdata 8/32, od niej pochodzi większość współczesnych systemów uniksowych. Przeniesiono ją m.in. wkrótce na inne modele PDP-11 oraz na serię komputerów VAX. Wersja dostępna na komputerach VAX była znana pod nazwą 32V. Badania były kontynuowane.

Po upowszechnieniu siódmego wydania systemu w 1978 r. specjalna grupa pod nazwą UNIX Support Group (USG) przejęła nadzór administracyjny i odpowiedzialność od zespołu badawczego w kwestiach dystrybucji systemu UNIX wewnętrz AT&T – firmy, z której wywodziły się zakłady Bell Laboratories. UNIX stał się produktem, a nie tylko narzędziem badawczym. Mimo to zespół badawczy pracował dalej nad własną wersją systemu UNIX, która miała służyć do celów wewnętrznych zespołu. Następne w kolejności wydanie systemu – *Version 8* – zawierało usprawnienie zwane systemem strumieniowym wejścia-wyjścia (ang. *stream I/O system*), pozwalające na elastyczne konfigurowanie modułów komunikacji międzyprocesowej jądra. Zawierała

* Autorem języka C jest D.M. Ritchie – Przyp. tłum.

ona również system zdalnych plików RFS, podobny do używanego na komputerach Sun systemu NFS. Potem nadeszły wersje 9 i 10 (tę ostatnią wersję, pochodzącą z 1989 r., udostępniono tylko wewnątrz zakładów Bell Laboratories).

Grupa USG zajmowała się głównie systemem UNIX na terenie firmy AT&T. Pierwszym systemem, opracowanym przez USG, rozprowadzonym na zewnątrz w 1982 r., był System III. System III zawierał cechy wersji 7 i 32V oraz kilku systemów uniksowych opracowanych poza zespołem badawczym. Do Systemu III zostały także włączone cechy systemu UNIX/RT, czyli wersji pracującej w czasie rzeczywistym, jak również liczne fragmenty z pakietu narzędziowego Programmer's Work Bench (PWB).

W 1983 r. grupa USG opublikowała System V, wzorowany w znacznej mierze na Systemie III. Uwalnianie się spod władzy AT&T różnych firm Bella o profilu programowania systemowego zmusiło firmę AT&T do ostrej kampanii rynkowej na rzecz Systemu V. Grupę USG przekształcono na UNIX System Development Laboratory (USDL), po czym zespół ten doprowadził do ukazania się w 1984 r. wersji UNIX System V Release 2 (V.2). Kolejne wydanie – UNIX System V Release 2, Version 4 (V.2.4) – zostało wzbogacone o nową implementację pamięci wirtualnej ze stronicowaniem w trybie kopирования przy zapisie i pamięcią dzieloną. Zespół USDL zastąpiono z kolei organem nazwanym AT&T Information Systems (AT&TIS), który w 1987 r. zajął się dystrybucją wersji System V Release 3 (V.3). W wersji V.3 zaadaptowano implementację systemu strumieni wejścia-wyjścia z wydania *Version 8* (V8) i udostępniono ją jako STREAMS. Zawiera ona również system zdalnych plików RFS, podobny do systemu NFS.

Male rozmiary, modularność i przejrzysta konstrukcja wczesnych systemów UNIX spowodowały, że prace nad systemami wzorowanymi na systemie UNIX podjęto wiele innych ośrodków informatycznych, m.in. firmy Rand i BBN, University of Illinois, Harvard University i Purdue University, a nawet korporacja DEC. Spośród grup zajmujących się rozwojem systemu UNIX i nie pochodzących z Bell Laboratories ani z AT&T najbardziej wpływowy okazał się jednak University of California w Berkeley.

Pierwszą pracą z Berkeley było dodanie w 1978 r. do systemu UNIX działającego na komputerze VAX pamięci wirtualnej, stronicowania na żądanie i zastępowania stron. Wykonali to Bill Joy i Ozalp Babaoglu na wersji 32V systemu, tworząc w ten sposób wersję 3BSD systemu UNIX. Ta wersja była pierwszą implementacją każdego z tych udoskonalień pośród systemów UNIX. Wielka przestrzeń wirtualna systemu 3BSD umożliwiła opracowywanie bardzo dużych programów, takich jak Franz LISP, pochodzący także z Berkeley. Prace nad zarządzaniem pamięcią przekonały rządową agencję DARPA (ang. Defence Advanced Research Projects Agency) do subsydiowania

nia zespołu z Berkeley w celu opracowania standardowego systemu UNIX na potrzeby rządu – w efekcie powstała wersja systemu UNIX 4BSD.

Prace nad systemem 4BSD dla agencji DARPA były nadzorowane przez komitet kierowniczy, który skupił wiele znanych osobistości z kręgów unikso-wych i organizacji sieci. Jednym z celów tych prac było przygotowanie dla DARPA podstaw do realizacji protokołów sieci Internet (TCP/IP). Stosowne oprogramowanie wykonano w sposób ogólny. W systemie 4.2BSD można utrzymywać jednolitą komunikację za pośrednictwem różnorodnych rozwiązań sieciowych, w tym sieci lokalnych (takich jak Ethernet i sieci pierścieniowe z zetonami) oraz sieci globalnych (jak np. NSFNET). Implementacja systemu 4BSD miała najważniejszy wpływ na obecną popularność tych protokołów. Użyto jej jako podstawy do wielu konkretnych implementacji wykonanych przez dostawców uniksowych systemów komputerowych, a nawet wzorowano się na niej przy opracowywaniu innych systemów operacyjnych. Ona też pozwoliła sieci Internet rozrosnąć się z 60 połączonych ze sobą sieci w 1984 r. do ponad 8000 sieci z liczbą użytkowników szacowaną w 1993 r. na 10 mln.

Ponadto w Berkeley w celu ulepszenia projektu i implementacji wdrożono do systemu UNIX wiele cech ze współczesnych mu systemów operacyjnych. Nowy moduł sterujący terminaliem zawierał wiele funkcji konwersacyjnego wierszowego edytora systemu operacyjnego TENEX (TOPS-20). W Berkeley napisano nowy interfejs użytkownika (powłokę C), nowy edytor tekstowy (ex/vi), kompilatory języków Pascal i LISP oraz wiele nowych programów systemowych. Inspiracją do pewnych ulepszeń dotyczących wydajności systemu 4.2BSD stał się system operacyjny VMS.

Oprogramowanie uniksowe z Berkeley jest rozpowszechniane w pakietach *Berkeley Software Distributions*. Wygodnie jest więc odwoływać się do systemów Berkeley VAX UNIX za pomocą oznaczeń 2BSD i 4BSD, choć miały one faktycznie kilka osobnych wydań, wśród których najbardziej znane są 4.1BSD i 4.2BSD. Ogólną numerację 2BSD i 4BSD odnosi się do wersji systemu UNIX z Berkeley działających na komputerach PDP-11 i VAX. Wydanie 4.2BSD, które ukazało się w 1983 r., było ostateczną wersją oryginalnego projektu Berkeley DARPA UNIX. System 2.9BSD jest odpowiednikiem systemu 4.2BSD dla komputerów PDP-11.

W 1986 r. ukazał się system 4.3BSD. Był on tak podobny do 4.2BSD, że jego podręczniki wszechstronnej opisywały wydanie 4.2BSD niż robiły to podręczniki tej wersji. System 4.3BSD zawierał jednak liczne zmiany wewnętrzne, włączając w to usunięcie błędów i poprawę działania. Dodano również pewne nowe udoskonalenia, w tym możliwość wykonywania protokołów Xerox Network System.

System 4.3BSD Tahoe był następną wersją, wydaną w 1988 r. Zawierał on różne nowe rozwiązania, takie jak ulepszona kontrola zagęszczenia komu-

nikatów w sieci i usprawnione działanie protokołów TCP/IP. Oddzielono również konfiguracje dysków od modułów obsługi urządzeń – określa się ją obecnie na podstawie samych dysków. Dołączono także rozszerzony mechanizm operowania strefami czasu. System 4.3BSD Tahoe opracowano w istocie dla komputera Power 6 firmy Computer Console Inc. z przeznaczeniem do użycowania w systemie CCI Tahoe, a nie – jak zwykle – dla komputera VAX. Odpowiednie wydanie dla komputera PDP-11 otrzymało symbol 2.10.1BSD, jego rozpowszechnianiem zajęła się firma USENIX Association, publikująca także podręczniki wersji 4.3BSD. Do wersji 4.3BSD Reno dodano implementację działań sieciowych według standardu ISO/OSI.

Ostatnie wydanie systemu z Berkeley, 4.4BSD, zamknięto w czerwcu 1993 r. Zawiera ono nową realizację protokołu sieciowego X.25 oraz odpowiada wymaganiom standardu POSIX. Zasadniczo odnowiono w nim też organizację systemu plików, zaopatrując go w nowy interfejs wirtualnego systemu plików i możliwość stosowego (ang. *stackable*) rozplanowywania systemów plików, dzięki której można je układać warstwami jeden nad drugim w celu łatwego dodawania nowych właściwości. To wydanie zawiera również implementację systemu NFS (zob. rozdz. 17), a także nowy system plików z zastosowaniem rejestrów (zob. rozdz. 13). System pamięci wirtualnej 4.4BSD pochodzi z systemu Mach. Dodano również kilka innych zmian; polepszono bezpieczeństwo i unowocześniono strukturę jądra. Na wydaniu 4.4 zespół z uniwersytetu w Berkeley zakończył swoje prace badawcze.

System 4BSD wybrano jako system operacyjny komputerów VAX od chwili jego pojawienia się (1979), aż do ukazania się implementacji BSD o nazwie Ultrix, opracowanej przez firmę DEC*. System 4BSD jest wciąż najodpowiedniejszy dla wielu instalacji badawczych i sieciowych. Wiele instytucji zakupiło licencję systemu 32V i zamówiło wersję 4BSD z Berkeley, nie troszcząc się nawet o otrzymanie taśmy z wersją 32V.

Obecny zbiór systemów operacyjnych UNIX nie ogranicza się jednak do produktów z zakładów Bell Laboratories (będących obecnie własnością firmy Lucent Technology) ani Berkeley University. Firma Sun Microsystems przyczyniła się do popularyzacji charakterystycznych cech odmiany BSD systemu UNIX, zaopatrując w nią swoje stacje robocze. W miarę wzrostu popularności systemu UNIX przenieszono go na wiele różnych komputerów i systemów komputerowych. Powstało wiele różnorodnych systemów uniksowych i unikso podobnych. Firma DEC dostarcza swój UNIX (pod nazwą Ultrix) dla swoich stacji roboczych; teraz zastępuje się system Ultrix innym systemem – OSF/I, także wywodzącym się z systemu UNIX. W firmie Microsoft przepi-

* Komputery VAX są produktami firmy DEC (Digital Equipment Corporation). – Przyp. tłum.

sano system UNIX na rodzinę mikroprocesorów Intel 8088 i nazwano go XENIX. Nowy produkt tej firmy – system operacyjny Windows NT – pozostaje pod dużym wpływem systemu UNIX. IBM dysponuje systemem UNIX (AIX) na swoich komputerach PC, stacjach roboczych i w dużych instalacjach. Praktycznie biorąc, UNIX występuje na prawie wszystkich komputerach ogólnego przeznaczenia; pracuje w komputerach osobistych, stacjach roboczych, minikomputerach, komputerach głównych i superkomputerach – od Apple Macintosh II po system Cray II. Dzięki temu, że jest tak szeroko dostępny, UNIX jest używany w różnych środowiskach – od akademickich aż po wojskowe i nadzór nad procesami produkcyjnymi. Większość z tych systemów opiera się na wydaniach: Version 7, System III, 4.2BSD lub System V.

Szeroka popularność systemu UNIX wśród dostawców komputerów spowodowała, że stał się on najbardziej przenośnym systemem operacyjnym, spełniającym oczekiwania użytkowników na niezależność środowiska uniksowego od jakiegokolwiek konkretnego producenta komputerów. Jednak wielka liczba implementacji systemu doprowadziła do zauważalnych różnic w programowaniu oraz interfejsie użytkownika w wersjach rozprowadzanych przez różnych dystrybutorów. Aby rzeczywiście uniezależnić się od dystrybutorów, osoby opracowujące programy użytkowe potrzebują spójnych interfejsów, które powinny umożliwiać działanie aplikacji uniksowych we wszystkich systemach UNIX. Na pewno obecnie tak się nie dzieje. Zagadnienie to nabrało wagi, od kiedy UNIX stał się ulubioną platformą pisania rozmaitych aplikacji, poczynając od baz danych, a na grafice i sieciach kończąc. Na rynku pojawiło się ogromne zapotrzebowanie na uniksowe standardy.

Podjęto prace nad kilkoma projektami standaryzacji, poczynając od */usr/group 1984 Standard* – sponsorowanego przez grupę użytkowników przemysłowych UniForum. Od tego czasu sprawą tą zajęło się wiele oficjalnych organów do spraw standaryzacji, w tym takie instytucje, jak IEEE i ISO (standard POSIX). Międzynarodowe konsorcjum X/Open Group ukończyło prace nad projektem XPG3, dokumentującym powszechnie środowisko aplikacji (ang. *Common Application Environment*), do którego zalicza się standard interfejsu IEEE. Niestety, standard XPG3 oparto na wstępnych założeniach standardu języka ANSI C, zamiast na jego ostatecznym standardzie, wskutek czego wymagał on przeróbek. W efekcie w 1993 r. pojawił się standard XPG4. W 1989 r. komitet standaryzacyjny ANSI ustandaryzował język programowania C, udostępniając specyfikację języka ANSI C*, do której szybko dostosowali się dostawcy. W miarę postępu tych prac różne odmiany

* Zob. B.W. Kernighan, D.M. Ritchie: Język ANSI C. WNT, Warszawa 1994. – Przyp. tłum.

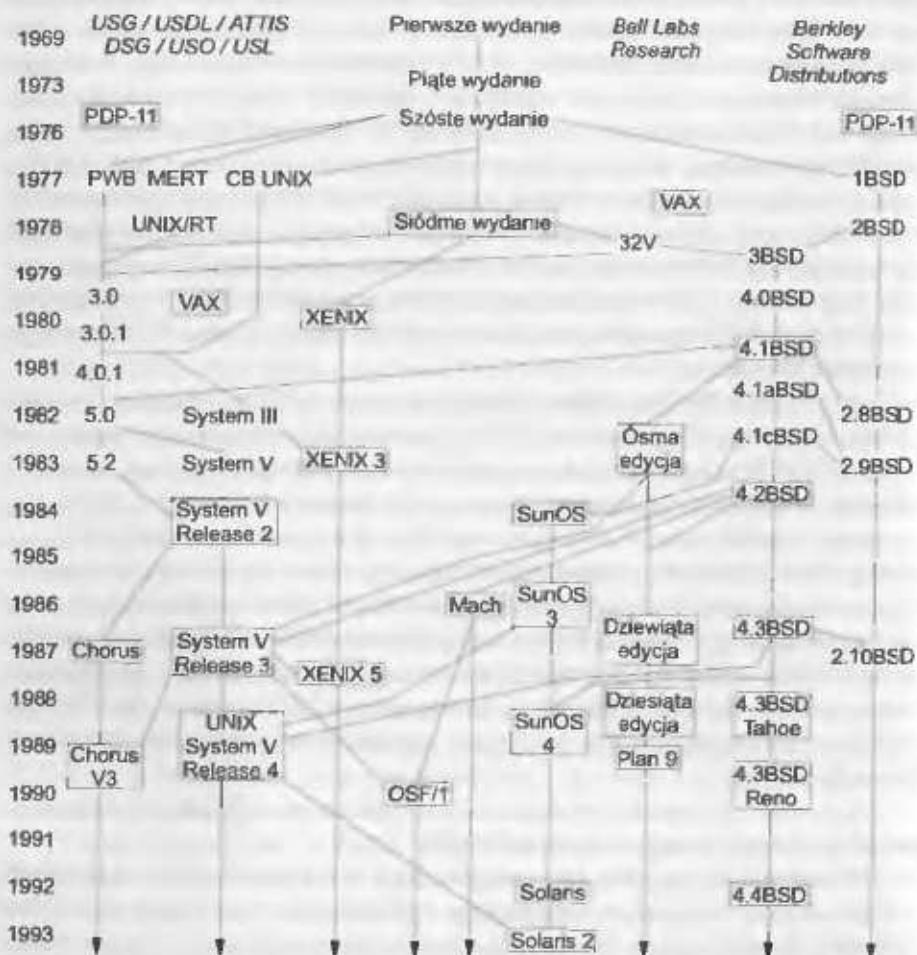
systemu UNIX będą się zbiegać ze sobą, doprowadzając do powstania jednego interfejsu programowania w systemie UNIX, co wpłynie na jeszcze większą popularność tego systemu. W rzeczywistości istnieją dwa oddzielne ośrodkи skupiające głównych dostawców systemów uniksowych, którzy pracują nad tym problemem: zarządzane przez firmę AT&T stowarzyszenie UNIX International (UI) oraz konsorcjum Open System Foundation (OSF) – oba one przyjęły standard POSIX. Ostatnio wielu dostawców należących do tych dwóch grup doszło do porozumienia w sprawie dalszej standaryzacji (uzgodnienia COSE); dotyczyło to środowiska okien Motif, rozwiązań sieciowych ONC+ (które zawiera opracowane przez firmę Sun standardy RPC i NFS) oraz DCE (zawierające system plików AFS i pakiet RPC wywołań zdalnych procedur).

W firmie AT&T w 1989 r. zastąpiono grupę ATTIS zespołem o nazwie UNIX Software Organisation (USO), który rozesłał pierwszy połączony system UNIX, oznaczony jako czwarte wydanie Systemu V (ang. *System V Release 4 – SVR4*). System ten łączy cechy Systemu V, wersji 4.3BSD oraz systemu SunOS firmy Sun, w tym: możliwość używania długich nazw, system plików z Berkeley, zarządzanie pamięcią wirtualną, dowiązania symboliczne, wiele grup dostępu, sterowanie zadaniami i niezawodne sygnały. Jest on także zgodny z opublikowanym standardem POSIX.1 ze zbioru standardów POSIX. Po wyprodukowaniu systemu SVR4 zespół USO przekształcił się w osobną, lecz pozostającą pod nadzorem AT&T, firmę o nazwie Unix System Laboratories (USL); w 1993 r. została ona kupiona przez korporację Novell, Inc.

Rysunek 21.1 zawiera podsumowujące zestawienie obrazujące zależności między różnymi wersjami systemu UNIX.

Z osobistego projektu dwu pracowników firmy Bell Laboratories system UNIX wyrósł na system operacyjny definiowany przez międzynarodowe ośrodkи normalizacyjne. Mimo to system ten jest wciąż interesujący z akademickiego punktu widzenia. Jesteśmy przekonani, że UNIX stał się i pozostał ważnym elementem teorii i praktyki systemów operacyjnych. Jest on wspaniałym obiektem studiów akademickich. Na przykład systemy operacyjne Tunis, Xinu oraz Minix są oparte na koncepcjach systemu UNIX, ale zostały opracowane z wyraźnym celem edukacyjnym. Wciąż opracowuje się mnóstwo systemów badawczych powiązanych z systemem UNIX, w tym takie systemy, jak Mach, Chorus, Comandos i Roisin. Pierwi twórcy systemu UNIX – Ritchie i Thompson – zostali za swoją pracę uhonorowani w 1983 r. nagrodą Turinga przyznawaną przez organizację Association for Computing Machinery.

Wersja systemu UNIX, którą posłużyono się w tym rozdziale, to odmiana systemu 4.3BSD dla komputera VAX. System ten wybrano ze względu na to,



Rys. 21.1 Historia wersji systemu UNIX (zaczerpnięty z: Leffler S., Karels M., McKusick M., Quarterman J. *The Design and Implementation of the 4.3 BSD UNIX Operating System*, (fig 1.1, p.5). ©1989 Addison Wesley Publishing Company Inc., Reprinted by permission of Addison Wesley Longman)

że zrealizowano w nim wiele interesujących koncepcji dotyczących systemów operacyjnych, takich jak stronicowanie na żądanie z zastosowaniem gron i pracy w sieci. Ta wersja systemu wywarła również wpływ na inne systemy uniksowe, ich standardy oraz na działania sieciowe. Implementację dla komputera VAX wybrano z tego powodu, że system 4.3BSD został opracowany dla komputera VAX, który ciągle jest wygodnym punktem odniesienia, po-

mimo widocznej ostatnio mnogości implementacji na innych rodzajach sprzętu (takiego jak Motorola 68040 i 88000, Intel i486, Sun SPARC, DEC Alpha, HP Precision oraz procesory MIPS R4000).

21.2 ■ Podstawy projektu

UNIX zaprojektowano jako system z podziałem czasu. Standardowy interfejs użytkownika (powłoka) jest prosty i może być zastąpiony przez inny, stosowanie do potrzeb. System plików jest wielopoziomowym drzewem, w którym użytkownicy mają prawo tworzyć własne podkatalogi. Pliki danych każdego użytkownika są po prostu ciągami bajtów.

Pliki dyskowe i urządzenia wejścia-wyjścia są traktowane możliwie jednolicie. Toteż osobliwości i szczegóły związane z urządzeniami są tak dalece, jak tylko jest to możliwe, zawarte w jądrze systemu, przy czym nawet w jądrze większość z nich zamknięto w modułach obsługi urządzeń.

UNIX umożliwia wieloprocesowość. Proces może łatwo tworzyć nowe procesy. Planowanie przydziału procesora jest prostym algorytmem priorytetowym. W systemie 4.3BSD zastosowano stronicowanie na żądanie jako mechanizm wspierający zarządzanie pamięcią i podejmowanie decyzji o przydiale procesora. Jeśli systemowiaczyna doskwierać nadmierne zastępowanie stron, to stosuje się algorytm wymiany.

Ponieważ system UNIX w zaraniu miał służyć wygodzie jednego programisty, Kena Thompsona, do którego dołączył potem drugi – Dennis Ritchie, więc jest to system wystarczająco mały, aby można go było zrozumieć. W doborze większości algorytmów kierowano się *prostotą*, a nie szybkością lub wyrafinowaniem. Zamiar polegał na uzyskaniu jądra i bibliotek dających mały zbiór właściwości, lecz wystarczająco mocny, aby zainteresowani mogli zbudować na tej podstawie bardziej złożony system, gdyby powstała taka konieczność. Przejrzysty projekt systemu UNIX zaowocował wieloma naśladownictwami i modyfikacjami.

Chociaż projektanci systemu UNIX mieli sporą wiedzę o innych systemach operacyjnych, jego implementacji nie poprzedziło opracowanie starannej wyartykulowanego projektu. Ta elastyczność okazała się jednym z kluczowych czynników w rozwoju systemu. Przyjęto jednak pewne założenia projektowe, chociaż nie zostały one wypowiedziane na początku.

UNIX został zaprojektowany przez programistów dla programistów. W związku z tym od początku był systemem interakcyjnym. Udogodnieniom do opracowywania programów nadawano zawsze wysoką rangę. Do udoskonalenia takich należy program *make* (można go używać do sprawdzania, które pliki ze zbioru plików źródłowych programu należy skompilować, a następnie

do powodowania ich komplikacji) oraz systemu nadzorowania kodu źródłowego (ang. *Source Code Control System* – SCCS), który zapewnia dostępność kolejnych wersji plików, bez konieczności przechowywania całych ich zawartości dla każdego kroku^{*}.

Większość kodu systemu operacyjnego UNIX jest napisana w języku C, który opracowano specjalnie w tym celu, ponieważ ani Thompson, ani Ritchie nie lubili programować w asemblerze. Unikanie języka asemblera było również konieczne ze względu na niepewność co do maszyny lub maszyn, na których UNIX miał pracować. Ułatwiało to wiele problemu przenoszenia oprogramowania UNIX z jednego systemu sprzętowego na inne.

Opracowywane systemy UNIX od samego początku były dostępne bezpośrednio w postaci źródłowej, a ich twórcy mieli za podstawę działania systemy nie całkiem ukończone. Ten schemat postępowania znacznie ułatwiał tropienie i usuwanie niedociągnięć, jak również odkrywanie nowych możliwości i ich implementowanie. Doprowadziło to także do istniejącego obecnie nadmiaru wariantów systemu UNIX. Jednak zalety przeważyły wady: jeśli coś było nie w porządku, to można było to naprawić na lokalnym stanowisku, bez konieczności czekania na następne wydanie systemu. Poprawki takie, a także nowe udogodnienia można było włączyć do późniejszej upowszechnianych wersji.

Ograniczone możliwości komputera PDP-11 (i wcześniejszych komputerów używanych przez UNIX) wymuszyły pewną elegancję. Podczas gdy inne systemy mają wyszukane algorytmy postępowania w warunkach patologicznych, system UNIX wykonuje po prostu kontrolowane załamanie, zwane *paniką* (ang. *panic*). Zamiast usiłować naprawiać takie sytuacje, UNIX próbuje im zapobiegać. Tam gdzie w innych systemach zastosowano by brutalne metody lub rozwijanie makrowywołań, w systemie UNIX musiano postąpić znacznie subtelniej lub przynajmniej prościej.

Te wczesne zalety systemu UNIX przysporzyły mu popularności, która z kolei – rodząc nowe zapotrzebowania – wystawiła zalety te na próbę. UNIX używano do zadań takich, jak organizacja pracy w sieciach, grafika i operacje w czasie rzeczywistym. Zastosowania te nie zawsze pasowały do oryginalnego, ukierunkowanego tekstowo modelu systemu UNIX. Wykonano więc kolejne zmiany pewnych właściwości wewnętrznych i podawano nowe interfejsy programowe. Te nowe udogodnienia i inne interfejsy – zwłaszcza do pracy w systemie okien – wymagały wielkich ilości kodu, co radykalnie zwiększyło rozmiary systemu. Na przykład zarówno praca sieciowa, jak

* Jako ciekawostkę historyczną odnotujmy, że już w połowie lat siedemdziesiątych możliwość taka była zapewniona w systemie operacyjnym GEORGE 3, brytyjskiej firmy ICI. Pozwalało na to programowany edytor systemu GEORGE 3. – Przyp. tłum.

i systemy okien podwoiły rozmiar systemu. Z kolei ta sytuacja ukazała się w systemie UNIX. Gdy tylko w przemyśle pojawiło się nowe rozwiązanie, system UNIX z reguły wchłaniał je, wciąż pozostając sobą.

21.3 ■ Interfejs programisty

Podobnie jak większość systemów operacyjnych, UNIX składa się z dwóch oddzielnych części: jądra i programów systemowych. Na system operacyjny UNIX można patrzeć jak na system złożony z warstw, jak na rys. 21.2. Wszystko to, co znajduje się poniżej interfejsu funkcji systemowych^{*} i powyżej sprzętu fizycznego, tworzy jądro. W jądrze jest zrealizowany system plików, planowanie przydziału procesora, zarządzanie pamięcią i inne funkcje systemu operacyjnego, do których dostęp odbywa się za pomocą wywołań systemowych. W realizacji swoich pozytycznych działań, takich jak komplikowanie lub manipulowanie plikami, programy systemowe wspierają się odwołaniami do procedur jądra.

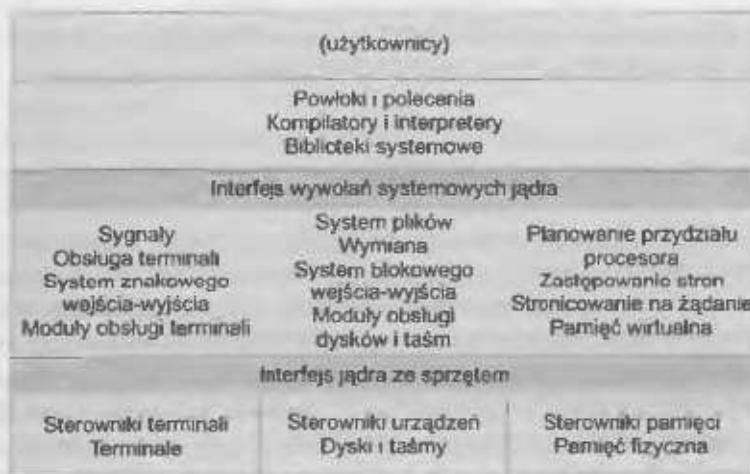
Wywołania systemowe określają *interfejs programisty* systemu UNIX. Zbiór ogólnie dostępnych programów systemowych definiuje *interfejs użytkownika*. Oba interfejsy definiują kontekst realizowany przez jądro.

Odwołania do systemu w wersji 4.2BSD na komputerze VAX są wykonywane przez przejście do komórki 40 wektora przerwań komputera VAX. Parametry są przekazywane do jądra na stosie sprzętowym. Jądro zwraca wartości funkcji w rejestrach R0 i R1. Rejestr R0 może przekazywać kod błędu. Wartość bitu przeniesienia odróżnia normalny powrót z jądra od powrotu z błędem.

Na szczeblu z tego poziomu szczegółowości rzadko kiedy zdaje sobie sprawę programista pracujący w systemie UNIX. Większość programów systemowych napisano w języku C, a dokumentacja pt. *UNIX Programmer's Manual* zawiera wszystkie funkcje systemowe w postaci funkcji języka C. Program systemowy napisany w języku C dla wersji 4.3BSD na komputerze VAX może być na ogół przeniesiony do innego systemu 4.3BSD i po prostu skompilowany, nawet jeśli te dwa systemy różnią się zasadniczo między sobą. Szczegóły odwołań do systemu są znane tylko kompilatorowi. Ta właściwość jest podstawowym źródłem przenośności programów systemu UNIX.

Odwołania do systemu UNIX można z grubsza podzielić na trzy kategorie: manipulowanie plikami, sterowanie procesami oraz manipulowanie informacją. W rozdziale 3 zamieściliśmy jeszcze czwartą kategorię – manipu-

* Inaczej: interfejs wywołań systemowych lub odwołań do systemu (ang. *system-call interface*) – Przyp. tłum.



Rys. 21.2 Warstwowa struktura systemu 4.3BSD

lowanie urządzeniami. Ponieważ jednak urządzenia w systemie UNIX są traktowane jako (specjalne) pliki, więc do ich obsługi służą te same funkcje systemowe, co do obsługi plików (choć istnieje specjalna funkcja systemowa do ustawiania parametrów urządzeń).

21.3.1 Manipulowanie plikami

Plik w systemie UNIX jest ciągiem bajtów. Różne programy odnoszą się do plików jako do rozmaitych struktur, lecz jądro nie narzuca plikom żadnej struktury. Na przykład konwencją dla plików tekstowych są wiersze znaków ASCII rozdzielane pojedynczym znakiem nowego wiersza (któremu w kodzie ASCII odpowiada znak LF – ang. *linefeed*), lecz jądro nie wie nie o takiej konwencji.

Pliki są zorganizowane w drzewiastą strukturę *katalogów*. Katalogi też są plikami, które zawierają informacje o tym, jak odnaleźć inne pliki. Nazwą ścieżki do pliku jest napis, który identyfikuje plik przez określenie drogi prowadzącej do tego pliku w strukturze katalogowej. Pod względem składni nazwę ścieżki tworzą poszczególne nazwy plików pooddzielane ukośnymi kreskami. Na przykład w */usr/local/font* pierwsza ukośna kreska symbolizuje korzeń drzewa katalogów, zwany *katalogiem głównym* (ang. *root*)*. Następny element – *usr* – oznacza podkatalog korzenia, *local* oznacza podkatalog katalogu *usr*, a *font* jest nazwą pliku lub katalogu w katalogu *local*. Na podsta-

* Dosię: korzeniowym. – Przyp. tłum.

wie składni nazwy ścieżki nie można wywnioskować, czy nazwa *font* odnosi się do pliku zwykłego czy do katalogu.

W systemie plików UNIX używa się zarówno *bezwzględnych*, jak i *względnych* nazw ścieżek. Nazwy bezwzględne rozpoczynają się od korzenia systemu plików i są rozpoznawane przez występowanie ukośnej kreski na ich początku. Napis */usr/local/font* jest bezwzględną nazwą ścieżki. Względne nazwy ścieżek rozpoczynają się w *katalogu bieżącym*, który jest atrybutem procesu uzyskującego dostęp do nazwy ścieżki. Zatem nazwa *local/font* wskazuje na plik lub katalog nazwany *font* w katalogu *local* położonym w katalogu bieżącym, którym może być – lecz nie musi – katalog */usr*.

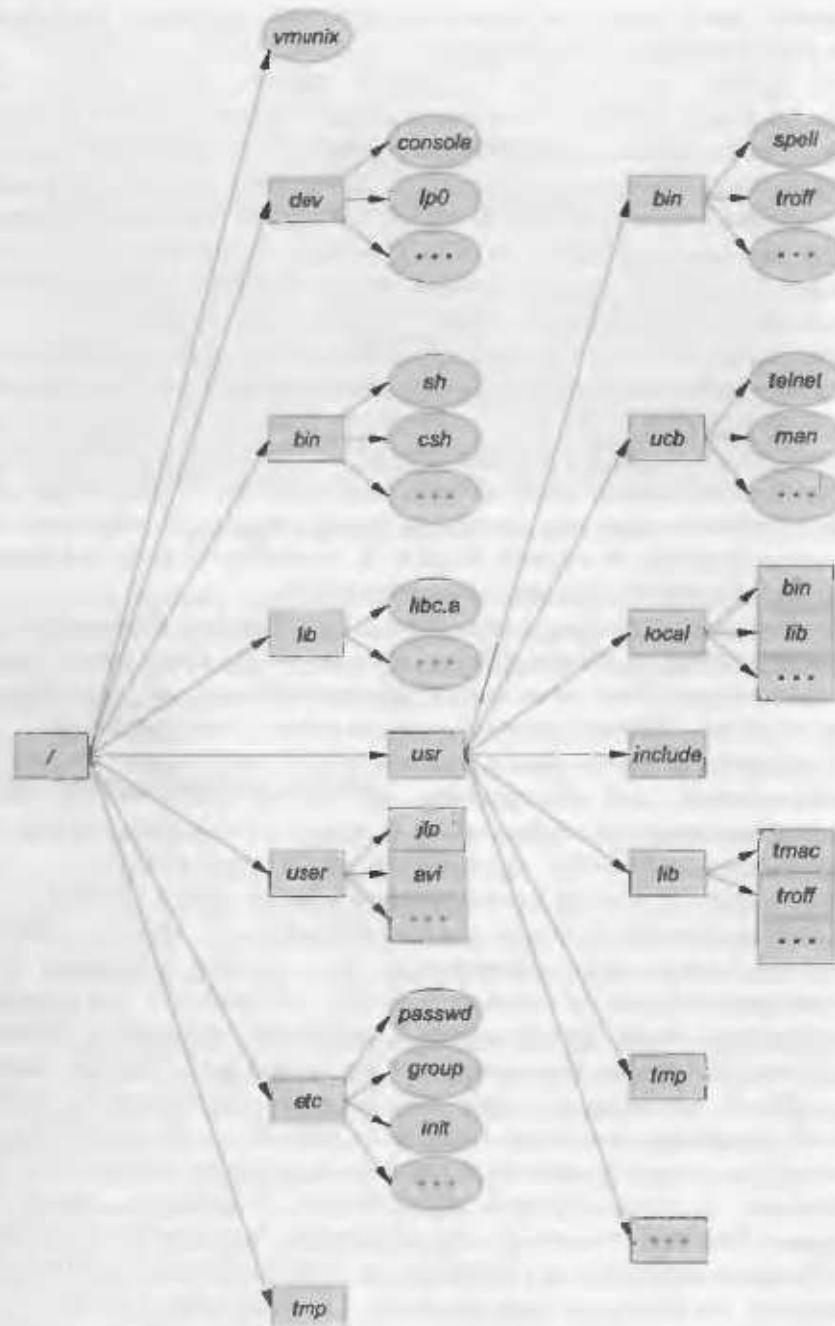
Plik może być znany pod więcej niż jedną nazwą w jednym lub większej liczbie katalogów. Takie zwiększone nazwy są nazywane *dowiązaniami*, przy czym wszystkie dowiązania są traktowane przez system operacyjny jednakowo. System 4.3BSD dopuszcza również *dowiązania symboliczne*, które są plikami zawierającymi nazwy ścieżek innych plików. Te dwa rodzaje dowiązań są również znane jako *dowiązania twarda i miękkie*. W odróżnieniu od dowiązania twardego, dowiązanie miękkie (tj. symboliczne) może wskazywać na katalog i wykraczać poza granice systemu plików.

Występująca w katalogu nazwa pliku „...” jest twardym dowiązaniem do tego samego katalogu. Nazwa pliku „...” oznacza twarda dowiązanie do katalogu nadziedziczonego. Jeżeli zatem nazwę katalogu bieżącego jest */user/jlp/programs*, to nazwa *./bin/wdf* jest równoznaczna nazwie */user/jlp/bin/wdf*.

Urządzenia sprzętowe mają w systemie plików swoje nazwy. Te *specjalne pliki urządzeń*, czyli *pliki specjalne*, są znane jądrou jako interfejsy urządzeń, co nie przeszkadza użytkownikom uzyskiwać do nich dostęp za pomocą prawie takich samych funkcji systemowych, jak do innych plików.

Na rysunku 21.3 widać typowy uniksowy system plików. Korzeń (/) na ogół zawiera niewiele katalogów, a wśród nich takie, jak: *vmunix* – binarny, rozruchowy obraz systemu operacyjnego; */dev* – katalog zawierający pliki specjalne urządzeń, jak */dev/console*, */dev/lp0*, */dev/mt0* itd.; */bin* – katalog przechowujący obrazy binarne ważnych uniksowych programów systemowych. Inne pliki binarne mogą znajdować się w katalogach: */usr/bin* (programy aplikacji systemowych, takie jak programy formatowania tekstu), */usr/ucb* (programy systemowe napisane w Berkeley – nie w AT&T) lub */usr/local/bin* (programy systemowe napisane na lokalnym stanowisku). Pliki biblioteczne, w rodzaju bibliotek podprogramów do języka C, Pascala lub Fortranu, są trzymane w katalogu */lib* (lub */usr/lib*, względnie */usr/local/lib*).

Pliki poszczególnych użytkowników są przechowywane w oddzielnych katalogach, zazwyczaj w katalogu */user*. Tak więc katalog użytkownika *carol* będzie na ogół identyfikowany z nazwą */user/carol*. W dużych systemach katalogi mogą być dalej pogrupowane w celu ułatwienia administro-



Rys. 21.3 Typowa struktura katalogów systemu UNIX

wania, tworząc strukturę plików z takimi elementami, jak `/user/proflavi` i `/user/staff/carol`. Pliki i programy służące do zarządzania, jak na przykład plik haseł, są przechowywane w katalogu `/etc`. Pomocnicze pliki tymczasowe trafiają do katalogu `/tmp`, z którego są z reguły usuwane podczas rozruchu systemu, lub do katalogu `/usr/tmp`.

Struktura każdego z tych katalogów może się dalej komplikować. Na przykład tablice opisujące kroje pisma używane przez program formatujący `troff` w systemie składu Mergenthaler 202 znajdują się w katalogu `/usr/lib/troff/dev202`. Wszystkie konwencje dotyczące umiejscowienia poszczególnych plików i katalogów są określone przez programistów i ich programy. Jądro systemu operacyjnego wymaga do swojego działania obecności tylko programu `/etc/init`, używanego do zapoczątkowywania procesów obsługi terminali.

Do wykonywania podstawowych operacji na plikach służą funkcje systemowe: `creat`, `open`, `read`, `write`, `close`, `unlink` i `trunc`. Funkcja `creat`, mając daną nazwę ścieżki, tworzy (pusty) plik (lub obciną zawartość istniejącego). Istniejący plik jest otwierany za pomocą funkcji systemowej `open`, która przyjmuje na wejściu nazwę ścieżki i tryb pracy (taki jak czytanie, pisanie lub czytanie i pisanie), a zwraca małą liczbę całkowitą, zwaną *deskryptorem pliku*. Deskryptor pliku może być potem przekazany do funkcji systemowych `read` lub `write` (wraz z adresem bufora i liczbą bajtów do przesłania) w celu wykonania przesłania danych do lub z pliku. Zamknięcie pliku następuje wtedy, kiedy jego deskryptor zostanie przekazany do funkcji `close`. Wywołanie `trunc` redukuje długość pliku do zera.

Deskryptor pliku jest indeksem do małej tablicy plików otwartych w danym procesie. W typowych programach deskryptory zaczynają się od 0 i rzadko kiedy przekraczają wartość 6 lub 7, w zależności od maksymalnej liczby jednocześnie otwartych plików.

Każda operacja `read` lub `write` uaktualnia bieżącą odległość w pliku, przechowywaną w elemencie tablicy plików. Odległość ta służy do określania miejsca w pliku dla następnych operacji `read` lub `write`. Funkcja systemowa `lseek` pozwala jawnie określać to miejsce. Za jej pomocą można też tworzyć luźne pliki (tj. takie, w których wnętrzu występują „dziury”). Wywołyń systemowych `dup` i `dup2` można użyć do utworzenia nowego deskryptora pliku, będącego kopią istniejącego deskryptora. Można to również wykonać za pomocą funkcji systemowej `fcreat`, która ponadto sprawdza lub ustawia różne parametry otwartego pliku. Na przykład może ona spowodować, że wszystkie następne operacje pisania do otwartego pliku będą powodowały dopisywanie informacji na jego końcu. Istnieje także dodatkowa funkcja systemowa `ioctl`, umożliwiająca manipulowanie parametrami urządzeń; korzystając z niej można na przykład określić szybkość przesyłania portu szeregowego lub zwinąć taśmę.

Z pomocą funkcji systemowej `stat` można otrzymać informacje o pliku (jego rozmiar, tryby ochrony, dane o właścicielu itp.). Kilka funkcji systemowych pozwala zmieniać niektóre z tych informacji: `rename` (zmiana nazwy pliku), `chmod` (zmiana trybu ochrony) i `chown` (zmiana właściciela i grupy). Wiele z tych funkcji ma warianty pozwalające posługiwać się deskryptorami plików zamiast ich nazwami. Funkcja systemowa `link` tworzy twardy dowiązanie do istniejącego pliku, dodając plikowi nową nazwę. Dowiązanie usuwa się za pomocą funkcji `unlink`. Jeśli było to ostatnie dowiązanie, to plik jest usuwany. Funkcja systemowa `symlink` tworzy dowiązanie symboliczne.

Katalogi są tworzone za pomocą funkcji systemowej `mkdir`, a usuwane – za pomocą `rmdir`. Zmiany bieżącego katalogu dokonuje się przy użyciu operacji `cd`.

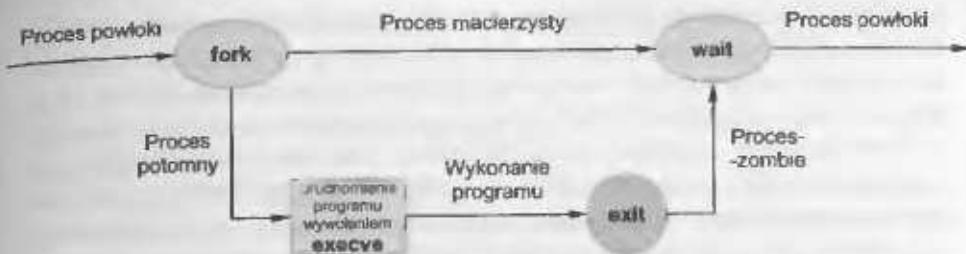
Choć jest możliwe stosowanie w odniesieniu do katalogów standardowych operacji plikowych, nie jest to zalecane, gdyż katalogi mają wewnętrzną strukturę, która musi być zachowana. W zamian można skorzystać z odrętnego zbioru funkcji systemowych służących do otwierania katalogu, przeglądania kolejnych wpisów plików w obrębie katalogu, zamknięcia katalogu i wykonywania innych działań. Są to funkcje: `opendir`, `readdir`, `closedir` i inne.

21.3.2 Nadzorowanie procesów

Procesem jest wykonujący się program. Procesy są rozróżniane za pomocą identyfikatorów procesów, które są liczbami całkowitymi. Nowy proces tworzy się za pomocą funkcji systemowej `fork`. Zawiera on kopię przestrzeni adresowej procesu pierwotnego (ten sam program i te same zmienne, z tymi samymi wartościami). Oba procesy (macierzysty i potomny) kontynuują działanie od instrukcji występującej po `fork`, z jedną różnicą – funkcja `fork` przekazuje zero do nowego procesu (do potomka), natomiast do procesu macierzystego przekazuje (niezerowy) identyfikator procesu potomnego.

Na ogół po rozwidleniu jeden z dwóch procesów używa funkcji systemowej `execve`, aby zastąpić przestrzeń swojej pamięci wirtualnej nowym programem. Funkcja `execve` wprowadza plik binarny do pamięci (niszcząc obraz pamięci programu zawierającego wywołanie funkcji `execve` i rozpoczyna jego wykonanie).

Proces może zakończyć działanie za pomocą funkcji systemowej `exit`, a jego proces macierzysty może czekać na to zdarzenie, używszy wywołania funkcji `wait`. Jeśli proces potomny ulegnie załamaniu, to system symuluje wywołanie funkcji `exit`. Funkcja systemowa `wait` dostarcza identyfikatora zakończonego procesu potomnego, tak że proces macierzysty może określić, który z jego, być może wielu, potomków zakończył działanie. Druga funkcja systemowa – `wait3` – jest podobna do `wait`, lecz ponadto umożliwia proceso-



Rys. 21.4 Powłoka w celu wykonania programu wywołuje podproses

wi macierzystemu zebranie statystyki o przebiegu działań potomka. W czasie między zakończeniem działania procesu potomnego a zakończeniem przez proces macierzysty którejś z systemowych operacji czekania, potomek jest *nieżywy* (ang. *defunct*). Nieżywy proces nie może zrobić nic prócz swojego zakończenia, aby przodek mógł zebrać informację o jego stanie. Jeśli przodek nieżywego procesu zakończy działanie wcześniej niż jego potomek, to nieżywy proces zostaje odziedziczony przez proces początkowy* (który zaczyna na niego oczekwać), stając się „procesem zombie” (ang. *zombie*). Typowe zastosowanie tych możliwości jest pokazane na rys. 21.4.

Najprostszą formą komunikacji między procesami są *potoki* (ang. *pipes*), które mogą zostać utworzone przed operacją fork, i których punkty końcowe zestawia się ze sobą między wykonaniem funkcji fork a wykonaniem funkcji execve. Potok jest w istocie kolejką bajtów organizowaną między dwoma procesami. Dostęp do potoku odbywa się za pomocą deskryptora potoku, tak jak do zwykłego pliku. Jeden proces pisze do potoku, a drugi z niego czyta. Pierwotny rozmiar potoku został w systemie ustalony. W wersji 4.3BSD potoki zrealizowano powyżej systemu gniazd, którego bufora mają zmienną długość. Czytanie z pustego potoku lub pisanie do potoku wypełnionego powoduje wstrzymywanie procesu do czasu, aż stan potoku ulegnie zmianie. Umieszczenie potoku między procesem macierzystym a jego potomkiem wymaga specjalnych ustawień (gdyż tylko jeden proces czyta i jeden pisze).

Wszystkie procesy użytkowników są potomkami jednego pierwotnego procesu – procesu *init*. Port każdego terminalu gotowego do prowadzenia konwersacji ma aktywny proces *getty*, zainicjowany dla niego przez proces *init*. Proces *getty* określa początkowe parametry linii terminalu i oczekuje na nazwę rejestracyjną (ang. *login name*) użytkownika, którą przekazuje za pomocą operacji execve jako argument do procesu *login*. Proces *login* pobiera hasło użytkownika, szyfruje je i porównuje wynik z zaszyfrowanym napisem z pliku */etc/passwd*. Jeśli porównanie wypada pomyślnie, to zezwala się użytko-

* Proces, od którego system rozpoczyna działanie (ang. *init*). – Przyp. tłum.

kownikowi na wejście do systemu. Proces *login* rozpoczyna wykonywanie procesu *shell*, czyli interpretatora poleceń, ustawiając numeryczny identyfikator użytkownika procesu według danych rejestracyjnych użytkownika. (Rodzaj powłoki i identyfikator użytkownika są odnajdywane w pliku */etc/passwd* za pomocą nazwy rejestracyjnej użytkownika). Przez resztę sesji użytkownik z reguły komunikuje się z tą właśnie powłoką. W celu wykonywania poleceń przekazywanych przez użytkownika powłoka sama rozwidla się na podprocesy.

Jądro systemu posługuje się identyfikatorem użytkownika przy określaniu praw użytkownika do pewnych funkcji systemowych, zwłaszcza tych, które dotyczą dostępów do plików. Istnieje także identyfikator grupy (ang. *group identifier*), używany do kontroli podobnych przywilejów zbioru użytkowników. W systemie 4.3BSD proces może należeć do kilku grup jednocześnie. Na podstawie plików */etc/passwd* i */etc/group* proces rejestracyjny *login* umieszcza powłokę we wszystkich grupach, w których użytkownik ma prawo uczestniczenia.

W rzeczywistości jądro posługuje się dwoma identyfikatorami użytkownika. Skuteczny identyfikator użytkownika (ang. *effective user identifier*) jest identyfikatorem używanym do określania praw dostępu do pliku. Jeśli plik z programem, który ma być załadowany za pomocą funkcji *execve*, ma ustalony bit *setuid* w swoim i-węźle, to obowiązującym identyfikatorem użytkownika w danym procesie ustanawia się identyfikator użytkownika będącego właścicielem pliku, podczas gdy rzeczywisty identyfikator użytkownika (ang. *real user identifier*) pozostaje bez zmiany. Pozwala to pewnym procesom korzystać z większych niż zazwyczaj przywilejów, choć są wciąż wykonywane przez zwykłych użytkowników. Pomyśl wprowadzenia bitu *setuid* został opatentowany przez Dennisa Ritchiego (U.S. Patent 4135240) i jest jedną z cech wyróżniających system UNIX. Dla grup użytkowników istnieje podobny bit *setgid*. Proces może zbadać swoje identyfikatory użytkownika – rzeczywisty i skuteczny – odpowiednio za pomocą wywołań *getuid* i *geteuid*. Wywołania *getgid* i *getegid* umożliwiają odpowiednio zbadanie rzeczywistego i skutecznego identyfikatora dla grupy procesów. Pozostałe grupy procesów można odnaleźć za pomocą funkcji systemowej *getgroups*.

21.3.3 Sygnały

Sygnały są udogodnieniem pozwalającym obsługiwać sytuacje wyjątkowe na podobieństwo przerwań programowych. Jest 20 różnych sygnałów*, z których każdy odpowiada innej sytuacji. Sygnał może być wygenerowany

* Proces ten określamy mianem powłoki. – Przyp. tłum.

** W wersji 4.4BSD istnieje 31 różnych sygnałów. – Przyp. tłum.

przez przerwanie z klawiatury, błąd w procesie (w rodzaju zlego odniesienia do pamięci) lub za pomocą pewnej liczby zdarzeń asynchronicznych (np. sygnały od czasomierzy lub sygnały sterujące zadaniem pochodzące z powłoki). Prawie każdy sygnał może być też wytworzony za pomocą funkcji systemowej `kill`.

Sygnal *przerwania* (ang. *interrupt*), `SIGINT`, jest używany do zatrzymywania wykonywania polecenia przed jego zakończeniem. Zwykle wytwarzany znak `'C` (ASCII 3). W wersji 4.2BSD ważne znaki klawiatury są zdefiniowane w tablicy dla każdego terminalu i można je łatwo przeddefiniować. Sygnal *rezygnacji* (ang. *quit*), `SIGQUIT`, jest zwykle produkowany za pomocą znaku `\`` (ASCII 28). Sygnal rezygnacji zatrzymuje wykonywanie bieżącego programu i składuje obraz jego pamięci w bieżącym katalogu, w pliku o nazwie `core`. Pliku `core` mogą używać programy diagnostyczne. Wysłanie sygnału `SIGILL` jest powodowane przez użycie nielegalnego rozkazu, a `SIGSEGV` – przez usiłowanie zaadresowania pamięci poza dozwolonym obszarem pamięci wirtualnej procesu.

Mozna spowodować pomijanie większości sygnałów (aby nie wywoływały żadnych efektów) lub wyznaczyć procedurę w procesie użytkownika (procedurę obsługi sygnału), która będzie wywoływana po wystąpieniu sygnału. Procedura obsługi przechwyconego sygnału może zrobić bezpiecznie jedną z dwóch rzeczy: wywołać funkcję systemową `exit` lub zmodyfikować zmienną globalną. Jest jeden sygnał (sygnał `kill` – o numerze 9), `SIGKILL`, który nie może zostać pominięty ani przechwycony przez procedurę obsługi sygnału. Sygnał `SIGKILL` jest używany na przykład do likwidowania procesu, który wymknął się spod kontroli i ignoruje inne sygnały, takie jak `SIGINT` lub `SIGQUIT`.

Sygnały mogą ginąć. Jeśli nowy sygnał tego samego rodzaju zostanie wysłany, zanim poprzedni sygnał zostanie przyjęty przez proces, do którego został skierowany, to pierwszy sygnał będzie zasłonięty przez drugi i tylko ostatni sygnał będzie widoczny dla procesu. Innymi słowy, wywołanie procedury obsługi sygnału informuje proces, że sygnał wystąpił co najmniej jednorazowo. Nie ma również względnych priorytetów między sygnałami w systemie UNIX. Jeśli dwa różne sygnały zostaną wysłane do tego samego procesu w tym samym czasie, to nie jest określone, który z nich pierwszy dotrze do procesu.

Pierwotnie zamierzano stosować sygnały do obsługi zdarzeń wyjątkowych. Jednak, podobnie jak z używaniem większości innych funkcji systemu UNIX, zastosowanie sygnałów systematycznie się rozszerzało. W systemie 4.1BSD wprowadzono sterowanie zadaniami, w którym sygnały są używane do rozpoczętania i zatrzymywania podprocesów na życzenie. Rozwiążanie to umożliwiającej jednej powloce sterowanie wieloma proce-

sami – ich rozpoczęwanie, zatrzymywanie i przenoszenie na plan drugi wewnątrz życzeń użytkownika. W wersji 4.3BSD dodano sygnał SIGWINCH, wynaleziony przez firmę Sun Microsystems, do informowania procesu, że okno, w którym odbywa się wyprowadzanie wyników, zmieniło rozmiar. Sygnały są także używane do dostarczania pilnych danych z połączeń sieciowych.

Użytkownicy potrzebowali również bardziej niezawodnych sygnałów, bez błędów powodowanych szkodliwą rywalizacją, występującą w starej implementacji sygnałów. Toteż wersję 4.2BSD systemu wyposażono już w niezawodne i oddzielnie zaimplementowane możliwości sygnalizowania, wolne od rywalizacji. Poszczególne sygnały można blokować na czas wykonywania sekcji krytycznych, a nowe wywołanie systemowe pozwala na usypanie procesu do czasu przerwania. Możliwości te są podobne do działania przerwań sprzętowych i są obecnie częścią standardu POSIX.

21.3.4 Grupy procesów

Przy wykonywaniu wspólnych zadań często współpracują grupy powiązanych ze sobą procesów. Procesy mogą na przykład tworzyć potoki i komunikować się za ich pośrednictwem. Zbiór takich procesów jest nazywany *grupą procesów* lub *zadaniem*. Jest możliwe wysyłanie sygnałów do wszystkich procesów w grupie. Proces zwykle dziedziczy przynależność do grupy procesów po swoim przedku, ale wywołanie systemowe `setpgrp` pozwala mu też na zmianę grupy.

Grupy procesów są używane przez powłokę C do nadzorowania pracy wielu zadań. W danym czasie z terminalowych operacji wejścia-wyjścia może korzystać tylko jedna grupa procesów. Zadanie *pierwszoplanowe* angażuje uwagę użytkownika terminalu, natomiast wszystkie inne, nieprzyłączone zadania (zadania drugoplanowe) wykonują swoje działania bez konwersacji z użytkownikiem. Dostęp do terminalu jest nadzorowany przez grupowe sygnały procesów. Każde zadanie ma *terminal kontrolny* – również odziedziczyły po przedku. Jeśli grupa procesów terminalu kontrolnego pasuje do grupy danego procesu, to ten proces działa jako pierwszoplanowy i może wykonywać operacje wejścia-wyjścia. Jeśli to samo usiłuje zrobić proces z grupy nie dającej się dopasować (drugoplanowy), to do jego grupy procesów jest posyłany sygnał SIGTTIN lub SIGTTOU*. Sygnał taki zazwyczaj powoduje zamrożenie grupy procesów do czasu, aż zostanie ona wybrana na plan pierwszy przez użytkownika. Wówczas otrzyma ona sygnał SIGCONT,

* Sygnały te są generowane w przypadku próby (odpowiednio) czytania lub pisania z użyciem terminalu przez proces drugoplanowy. – Przyp. tłum.

wskazujący, że proces może wykonywać operacje wejścia-wyjścia. Podobnie w celu zamrożenia grupy procesów pierwszoplanowych można wysłać do niej sygnał SIGSTOP.

21.3.5 Dostęp do informacji systemowych

Istnieją funkcje systemowe do ustawiania i pobierania w mikrosekundach wartości zarówno czasomierza (`getitimer`, `setitimer`), jak i bieżącego czasu (`gettimeofday`, `settimeofday`). Ponadto procesy mogą pytać o swoje identyfikatory procesowe (`getpid`), identyfikatory ich grup (`getgid`), nazwę maszyny, na której pracują (`gethostname`), i o wiele innych danych.

21.3.6 Procedury biblioteczne

Interfejs odwołań do systemu UNIX jest wspomagany i rozszerzany przez wielki zbiór procedur bibliotecznych i plików nagłówkowych*. Pliki nagłówkowe zawierają definicje złożonych struktur danych używanych w funkcjach systemowych. Ponadto wielka biblioteka funkcji tworzy dodatkowe zaplecze programowe.

Na przykład funkcje uniksowego systemu wejścia-wyjścia umożliwiają czytanie i pisanie bloków bajtów. W niektórych zastosowaniach może być potrzebne czytanie i pisanie jednorazowo tylko jednego bajtu. Chociaż można by czytać lub pisać po jednym bajcie, odwołując się za każdym razem do systemu, powodowałoby to bardzo duży nakład dodatkowych operacji. Zamiast tego zbiór standardowych procedur bibliotecznych (standardowy pakiet wejścia-wyjścia) dostarcza za pośrednictwem pliku nagłówkowego `<stdio.h>` innego interfejsu, w którym czyta się i zapisuje jednorazowo po kilka tysięcy bajtów, używając lokalnych buforów, i wykonuje przesłania między nimi (w pamięci użytkownika) wówczas, gdy są potrzebne operacje wejścia-wyjścia. Standardowy pakiet wejścia-wyjścia realizuje również formatowane wejście-wyjście.

Dodatkowe oprogramowanie biblioteczne obejmuje funkcje matematyczne, dostęp do sieci, konwersje danych itd. Jądro systemu 4.3BSD zawiera ponad 150 funkcji systemowych; biblioteka programowa języka C ma ponad 300 funkcji**. Choć funkcje biblioteczne sprawdzają się w końcu, tam gdzie to jest niezbędne, do funkcji systemowych (na przykład biblioteczny podprogram `getchar` daje w wyniku wywołanie funkcji systemowej `read`, jeśli bufor

* Mowa o plikach z deklaracjami w języku C. – Przyp. tłum.

** Uwzględniając interfejsy graficzne i lokalne osobliwości, biblioteki języka C są jeszcze większe. – Przyp. tłum.

jest pusty), to jednak programista może w zasadzie nie odróżniać podstawowego zbioru wywołań jądra systemu od dodatkowych funkcji zawartych w bibliotece.

21.4 ■ Interfejs użytkownika

Zarówno programista, jak i użytkownik systemu UNIX mają do czynienia głównie ze zbiorem napisanych i gotowych do wykonania programów systemowych. Programy te wykonują niezbędne odwołania do systemu w celu realizowania swoich zadań, lecz użytkownik nie musi uświadamiać sobie tego, ponieważ odwołania te występują wewnętrz programów.

Popularne programy systemowe można zaliczyć do kilku kategorii. Większość z nich jest przeznaczona do pracy z plikami lub katalogami. Na przykład systemowymi programami do manipulowania katalogami są: *mkdir* służący do tworzenia nowego katalogu, *rmdir* pozwalający usuwać katalogi, *cd* używany do zmieniania katalogu bieżącego oraz *pwd* do drukowania bezwzględnej nazwy ścieżki katalogu bieżącego (roboczego).

Program *ls* wyprowadza wykaz nazw plików katalogu bieżącego. Dowolna z jego 18 opcji może spowodować dodatkowo wypisanie odpowiednich cech plików. Na przykład opcja *-l* wyraża życzenie wyprowadzenia długiego wykazu, zawierającego nazwy plików, właścicieli, tryby ochrony, datę i czas utworzenia pliku oraz jego rozmiar. Program *cp* tworzy nową kopię istniejącego pliku. Program *mv* przemieszcza plik z jednego miejsca do innego w drzewie katalogowym; w większości przypadków powoduje to tylko przenianowanie pliku, jednak w razie konieczności plik jest kopowany do nowego miejsca, a stara kopia usuwana. Plik jest usuwany za pomocą programu *rm* (który wykonuje funkcję systemową *unlink*).

Aby wyświetlić plik na terminalu, użytkownik może wykonać program *cat*. Program *cat* pobiera listę plików, łączy ich zawartość, po czym kopiuje wynik na standardowym wyjściu, którym jest zazwyczaj terminal. Na szybkim monitorze (w technologii CRT) tempo wyświetlania mogłoby być oczywiście za duże do czytania. Program *more* wyświetla plik w porcjach mieszczących się na ekranie, czekając za każdym razem przed wyświetleniem następnej porcji na naciśnięcie przez użytkownika jakiegoś klawisz na klawiaturze. Program *head* wyświetla tylko kilka pierwszych wierszy pliku, a program *tail* pokazuje kilka ostatnich wierszy.

Są to podstawowe programy systemowe, szeroko używane w systemie UNIX. Ponadto jest kilka edytorów (*ed*, *sed*, *emacs*, *vi* itd.), kompilatorów (C, Pascal, Fortran itp.) oraz programów formatowania tekstów (*troff*, *TEX*, *scribe* itp.). Istnieją także programy do sortowania (*sort*) i porównywania

plików (*cmp*, *diff*), przeszukiwania ich według wzorców (*grep*, *awk*), przesyłania poczty do innych użytkowników (*mail*) i wykonywania wielu innych czynności.

21.4.1 Powłoki i polecenia

Zarówno programy napisane przez użytkownika, jak i programy systemowe są zwykle wykonywane przez interpreter polecen. Jak każdy proces, interpreter polecen w systemie UNIX jest procesem użytkownika. Nazywa się go *powłoką* (ang. *shell*), ponieważ otacza jądro systemu operacyjnego. Użytkownicy mogą pisać własne powłoki i rzeczywiście – w powszechnym użyciu znajduje się kilka takich programów. Prawdopodobnie najszerzej używana – lub przynajmniej najłatwiej dostępna – jest *powłoka Bourne'a*, której autorem jest Steve Bourne. *Powłoka C*, będąca w większej części dziełem Billa Joya, założyciela firmy Sun Microsystems, cieszy się największą popularnością w systemach BSD. *Powłoka Korna*, autorstwa Dave'a Korna, spopularyzowała się dzięki temu, że łączy cechy powłoki Bourne'a i powłoki C.

Popularne powłoki mają w dużym stopniu wspólną składnię języka polecen. UNIX jest zwykle użytkowany konwersacyjnie. Powłoka objawia swoją gotowość do przyjęcia następnego polecenia przez wyświetlenie znaku zięły, po którym użytkownik wpisuje polecenie w jednym wierszu. Na przykład w wierszu

% *ls -l*

znak procentu jest typowym znakiem zięły powłoki C, a tekst *ls -l* (napisany przez użytkownika) oznacza polecenie wyprowadzenia katalogu (w wersji pełnej). Polecenia mogą zawierać argumenty, które użytkownik pisze po nazwie polecenia w tym samym wierszu, oddzielając je odstępami (spacjami lub znakami tabulacji).

Choć istnieje niewielka liczba polecen wbudowanych w powłokę (np. polecenie *cd*), to jednak typowe polecenie jest wykonywalnym, binarnym plikiem wynikowym. Powłoka utrzymuje wykaz kilku katalogów zwany *ściezką wyszukiwania*. Dla każdego polecenia jest przeglądany każdy z katalogów umieszczonych na ścieżce wyszukiwania w celu odnalezienia pliku o takiej samej nazwie. Jeśli plik zostanie znaleziony, to będzie załadowany do pamięci i wykonany. Ścieżka wyszukiwania może być określona przez użytkownika. Katalogi */bin* i */usr/bin* prawie zawsze znajdują się na ścieżce wyszukiwania, a typowa ścieżka wyszukiwania w systemie BSD może wyglądać następująco:

(*/home/proflavi/bin /usr/local/bin /usr/ucb /bin /usr/bin*)

Plik wynikowy polecenia `ls` ma nazwę `/bin/ls`, a sama powłoka nazywa się `/bin/sh` (powłoka Bourne'a) lub `/bin/csh` (powłoka C).

Wykonanie polecenia odbywa się za pomocą funkcji systemowej `fork`, po której występuje operacja `execve` odniesiona do pliku z poleceniem. Powłoka wykonuje wtedy zazwyczaj operację `wait`, aby zawiesić własne działanie do czasu zakończenia polecenia (rys. 21.4). Za pomocą prostego oznaczenia w składni polecenia (znak & na końcu wiersza z poleceniem) można wskazać, żeby powłoka nie czekała na zakończenie polecenia. Polecenie wykonywane w ten sposób, że powłoka interpretuje następne polecenia, nazywa się *drugoplanowym* albo mówi się, że polecenie pracuje w tle. Procesy, na których wykonanie powłoka musi czekać, nazywają się *pierwszoplanowymi*.

Jak już wspomnieliśmy, powłoka C w systemach 4.3BSD ma udogodnienia nazywane *sterowaniem zadaniami* (ang. *job control*), częściowo zaimplementowane w jądrze. Sterowanie zadaniami umożliwia przemieszczanie procesów między pierwszym a drugim planem. Procesy mogą być wstrzymywane i wznowiane w zależności od różnych warunków, takich jak zapotrzebowanie zadania w tle na dane z terminalu użytkownika. Schemat ten umożliwia wykonywanie większości zabiegów sterujących procesami zawartymi w interfejsach okien lub warstw, bez potrzeby użycia specjalnego sprzętu. Sterowanie zadaniami jest także bardzo wygodne w systemach okien, takich jak X Window System, opracowany w MIT. Każde okno jest traktowane jak terminal, dzięki czemu wiele procesów może być pierwszoplanowych (po jednym w oknie) w dowolnej chwili. Oczywiście, z każdym z okien mogą być związane procesy drugoplanowe. Powłoka Korna również umożliwia sterowanie zadaniami. Wydaje się, że sterowanie zadaniami (i grupy procesów) będzie standardem w przyszłych wersjach systemu UNIX.

21.4.2 Standardowe wejście-wyjście

Procesy mogą otwierać pliki na własne życzenie, lecz większość procesów pracuje przy założeniu, że w chwili ich rozpoczęcia są otwarte trzy pliki (o deskryptorach 0, 1 i 2). Te deskryptory plików są dziedziczone poprzez operację `fork` (i być może `execve`), która utworzyła proces. Są one znane jako *standardowe wejście (0)*, *standardowe wyjście (1)* i *standardowe wyjście diagnostyczne (3)*^{*}. Te trzy deskryptory są często związane z terminalem użytkownika. Dzięki temu program może czytać ze standardowego wejścia to, co napisze użytkownik, oraz posyłać wyniki na ekran użytkownika, pisząc na standardowym wyjściu. Deskryptor standardowego pliku diagnostycznego

* Angielskie nazwy brzmią odpowiednio: *standard input*, *standard output* i *standard error*. – Przyp. tłum.

Polecenie	Znaczenie polecenia
% ls > plik_a	kieruje wyniki polecenia <i>ls</i> do pliku <i>plik_a</i>
% pr < plik_a > plik_b	wejście z <i>plik_a</i> i wyjście do <i>plik_b</i>
% lpr < plik_b	wejście z pliku <i>plik_b</i>
%	przechowanie w pliku zarówno standardowego
% make program >& errs	wyjścia, jak i standardowej sygnalizacji błędów

Rys. 21.5 Standardowe przeadresowanie wejścia-wyjścia

jest również otwarty do pisania i używany jako wyjście dla komunikatów o błędach. Standardowe wyjście jest używane do wyprowadzania zwykłych wyników. Większość programów może również akceptować pliki na standardowym wejściu i standardowym wyjściu (zamiast końcówki konwersacyjnej). Dla programu nie ma znaczenia, skąd czerpie dane wejściowe i dokąd posyła wyniki. Jest to jedna z eleganckich cech projektu systemu UNIX*.

Popularne powłoki mają prostą składnię poleceń określającą zmiany w przypisaniach plików otwieranych jako standardowe strumienie na wejściu i wyjściu procesu. Zmiana standardowego pliku jest nazywana *przeadresowaniem wejścia-wyjścia* (ang. *I/O redirection*). Składnia przeadresowania wejścia-wyjścia jest pokazana na rys. 21.5. W tym przykładzie polecenie *ls* tworzy wykaz nazw plików w bieżącym katalogu, polecenie *pr* nadaje temu wykazowi postać odpowiednią dla formatu stron drukarki, a polecenie *lpr* ustawia sformatowane wyjście w kolejce do drukarki – na przykład do urządzenia */dev/lp0*. Następne polecenie wymusza kierowanie wszystkich wyników i wszystkich komunikatów o błędach do pliku. Gdyby nie było znaku *&*, to komunikaty o błędach ukazywałyby się na terminalu.

21.4.3 Potoki, filtry i skrypty powłokowe

Pierwsze trzy polecenia z rys. 21.5 można by zestawić w jedno polecenie

% *ls* | *pr* | *lpr*

Każda pionowa kreska informuje powłokę, że wyjście poprzedniego polecenia ma być przekazane jako wejście następnego polecenia. Do przeniesienia danych od jednego procesu do drugiego używa się potoku. Jeden proces pisze na jednym końcu potoku, drugi zaś czyta z drugiego końca potoku. W naszym przykładzie zapisywany koniec jednego potoku zostanie ustalony przez powłokę jako standardowe wyjście polecenia *ls*, a czytany koniec potoku będzie standardowym wejściem polecenia *pr*. Między poleceniami *pr* i *lpr* będzie ustalony inny potok.

* Niezależność procesów od rzeczywistych urządzeń zewnętrznych występowała już we wcześniejszych niż UNIX systemach operacyjnych – Przyp. tłum.

Polecenie takie jak *pr*, które przekazuje swoje standardowe wejście na standardowe wyjście, przetwarzając je przy tym w pewien sposób, nazywa się *filtrem* (ang. *filter*). Wiele poleceń systemu UNIX można używać jako filtrów. Przez składanie w potoki typowych poleceń można otrzymywać skomplikowane działania. Z kolei typowych funkcji, takich jak formatowanie wyjścia, nie trzeba wbudowywać w liczne polecenia, ponieważ wyjście prawie każdego programu może być przepuszczone przez filtr *pr* (lub jakiś inny, odpowiedni filtr).

Obie popularne powłoki systemu UNIX są także językami programowania ze zmiennymi powłokowymi i typowymi dla języków wyższego poziomu konstrukcjami sterującymi (pętle, działania warunkowe). Wykonanie polecenia jest analogiczne do wywołania podprogramu. Plik poleceń dla powłoki, nazywany *skryptem powłokowym* (ang. *shell script*)¹, może być wykonany tak jak każde inne polecenie, z automatycznym wywołaniem odpowiedniej powłoki w celu jego odczytywania. *Programowanie powłokowe* (ang. *shell programming*) można zatem wykorzystać do wygodnego zestawiania zwykłych programów w wyrafinowane zastosowania, bez zadnej konieczności programowania w konwencjonalnych językach.

Ten zewnętrzny obraz systemu, widziany przez użytkownika, jest po-wszechnie uważany za definicję systemu UNIX, wszakże jest to definicja najłatwiej poddająca się zmianom. Napisanie nowej powłoki, z zupełnie odmienną składnią i semantyką zmieniłoby bardzo sposób widzenia systemu przez użytkownika, nie zmieniając jądra, a nawet interfejsu programisty. Obecnie istnieje kilka sterowanych za pomocą ofert (menu) i ikon interfejsów z systemem UNIX, a X Window System gwałtownie urasta do rangi standaru. Sercem systemu UNIX jest niewątpliwie jego jądro. Zmiana jądra jest znacznie bardziej trudna niż zmiana interfejsu użytkownika, gdyż wszystkie programy zależą od funkcji systemowych, które dla zachowania spójności są realizowane przez jądro. Oczywiście, w celu zwiększenia funkcjonalności można dodawać nowe wywołania systemowe, lecz trzeba wtedy modyfikować programy, by mogły korzystać z tych nowych wywołań.

21.5 ■ Zarządzanie procesami

Ważnym zagadnieniem projektowym w systemach operacyjnych jest reprezentacja procesów. Jedną z zasadniczych różnic między systemem UNIX a wieloma innymi systemami jest łatwość tworzenia wielu procesów i mani-

¹ Albo „scenariuszem powłokowym”, niezależnie od specyficznej unikowej terminologii warto tu przytoczyć synonimy z innych środowisk: plik wsadowy i makrodefinicja. —Przyp. tłum.

pulowania nimi. Procesy są reprezentowane w systemie UNIX przez rozmaite bloki kontrolne. W przestrzeni pamięci wirtualnej dostępnej dla procesu użytkownika nie ma żadnych systemowych bloków kontrolnych – związane z procesami bloki kontrolne są przechowywane w jądrze. Jądro używa zawartych w tych blokach informacji do sterowania procesami i planowania przydziału procesora.

21.5.1 Bloki kontrolne procesu

Podstawową strukturą danych związaną z procesami jest *struktura procesu* (ang. *process structure*). Struktura procesu zawiera wszystkie niezbędne informacje o procesie potrzebne systemowi na wypadek wymiany procesu, tj. odesłania go z pamięci operacyjnej na dysk, a w tym: jednoznaczny identyfikator procesu, dane dotyczące planowania (jak priorytet procesu) i wskaźniki do innych bloków kontrolnych. Istnieje tablica struktur procesów, której długość jest definiowana podczas konsolidacji systemu. Struktury procesów gotowych do wykonania są połączone ze sobą przez listę przydziału procesora w listę dwukierunkową (kolejka procesów gotowych). Z każdej struktury procesu prowadzą wskaźniki do odpowiadającego mu procesu macierzystego, jego najmłodszego, żyjącego potomka i do wielu innych pokrewnych informacji, w rodzaju wykazu procesów dzielących ten sam kod programu (zwany tekstem).

Wirtualna przestrzeń adresowa procesu użytkownika jest podzielona na segmenty tekstu* (kodu programu), danych i stosu. Segmente danych i stosu znajdują się zawsze w tej samej przestrzeni adresowej, ale mogą rosnąć niezależnie i zwykle w przeciwnych kierunkach. Stos najczęściej rośnie w dół, a dane w górę, ku niemu. Segment tekstu jest czasem (jak w przypadku mikroprocesora Intel 8086 z odrębnymi obszarami rozkazów i danych) w innej przestrzeni adresowej niż dane wraz ze stosem i na ogół jest przeznaczony tylko do czytania. Program uruchomieniowy (ang. *debugger*) określa dla segmentu tekstu tryb czytania i pisania, aby można było wstawać do niego punkty kontrolne.

Każdy proces z dzielonym tekstem (pod systemem 4.3BSD – prawie wszystkie procesy) ma wskaźnik w swojej strukturze procesu prowadzący do *struktury tekstu* (ang. *text structure*). Struktura tekstu przechowuje informację o tym, ile procesów używa danego segmentu tekstu, włącznie ze wskaźnikiem do listy ich struktur, oraz gdzie na dysku znajduje się tablica stron danego segmentu tekstu na wypadek jego wymiany. Sama struktura tekstu rezyduje

* „Tekst” na określenie segmentu rozkazów maszynowych jest kolejnym, mocą zwyczaju utartym, terminem unikowym – Przyp. tłum.

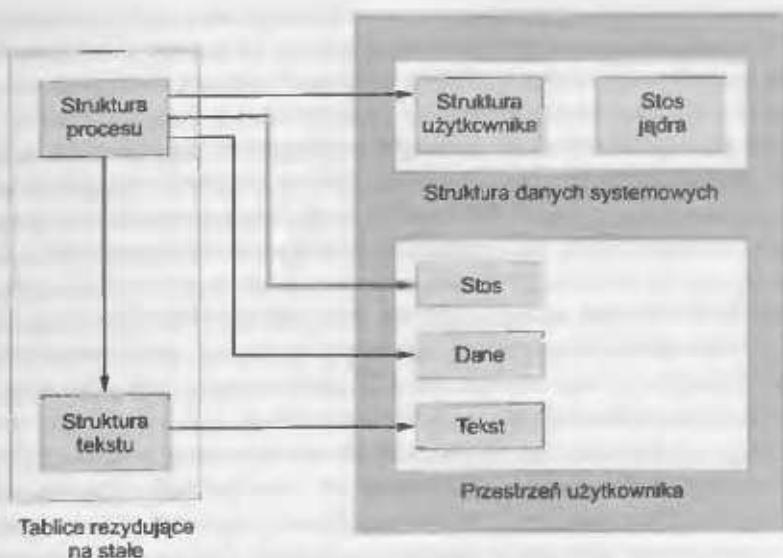
na stałe w pamięci głównej – miejsce na tablicę takich struktur zostaje przydzielone podczas konsolidacji systemu. Segmente tekstu, danych oraz stosu procesów mogą podlegać wymianie. Gdy segmenty są z powrotem sprowadzane do pamięci, wówczas dzieli się je na strony.

W tablicach stron są przechowywane informacje o odwzorowaniach wirtualnej pamięci procesów na pamięć fizyczną. Struktura procesu zawiera wskaźniki do tablicy stron, używane podczas pobytu procesu w pamięci głównej, lub adres procesu na urządzeniu wymiany – jeśli proces został usunięty z pamięci. Nie ma specjalnej, osobnej tablicy stron dla dzielonego segmentu tekstu. Każdy proces korzystający ze wspólnego segmentu tekstu ma jego strony odnotowane we własnej tablicy stron.

Informacje o procesie potrzebne tylko wówczas, gdy proces przebywa w pamięci na stałe (nie podlega wymianie), są przechowywane w *strukturze użytkownika*, inaczej – w *u-strukturze*, a nie w strukturze procesu. Struktura użytkownika jest odwzorowywana w trybie tylko do czytania w wirtualnej przestrzeni adresowej użytkownika, więc procesy użytkowe mogą czytać jej zawartość. Jądro ma prawo zapisywania tej struktury. W komputerach VAX znajduje się w niej kopia bloku kontrolnego procesu komputera VAX – w ten sposób przechowuje się uniwersalne rejesty procesu, wskaźnik stosu, licznik rozkazów oraz rejesty bazowe tablicy stron, w czasie gdy proces nie jest wykonywany. Istnieje obszar służący do przechowywania parametrów funkcji systemowych oraz ich wartości. Są w nim przechowywane identyfikatory wszystkich użytkowników i grup związanych z procesem (oprócz skutecznego identyfikatora użytkownika, który znajduje się w strukturze procesu). Przechowuje się tam też struktury danych sygnałów, czasomierzy i parametrów ograniczających. Z rzeczy w bardziej oczywisty sposób niezbędnych dla zwykłego użytkownika utrzymuje się w strukturze użytkownika dane o bieżącym katalogu i tablicę otwartych plików.

Każdy proces ma zarówno fazę użytkownika, jak i fazę systemową. Większość zwykłej pracy jest wykonywana w *trybie użytkownika*, jednak wykonywanie funkcji systemowej przebiega w *trybie systemowym*. Fazy systemu i użytkownika nigdy nie występują w procesie jednocześnie. Gdy proces działa w trybie systemowym, wówczas korzysta ze *stosu jądra*, a nie ze stosu znajdującego się w jego własnym obszarze użytkownika. Stos jądra dla danego procesu znajduje się zaraz za strukturą użytkownika. Stos jądra wraz ze strukturą użytkownika tworzą *systemowy segment danych procesu*. Jądro ma własny stos, którego używa wtedy, kiedy nie wykonuje pracy na zamówienie procesu (np. do obsługiwanego przerwań).

Na rysunku 21.6 widać, jak można użyć struktury procesu do odnajdywania różnych części procesu.



Rys. 21.6 Odnajdywanie części procesu za pomocą struktury procesu

Funkcja systemowa **fork** przydaje miejsce na nową strukturę procesu (z nowym identyfikatorem procesu) dla procesu potomnego i kopiuje strukturę użytkownika. Nowa struktura tekstu nie jest na ogół potrzebna, ponieważ procesy dzielą teksty; uaktualnia się po prostu odpowiednie liczniki i wykazy. Zakłada się nową tablicę stron i przydziela nowe miejsce w pamięci głównej na segmenty danych i stosu procesu potomnego. Kopiowanie struktury użytkownika zachowuje otwarte deskryptory plików, identyfikatory użytkownika i grupy, stan obsługi sygnałów i większość tego rodzaju cech procesu.

Funkcja systemowa **vfork** nie kopiuje do nowego procesu danych i stosu. Zamiast tego nowy proces dzieli po prostu tablicę stron ze starym procesem. Niemniej jednak, zawsze tworzy się nową strukturę użytkownika i strukturę procesu. Funkcja **vfork** jest często używana przez powłokę w celu wykonania polecenia i czekania na jego zakończenie. Proces macierzysty używa funkcji **vfork** do utworzenia procesu potomnego. Ponieważ proces potomny od razu przystępuje do wykonania operacji **execve** w celu całkowitej zmiany swojej wirtualnej przestrzeni adresowej, nie ma powodu, aby dostarczać mu pełny obraz procesu macierzystego. Struktury danych takie jak te, które są potrzebne do manipulowania potokami, mogą między wykonaniem funkcji **vfork** a **execve** pozostawać w rejestrach. Pliki w jednym procesie mogą być zamknięte bez oddziaływanego na inny proces, ponieważ zaangażowana w to jądrowa struktura danych zależy od struktury użytkownika, która nie podlega dzieleniu. Po wywołaniu funkcji **vfork** wykonywanie procesu macierzystego jest

zawieszane do czasu, gdy potomek wywoła funkcję `execve` lub zakończy działanie, zatem proces macierzysty nie zmieni pamięci potrzebnej potomkowi.

Gdy proces macierzysty jest duży, wówczas funkcja `vfork` może przyczynić się do znacznego zaoszczędzenia czasu procesora. Jednakże jest to dość niebezpieczne wywołanie systemowe, gdyż do czasu wystąpienia `execve` każda zmiana pamięci pojawia się w obu procesach. Inną możliwością jest dzielenie wszystkich stron przez podwojenie tablicy stron z oznaczeniem w elementach obu tablic, że strony mają być *kopiowane przy zapisie*. Sprzętowe bity ochrony ustawia się tak, by wychwycić każdą próbę pisania na stronach dzielonych. Jeśli zdarzenie takie wystąpi, to rezerwuje się nową ramkę i kopiuje do niej stronę dzieloną. Tablice stron koryguje się tak, aby pokazywały, że ta strona nie podlega już dzieleniu (a więc nie musi być dłużej chroniona przed pisaniem), po czym wykonywanie procesu może zostać wznowione.

Funkcja systemowa `execve` nie tworzy żadnej nowej struktury procesu lub użytkownika – w zamian zastępuje tekst i dane procesu. Otwarte pliki są zachowywane (choć istnieje sposób określenia, aby deskryptory pewnych plików zostały przy wywołaniu `execve` zamknięte). Zachowuje się większość cech dotyczących obsługi sygnałów, lecz z oczywistych przyczyn kasuje się ustalenia dotyczące wywołania specyficznych procedur użytkownika na wypadek sygnałów. Identyfikator procesu i większość innych właściwości pozostają nie zmienione.

21.5.2 Planowanie przydziału procesora

Planowanie przydziału procesora w systemie UNIX zaprojektowano z myślą o procesach interakcyjnych. Procesy otrzymują małe kwanty czasu procesora według algorytmu priorytetowego, który dla zadań ograniczonych przez procesor redukuje się do algorytmu rotacyjnego.

Każdy proces ma przypisany *priorytet planowania*. Większe liczby oznaczają niższe priorytety. Procesy wykonujące operacje dyskowe lub inne ważne prace mają priorytety mniejsze niż „pzero” i nie mogą być zlikwidowane przez sygnały. Procesy zwykłych użytkowników mają priorytety dodatnie i mniejszą szansę na działanie niż dowolny proces systemowy, choć mogą udzielać sobie wzajemnie pierwszeństwa za pomocą polecenia `nice`.

Im więcej proces zużywa czasu procesora, tym niższy (większa liczba dodatnia) staje się jego priorytet – i na odwrót, tak że w planowaniu przydziału procesora istnieje ujemne sprzężenie zwrotne, więc zmonopolizowanie procesora przez jeden proces jest rzeczą trudną. W celu uniknięcia głodzenia stosuje się postarzanie procesów.

W starszych wersjach systemu UNIX stosowano w planowaniu rotacyjnym jednosekundowy kwant czasu. System 4.3BSD zmienia zaplanowanie

procesów co 0,1 s, a priorytety przelicza co sekundę. Planowanie rotacyjne jest realizowane przy użyciu mechanizmu *odliczania czasu* (ang. *timeout*), który powoduje, że moduł obsługi przerwań zegarowych wywołuje podprogram jądra po upływie określonego czasu. Wywoływany w tym przypadku podprogram dokonuje ponownego zaplanowania procesora, po czym przedkłada żądanie uaktywnienia samego siebie po odmierzeniu kolejnego okresu. Okresowe przeliczanie priorytetów odbywa się również za sprawą podprogramu, który zaplansowuje własne uaktywnianie na zasadzie *przerwań zegarowych*.

W jądrze nie ma wywłaszczenia jednego procesu przez drugi. Proces może rzucić się procesora, jeśli oczekuje na wejście-wyjście lub z powodu wyčerpania przydzielonego czasu. Proces, który zwalnia procesor, zostaje uspiony w oczekiwaniu na *zdarzenie*. Służąca do tego celu elementarna operacja jądra zwiększa się *sleep* (nie należy jej mylić z mającym taką samą nazwę podprogramem bibliotecznym wywoływanym przez użytkownika). Pobiera ona argument traktowany jako adres struktury danych w jądrze związanej ze zdarzeniem, którego wystąpienie warunkuje obudzenie procesu. Gdy dane zdarzenie wystąpi, wówczas zorientowany o tym proces systemowy wywołuje operację *wakeup* (pobudka!), zaopatrując ją w adres odpowiadający zdarzeniu, a wtedy wszystkie procesy, które spały pod tym samym adresem, są wstawiane do kolejki procesów kandydujących do procesora (gotowych).

Na przykład proces czekający na zakończenie dyskowej operacji wejścia-wyjścia *zaśnie* pod adresem nagłówka bufora odpowiadającego przesyłanym danym. Kiedy procedura obsługi przerwania w module sterującym dysku odnotuje zakończenie przesyłania danych, wtedy wywoła operację *wakeup* w odniesieniu do nagłówka tego bufora. Procedura obsługi przerwania używa stosu jądra niezależnie od tego, jaki proces był aktualnie wykonywany, i operacja *wakeup* jest wykonywana z tego procesu systemowego.

Proces, który rzeczywiście podejmie pracę, zostaje wybrany przez planistę przydziału procesora. Operacja *sleep* ma w tym celu drugi argument określający priorytet zaplanowania. Jeśli priorytet ten jest mniejszy niż „*pzero*”, to proces będzie chroniony przed przedwczesnym obudzeniem przez jakieś zdarzenie wyjątkowe, takie jak wygenerowanie sygnału.

Jeśli nadjejdzie sygnał, to pozostaje on nie obsłużony dopóki proces, którego ten sygnał dotyczy, nie wejdzie ponownie w systemowy tryb pracy. Na ogół nie trwa to długo, gdyż sygnał z reguły powoduje obudzenie procesu, jeśli ten oczekiwany z jakiegoś innego powodu.

Ze zdarzeniami nie jest wiązana żadna pamięć, więc proces wywołujący podprogram, który wykonuje operację *sleep* oczekującą na zdarzenie, musi być przygotowany na obsługę przedwczesnego powrotu, włączając w to możliwość zaniku przyczyny oczekiwania.

W mechanizmie zdarzeń dochodzi do szkodliwej *rywalizacji* (ang. *race conditions*). Gdyby proces zdecydował się (sprawdziwszy np. stan jakiegoś znacznika w pamięci) na zainicjowanie w oczekiwaniu na zdarzenie i zdarzenie to wystąpiłoby, zanim proces zdolalby wykonać elementarną operację usypiania, to proces mógłby pozostać uspiony na zawsze. Sytuacji tej zapobiega się przez zwiększenie sprzętowego priorytetu procesora na czas wykonywania sekcji krytycznej, tak aby nie mogło wystąpić żadne przerwanie, wówczas będzie więc wykonywany tylko proces potrzebujący zdarzenia, aż nie zostanie uspiony. Ze sprzętowego priorytetu procesora korzysta się w ten sposób do ochrony sekcji krytycznych w obrębie całego jądra, co jest największą przeszkołą w przenoszeniu systemu UNIX na maszyny wieloprocesorowe. Jednak pomimo tego utrudnienia dokonano już wielu takich przeniesień.

Wiele procesów, takich jak edytory tekstu, jest ograniczonych przez wejście-wyjście, toteż podstawą ich planowania będą na ogół oczekiwania na operacje wejścia-wyjścia. Doświadczenie podpowiada, że planista przydziału procesora w systemie UNIX będzie spisywać się najlepiej w odniesieniu do zadań zależnych od wejścia-wyjścia. Można to zaobserwować na podstawie wykonywania pewnej liczby zadań ograniczonych przez wejście-wyjście, takich jak programy formatowania tekstu lub interpretery języków programowania.

To, co nazywaliśmy tutaj planowaniem przydziału procesora, odpowiada ściśle planowaniu *krótkoterminowemu* z rozdz. 4, chociaż ujemne sprzężenia zwrotne w schemacie priorytetów dotyczy długoterminowego planowania mieszkani zadań. Średnioterminowe planowanie dokonuje się za pomocą mechanizmu wymiany opisanego w p. 21.6.

21.6 ■ Zarządzanie pamięcią

Większość wczesnych prac nad systemem UNIX została wykonana dla komputera PDP-11. Miał on tylko osiem segmentów wirtualnej przestrzeni adresowej po co najwyżej 8192 B. Większe maszyny, takie jak PDP-11/70, umożliwiały rozdzielenie obszaru rozkazów i adresów danych. Dzięki temu przestrzeń adresowa i liczba segmentów mogła dwukrotnie wzrosnąć, lecz była to wciąż stosunkowo mała przestrzeń adresowa. Ponadto jądro było jeszcze bardziej ograniczane wskutek przeznaczenia jednego segmentu danych na wektor przerwań, drugiego na wskazywanie związkanych z każdym procesem systemowych segmentów danych i jeszcze innego na rejesty szyny UNIBUS (systemowej szyny wejścia-wyjścia). Co więcej, w mniejszych maszynach PDP-11 cała pamięć fizyczna była ograniczona do 256 KB. Łączne zasoby pamięci były niestarczające, aby usprawiedliwić lub urzeczywistnić złożone algorytmy zarządzania pamięcią. Toteż UNIX wymieniał całe obrazy pamięci procesów.

21.6.1 Wymiana

W systemach uniksowych poprzedzających wersję 3BSD zastosowano wymianę wyłącznie do obsługi rywalizacji między procesami. Jeśli rywalizacja jest za duża, to procesy są usuwane z pamięci operacyjnej do czasu, aż znajdzie się w niej wystarczająco dużo miejsca. Może też dochodzić do usuwania z pamięci wielu małych procesów przez kilka dużych procesów, a procesy wymagające większego obszaru pamięci niż pamięć operacyjna pozostająca poza jądrem w ogóle nie mogą być wykonane. Systemowy segment danych (*u*-struktura i stos jądra) oraz segment danych użytkownika (tekst, jeśli nie dzielony, dane i stos) umieszcza się w ciągłej pamięci głównej, aby usprawnić transmisję przy wymianie, toteż poważnym problemem może być zewnętrzna fragmentacja pamięci.

Przydział obszarów pamięci operacyjnej i przestrzeni wymiany odbywa się na zasadzie pierwszego dopasowania (ang. *first-fit*). Gdy rozmiar pamięci procesu wzrasta (z powodu powiększania się stosu lub obszaru danych), wówczas przydziela się nowy obszar pamięci wystarczający do pomieszczenia całego obrazu pamięci procesu. Po przekopiowaniu obrazu pamięci poprzednio zajmowana pamięć zostaje zwolniona, a odpowiednie tablice – aktualnione. (W niektórych systemach usiłuje się znaleźć ciągły obszar pamięci przylegający do końca bieżącego obszaru, aby uniknąć części kopiowania). Jeśli nic ma pojedynczego, wystarczająco dużego kawałka pamięci, to proces odsyła się na dysk, z którego powróci już w nowym rozmiarze.

Nie ma potrzeby wysyłania na dysk wspólnego segmentu tekstu, ponieważ jest on przeznaczony wyłącznie do czytania. Nic ma też powodu, aby dzielony segment tekstu dla jakiegoś procesu wczytywać do pamięci, jeśli znajduje się tam już inny jego egzemplarz. Jedną z głównych przyczyn tego, że utrzymuje się informację o wspólnie używanych segmentach tekstu, jest zysk w postaci mniejszej liczby wymian. Drugą przyczyną jest zmniejszenie ilości pamięci operacyjnej potrzebnej dla wielu procesów używających tego samego segmentu tekstu.

Decyzje o tym, który proces wysłać na dysk, a który wprowadzić do pamięci operacyjnej, są podejmowane przez *proces planujący* (zwany również *procesem wymiany* – ang. *swapper*). *Ten planista* (ang. *scheduler*) budzi się przynajmniej raz na 4 s, aby sprawdzić zapotrzebowanie na obustronną wymianę procesów. Jest większa szansa na to, że proces ulegnie wymianie, jeśli pozostaje bezczynny, przebywa w pamięci operacyjnej od dłuższego czasu lub jeśli jest wielki. Jeśli nie ma ewidentnych kandydatów do wymiany, to wybiera się procesy według ich wieku. Szansa na wprowadzenie procesu z powrotem do pamięci operacyjnej wzrasta, gdy przebywa on na dysku przez dłuższy czas lub jest mały. Dokonuje się sprawdzeń zapobiegających szam-

taniu, przede wszystkim nie dopuszczając do wymiany procesu, który nie pozostawał w pamięci przez pewien czas.

Jeśli zadania nie muszą być wymieniane, to przegląda się tablicę procesów, aby znaleźć proces zasługujący na sprowadzenie do pamięci operacyjnej (na podstawie rozmiaru procesu i czasu jego pobytu na dysku). Jeśli nie ma wystarczająco dużo wolnej pamięci, to dopóty odsyła się procesy na dysk, dopóki nie uzyska się odpowiednio dużo miejsca.

W systemie 4.3BSD przestrzeń wymiany jest przydzielana w kawałkach, które są wielokrotnością potęgi 2 i rozmiaru minimalnego (np. 32 strony); największy przydzielany obszar jest określony wielkością strefy wymiany na dysku. Jeśli do celów wymiany można użyć kilku logicznych stref dyskowych, to z powyższej przyczyny powinny być one wszystkie tej samej wielkości. Poszczególne logiczne strefy dyskowe powinny też znajdować się w zasięgu osobnych ramion dyskowych w celu zminimalizowania przeszukiwania dysku.

W wielu systemach wciąż jeszcze stosuje się schemat wymiany opisany powyżej. Natomiast we wszystkich systemach uniksowych z Berkeley zarządzanie rywalizacją procesów o pamięć polega przede wszystkim na stronicowaniu, a tylko pomocniczo na wymianie. Schemat stosowany do określania, który proces należy odesłać na dysk, który zaś z niego sprowadzić, jest w zarysie podobny do tradycyjnego, lecz różni się w szczegółach, a znaczenie wymiany jest mniejsze.

21.6.2 Stronicowanie

W Berkeley wprowadzono stronicowanie do systemu UNIX, poczynając od wersji 3BSD. Wersja VAX 4.2BSD jest systemem pamięci wirtualnej stronicowanej na żądanie. Stronicowanie wyeliminowało zewnętrzną fragmentację pamięci. (Istnieje, rzec jasna, fragmentacja wewnętrzna, lecz jest niewielka ze względu na stosunkowo mały rozmiar stron). Wymianę udaje się zredukować do minimum, gdyż w pamięci operacyjnej można umieścić więcej zadań dzięki temu, że stronicowanie umożliwia wykonywanie procesów reprezentowanych w pamięci tylko przez swoje części.

Stronicowanie na żądanie (ang. *demand paging*) realizuje się w prosty sposób. Kiedy proces potrzebuje strony, której nie ma, wtedy w jądrze jest zgłoszony brak strony, następuje przydzielenie ramki pamięci operacyjnej i przeczytanie do niej właściwej strony z dysku.

Istnieje kilka optymalizacji. Jeśli potrzebna strona wciąż jeszcze znajduje się w tablicy stron procesu, lecz została unieważniona przez proces zastępowania stron, to można przywrócić jej ważność i posłużyć się nią – bcz żadnej operacji wejścia-wyjścia. Podobnie, strony mogą być odzyskiwane z listy

wolnych ramek. Po rozpoczęciu większości procesów wiele ich stron wstępnie sprowadza się i odkłada na listę wolnych ramek w celu odzyskiwania przy użyciu tego mechanizmu. Można również spowodować, by przy rozpoczętym procesu nie było wstępnego stronicowania, lecz rzadko się tak robi, ponieważ zwiększa to nakłady na obsługę braków stron, przypominając bardziej czyste stronicowanie na żądanie.

Jeśli strona ma być pobrana z dysku, to należy ją zablokować* w pamięci podczas przesyłania. Takie zablokowanie zapewnia, że strona nie zostanie wybrana do zastąpienia. Po sprowadzeniu strony i właściwym jej odwzorowaniu, blokada musi być utrzymywana, jeśli do strony odnoszą się surowe, fizyczne operacje wejścia-wyjścia.

Bardziej interesujący jest *algorytm zastępowania stron* (ang. *page-replacement*). Komputer VAX nie ma sprzętowego bitu odniesienia do strony. Ten brak wsparcia sprzętowego powoduje bezużyteczność wielu algorytmów zarządzania pamięcią, na przykład metody zliczania częstości błędów stron. W systemie 4.2BSD stosuje się zmodyfikowany *algorytm drugiej szansy* (zegarowy) opisany w p. 9.5.4. Mapa pamięci operacyjnej poza jądrem (ang. *core map* lub *cmap*) jest ustawicznie i po kolei obiegana przez programową wskazówkę zegara. Gdy wskazówka zegara znajdzie się nad daną ramką i ramka jest oznaczona jako będąca w użyciu ze względu na pewną sytuację programową (np. używa tej ramki fizyczna operacja wejścia-wyjścia) lub jest już wolna, wówczas ramka ta pozostaje nietknięta, a wskazówka zegara przechodzi do następnej ramki. W przeciwnym razie lokalizuje się odpowiadającą tej ramce pozycję w tablicy stron tekstu lub procesu. Jeśli ta pozycja jest już unieważniona, to ramkę dodaje się do wykazu wolnych ramek, w przeciwnym razie unieważnia się wpis w tablicy stron. Ileż może to podlegać reklamacji (jeśli strona nie zostanie zmieniona przed następnym do niej odniesieniem, to wpis z powrotem nabierze ważności).

W systemie 4.3BSD Tahoe przewidziano możliwość skorzystania ze sprzętowego bitu odniesienia do strony. W tym systemie w pierwszym obiegu zegara zeruje się bity odniesień, a w drugim obiegu na wykazie stron nie zajętych i gotowych do zastąpienia umieszcza się te strony, których bity odniesienia pozostały wyzerowane. Oczywiście, jeśli strona jest zabrudzona (komputer VAX ma bit zabrudzenia), to zanim jej ramkę doda się do listy wolnych ramek, strona musi być zapisana na dysku.

Aby liczba ważnych stron danych procesu nie zmalała za bardzo, a także by urządzenie stronicujące nie załala powódź zamówień, wykonuje się odpowiednie sprawdzenia. Istnieje także mechanizm, za pomocą którego proces może ograniczyć ilość używanej przez siebie pamięci.

* Tzn. miejsce (ramkę), do którego zostanie sprowadzona – Przyp. tłum.

Wskazówka zegara w algorytmie LRU jest implementowana za pomocą *demonu stron* (ang. *pagedaemon*), który jest procesem numer 2 (pamiętajmy, że *planista* jest procesem numer 0, a proces *init* ma numer 1). Proces ten przez większość czasu jest uśpiony, lecz kilka razy na sekundę wykonuje się sprawdzenie (zaplanowane odliczaniem czasu), czy nie jest potrzebne jego działanie. Jeśli tak, to proces 2 zostaje obudzony. Gdy tylko liczba wolnych ramek zmniejszy się poniżej pewnego poziomu, określonego zmienną *lotsfree*, demon stron budzi się. Tak więc, gdy ciągle jest dość wolnej pamięci, demon stron nie obciąża systemu, bo nigdy nie musi pracować.

Przesunięcie wskazówki zegara przy każdym obudzeniu procesu demona stron (tj. liczba przeglądniętych ramek, która zwykle przewyższa liczbę wybrzuconych stron) jest ustalane zarówno na podstawie liczby ramek brakujących do osiągnięcia wartości *lotsfree*, jak i przez liczbę ramek, którą planista określił jako z różnych powodów wymaganą (im więcej potrzeba ramek, tym dłuższa jest droga wskazówki). Jeśli liczba wolnych ramek wzrasta do wartości *lotsfree*, zanim zakończy się oczekiwane przesunięcie, to wskazówka zatrzymuje się i demon stron usypia. Parametry określające zakres ruchu wskazówki zegara są wyznaczone przy rozruchu systemu stosownie do wielkości pamięci operacyjnej, tak aby demon stron nie zużywał więcej niż 10% całego czasu procesora.

Jeśli planista uzna, że system stronicowania jest przeciążony, to zacznie wymieniać całe procesy, co potrwa aż do ustąpienia przeciążenia. Do wymiany takiej dochodzi zazwyczaj tylko wtedy, kiedy wystąpi kilka warunków: system jest ustawicznie mocno załadowany, maleje ilość wolnej pamięci poniżej dolnej granicy — *minfree*, a średnia ilość ostatnio dostępnej pamięci jest mniejsza od pożąданej wartości *desfree*, przy czym *lotsfree* > *desfree* > *minfree*. Innymi słowy, wymianę spowoduje tylko chroniczny brak pamięci dla kilku procesów próbujących działać, i to tylko wtedy, gdy w danej chwili wolna pamięć jest bardzo mała. (Nadmierna aktywność stronicowania lub zapotrzebowanie na pamięć ze strony samego jądra mogą być również, choć w rzadkich przypadkach, uwzględnione w tych obliczeniach). Oczywiście, procesy mogą zostać wymienione przez planistę z innych przyczyn (choćażby z powodu długotrwałej bezczynności).

Parametr *lotsfree* jest równy na ogół jednej czwartej pamięci odwzorowanej w mapie obieganej przez wskazówkę zegara, a parametry *desfree* i *minfree* mają zwykle te same wartości w różnych systemach, lecz są ograniczane do ułamków dostępnej pamięci.

Każdy segment tekstu procesu jest z założenia wspólny i dostępny tylko do czytania. Schemat taki jest praktyczny przy stronicowaniu, ponieważ nie ma tu zewnętrznej fragmentacji, a zyskiwana przestrzeń wymiany przeważa znakomy koszt dzielenia, zwłaszcza że wirtualny obszar jądra jest duży.

Planowanie przydziału procesora, wymiana obszarów pamięci i stronicowanie współdziałają ze sobą. Im niższy jest priorytet procesu, tym bardziej jest prawdopodobne, że jego strony zostaną usunięte, i tym większa szansa, że zostanie wyrzucony z pamięci w całości. Preferencje wieku procesu przy wyborze kandydatów do wymiany strzegą przed szamotaniem, lecz stronicowanie jest tu skuteczniejsze. W idealnej sytuacji procesy nie są wymieniane, chyba że są bezczynne. W dowolnej chwili każdy proces potrzebuje tylko niewielkiego roboczego zbioru stron w pamięci operacyjnej, a demon stron może oddawać nie używane strony do dyspozycji innych procesów.

Wymagana przez proces ilość pamięci stanowi ulamek całego wirtualnego rozmiaru procesu, dochodzący do 0,5, jeśli dany proces był wymieniany dawno temu.

Strony sprzętowe komputera VAX, których rozmiar wynosi 512 B, są za małe, aby operacje wejścia-wyjścia były wydajne, toteż strony takie grupuje się po dwie, tak że faktycznie wszystkie stronicowane operacje wejścia-wyjścia odbywają się porcjami po 1024 B. Inaczej mówiąc, efektywny rozmiar strony nie musi się równać rozmiarowi strony sprzętowej danej maszyny, chociaż musi być jego wielokrotnością.

21.7 ■ System plików

System plików UNIX działa na dwu podstawowych obiektach: plikach i katalogach. Katalogi też są plikami, choć o specjalnej budowie, toteż podstawową koncepcją w systemie UNIX jest reprezentacja pliku.

21.7.1 Bloki i fragmenty

Większość systemu plików zajmuje *bloki danych*, które zawierają wszystko to, co użytkownicy umieszczają w swoich plikach. Rozważmy, jak te bloki danych są pamiętane na dysku.

Sprzętowy sektor dysku ma zwykle 512 B. Ze względu na szybkość działania jest wskazane, aby długość bloku była większa niż 512 B. Jednak zważywszy, że uniksowe systemy plików zawierają zazwyczaj wielką liczbę małych plików, użycie znacznie większych bloków spowodowałoby zbyt dużą fragmentację wewnętrzną. Z tego powodu we wcześniejszym systemie plików 4.1BSD blok był ograniczony do 1024 B (1 KB).

Rozwiążanie zastosowane w systemie 4.2BSD polega na zastosowaniu dla plików, które nie mają bloków adresowanych posrednio, dwóch rozmiarów bloków. Wszystkie bloki pliku, z wyjątkiem ostatniego, są *dużego rozmiaru* (i wynoszą np. 8 KB). Ostatni blok ma odpowiednią wielokrotność

rozmiaru mniejszego *fragmentu* (np. 1024 B) i mieści resztę pliku. Zatem plik o rozmiarze 18 000 B będzie składać się z dwóch bloków po 8 KB i jednego fragmentu wielkości 2 KB (który może nie być całkowicie zapelniony).

Rozmiary *bloku* i *fragmentu* są ustalane przy tworzeniu systemu plików, zgodnie z jego przewidywanym zastosowaniem. Jeśli zakłada się występowanie wielkiej liczby małych plików, to rozmiar fragmentu powinien być mały. Jeśli oczekuje się wielu transmisji wielkich plików, to rozmiar podstawowego bloku powinien być duży. Szczegóły implementacji wymuszają maksymalny stosunek bloku do fragmentu jako 8:1 oraz minimalny rozmiar bloku równy 4 KB. Tak więc typowy wybór dla pierwszej sytuacji może wynosić 4096:512, a dla drugiej – 8192:1024.

Załóżmy, że dane są pisane do pliku porcjami wielkości 1 KB, a rozmiary bloku i fragmentu w systemie plików wynoszą odpowiednio 4 KB i 512 B. Na dane przesypane za pierwszym razem system plików przydzieli fragment o wielkości 1 KB. Następne przesłanie spowoduje przydzielenie nowego fragmentu o wielkości 2 KB. Dane z pierwszego fragmentu będą musiały zostać przekopiowane do tego nowego fragmentu, a po nich zostanie dopisana druga porcja wynosząca 1 KB. Procedury przydziału będą usuływały znaleźć potrzebne miejsce na dysku bezpośrednio po zapisanym tam już fragmencie, aby nie trzeba było wykonywać żadnego kopирования, lecz jeśli okaże się to niemożliwe, to może się zdarzyć, że trzeba będzie wykonać aż siedem kopii, zanim fragment stanie się blokiem. Z opisanych powodów jest zalecane, by programy same rozstrzygały o rozmiarze bloku dla pliku, tak aby unikać przekopiowywania fragmentów przesyłanych plików.

21.7.2 I-węzły

Plik jest reprezentowany przez *i-węzeł* (zob. rys. 11.7). I-węzeł jest rekordem, w którym przechowuje się większość informacji o konkretnym pliku na dysku. Oryginalna (uniwersalna) nazwa i-węzła – *inode* – (wymawiana jako „aj-noud”) wywodzi się od *index node*, czyli „węzeł indeksu”, i była początkowo zapisywana jako „*i-node*” – po latach łącznik wyszedł z użycia. Termin ten jest też czasem zapisywany jako *I node*.

I-węzeł zawiera identyfikatory użytkownika oraz grupy użytkowników pliku, czasy ostatniej modyfikacji i ostatniego dostępu do pliku, licznik trwałych dowiązań do pliku (wpisów katalogowych) oraz typ pliku (plik zwykły, katalog, dowiązanie symboliczne, urządzenie znakowe, urządzenie blokowe lub gniazdo). W i-węźle jest ponadto miejsce na 15 wskaźników do bloków dyskowych z danymi pliku. Pierwszych 12 z tych wskaźników pokazuje na *bloki bezpośredni*; wskaźniki te zawierają adresy bloków z danymi pliku.

Zatem do danych z małych plików (mających nie więcej niż 12 bloków) można się odnosić bezpośrednio, ponieważ po otwarciu pliku kopia i-węzła jest przechowywana w pamięci głównej. Jeśli rozmiar bloku wynosi 4 KB, to do 48 KB danych można mieć dostęp bezpośredni za pomocą i-węzła.

Następne trzy wskaźniki w i-węźle wskazują na *bloki pośrednie*. Jeśli plik jest tak duży, że trzeba używać bloków pośrednich, to każdy z tych bloków ma długość bloku podstawowego; bloki-fragmenty stosuje się tylko w odniesieniu do bloków danych. Pierwszy wskaźnik bloku pośredniego jest *adresem bloku jednoposredniego* (ang. *single indirect block*). Blok jednoposredni jest blokiem indeksowym nie zawierającym danych, lecz adresy bloków z danymi. Dalej występuje wskaźnik *bloku dwuposredniego* (ang. *double indirect block*); adresuje on blok zawierający adresy bloków ze wskaźnikami do faktycznych bloków danych. Ostatni wskaźnik zawierałby adres *bloku trójposredniego* (ang. *triple indirect block*), jednak nie ma takiej potrzeby. Minimalny rozmiar bloku w systemie plików 4.2BSD wynosi 4 KB, zatem pliki o rozmiarach do 2^{32} B będą używały tylko podwójnych, a nie potrójnych adresów pośrednich. Wynika to z tego, że każdy wskaźnik bloku ma 4 bajty, mamy więc 49 152 bajty dostępu w blokach bezpośrednich, do 4 194 304 B można mieć dostęp za pomocą pojedynczego pośrednictwa, a przez podwójne pośrednictwo osiąga się 4 294 967 296 B, co daje łącznie 4 299 210 752 B, a to jest więcej niż 2^{32} B. Liczba 2^{32} jest ważna, gdyż odległość od początku pliku w strukturze pliku w pamięci operacyjnej jest przechowywana w 32-bitowym słowie. Dlatego pliki nie mogą mieć więcej niż 2^{32} B. Ponieważ wskaźniki plikowe są liczbami całkowitymi ze znakiem (aby w pliku było możliwe szukanie w przód i wstecz), więc rzeczywisty, maksymalny rozmiar pliku wynosi 2^{32-1} . Dwa gigabajty wystarczają w większości zastosowań.

21.7.3 Katalogi

Na omawianym poziomie implementacji nie ma różnicy między plikami a katalogami. Katalogi są przechowywane w blokach danych i reprezentowane przez i-węzły – tak samo jak zwykłe pliki. Od zwykłego pliku katalog odróżnia tylko pole typu w i-węźle. Jednak o zwykłych plikach nie zakłada się, że mają jakąś strukturę, podczas gdy struktura katalogów jest w systemie określona. W wersji 7 systemu UNIX nazwy plików były ograniczone do 14 znaków, katalogi były więc wykazami złożonymi z 16-bajtowych elementów: 2 bajty na numer i-węzła i 14 bajtów na nazwę pliku.

W systemie 4.3BSD nazwy plików są zmiennej długości (do 255 B), więc wpisy katalogowe są również zmiennej długości. Każdy wpis zawiera najpierw swoją długość, następnie nazwę pliku oraz numer i-węzła. Zmienna długość komplikuje nieco zarządzanie katalogiem i procedury przeszukujące,

lecz wyraźnie polepsza możliwości wybierania przez użytkowników nazw plików i katalogów, praktycznie nie ograniczając długości nazwy.

Pierwsze dwie nazwy w każdym katalogu mają postać „.” i „...”. Nowe pozycje katalogowe dodają się do katalogu na pierwszych wolnych miejscach, na ogół za wpisami istniejących plików. Stosuje się przeszukiwanie liniowe.

Użytkownik odwołuje się do pliku za pomocą nazwy ścieżki, natomiast system plików jako definicji plików używa i-węzłów. Jądro musi zatem odwzorowywać dostarczoną przez użytkownika nazwę ścieżki na i-węzeł. Do tego odwzorowywania służą katalogi.

Najpierw należy określić katalog początkowy. Jeśli pierwszym znakiem nazwy ścieżki jest /, to jako katalog początkowy obiera się katalog główny. Jeśli nazwa ścieżki zaczyna się od dowolnego znaku różnego od ukośnej kreski, to katalogiem początkowym będzie bieżący katalog danego procesu. Sprawdza się, czy katalog początkowy istnieje i jest plikiem odpowiedniego typu oraz czy dostęp do niego jest dozwolony – w razie potrzeby generuje się sygnał błędu. I-węzeł katalogu początkowego jest zawsze dostępny.

Następny element nazwy ścieżki, zakończony kolejnym znakiem / lub końcem nazwy ścieżki, jest nazwą pliku. Nazwy tej szuka się w katalogu początkowym i w przypadku jej braku – sygnalizuje błąd. Jeśli w nazwie ścieżki jest jeszcze następny element, to i-węzeł odnalezionej pliku musi odnosić się do katalogu. Jeśli tak nie jest lub jeśli dostęp jest zabroniony, to powstanie sygnału błędu. Nowy katalog jest przeszukiwany tak jak poprzedni. Takie postępowanie trwa aż do osiągnięcia końca nazwy ścieżki i przekazania potrzebnego i-węzła. Ten etapowy proces jest konieczny, ponieważ w dowolnym katalogu można napotkać punkt zamontowania* (lub dowiezanie symbolicne – zob. niżej), co powoduje, że dalsze tłumaczenie przenosi się do innej struktury katalogowej.

Twarde dowiezania są po prostu wpisami katalogowymi, jak inne. Obsługa dowiezań symbolicznych sprawdza się w większej części do zapoczątkowania przeszukiwania od nazwy ścieżki określonej przez zawartość dowiezania symbolicznego. Aby zapobiec powstawaniu nieskończonych pętli, liczy się dowiezania symboliczne napotykane podczas przeszukiwania nazw ścieżek i sygnalizuje błąd, gdy zostanie przekroczona ich górná granica (osiem).

Pliki niedyskowe (takie jak urządzenia) nie mają na dysku przydzielonych bloków danych. Jądro wykrywa typy tych plików (pamiętane w i-węzłach) i wywołuje odpowiednie moduły sterujące obsługą operacji wejścia-wyjścia odnoszących się do tych plików.

Po odnalezieniu i-węzła – na przykład wskutek wykonania funkcji systemowej open – zaktualizowana jest struktura pliku, w której zapamiętuje się wskaźnik

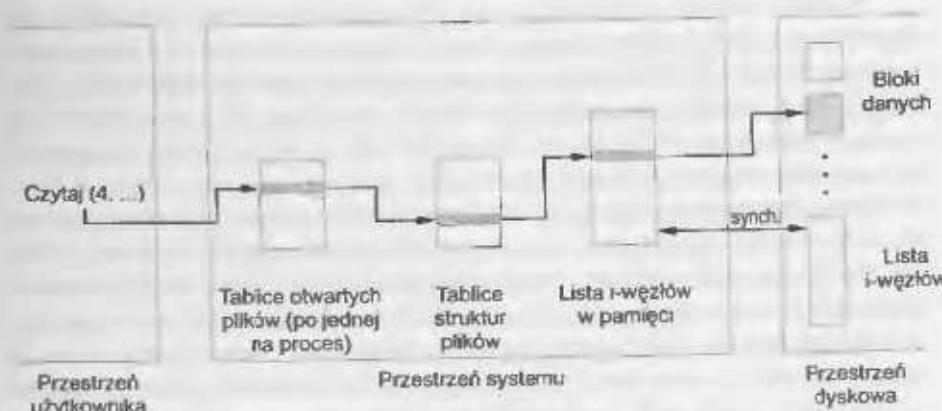
* Innego systemu plików. – Przyp. tłum.

do i-węzła pliku. Do tej struktury odnosi się deskryptor pliku, który przekazuje się użytkownikowi. System 4.3BSD uzupełniono o *pamięć podręczną nazw katalogów*, aby utrzymywać w niej wyniki ostatnich tłumaczeń nazw katalogów na i-węzły. Ulepszenie to spowodowało znaczne zwiększenie wydajności systemu plików.

21.7.4 Odwzorowanie deskryptora pliku na i-węzeł

Wywołania systemowe, które odnoszą się do otwartych plików, wskazują pliki za pomocą plików przekazywanych w ich argumentach deskryptorów. Deskryptor pliku jest używany przez jądro do indeksowania tablicy otwartych plików bieżącego procesu. Każdy element tej tablicy zawiera wskaźnik do struktury pliku. Ta z kolei struktura wskazuje na i-węzeł (rys. 21.7). Tablica otwartych plików ma ustaloną długość, możliwą do określenia tylko podczas rozruchu systemu. Liczba plików jednocześnie otwartych w systemie jest zatem ograniczona.

W argumentach wywołań systemowych *read* i *write* nie podaje się położenia w obrębie pliku. Zamiast tego w jądrze jest przechowywana *odległość w pliku* (ang. *file offset*), odpowiednio aktualniana po każdej operacji czytania lub pisania, stosownie do liczby rzeczywiście przesłanych danych. Za pomocą funkcji systemowej *Iseek* odległość tę można określić jawnie. Jeśli zamiast wskaźników do plików deskryptor pliku indeksował tablicę wskaźników do i-węzłów, to odległość ta musiałaby być przechowywana w i-węźle. Ponieważ jest możliwe otwarcie tego samego pliku przez więcej niż jeden proces, a każdy taki proces musi korzystać z własnej odległości w danym pliku, więc przechowywanie jej w i-węźle byłoby nieodpowiednie. Dlatego do pamiętania odległości w pliku używa się struktury pliku.



Rys. 21.7 Bloki kontrolne systemu plików

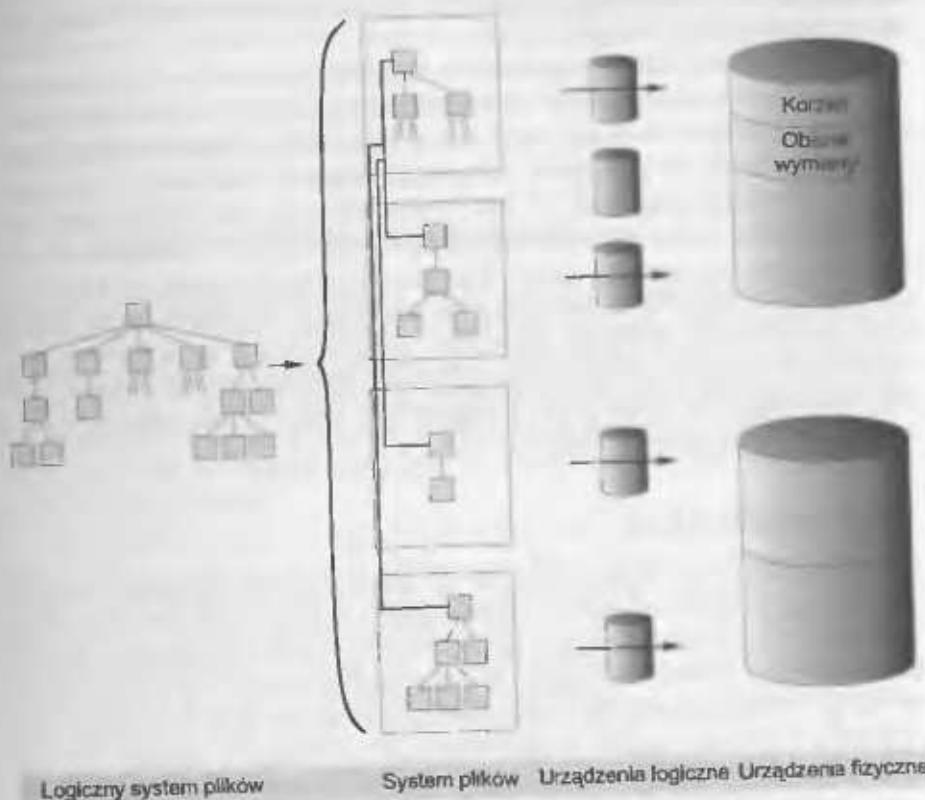
Struktury plików są dziedziczone przez procesy potomne utworzone przez funkcję `fork`, tak więc kilka procesów może posługiwać się *tą samą* odlegością w pliku.

Struktura i-węzła wskazywana przez strukturę pliku jest przechowywaną w pamięci operacyjnej kopią i-węzła z dysku i jest pamiętaana w tablicy o stałej długości. I-węzeł w pamięci operacyjnej ma kilka dodatkowych pól, w tym licznik wskazujących na niego struktur plików. Struktura pliku ma podobny licznik określający, ile odnosi się do niej deskryptorów plików. Jeśli któryś licznik przyjmuje wartość zero, to dany wpis przestaje być potrzebny, a zajmowane przez niego miejsce może być ponownie użyte.

21.7.5 Struktury dysków

System plików taki, jakim widzi go użytkownik, jest utworzony z danych przechowywanych w urządzeniu pamięci masowej, zazwyczaj dysku. Użytkownik wie na ogół tylko o jednym systemie plików, lecz ten logiczny system może się w rzeczywistości składać z kilku *fizycznych* systemów plików, z których każdy znajduje się w innym urządzeniu. Ponieważ urządzenia różnią się między sobą parametrami technicznymi, każde oddzielne urządzenie definiuje własny, fizyczny system plików. W rzeczywistości trzeba także dokonywać podziału wielkich urządzeń fizycznych, takich jak dyski, na wiele urządzeń *logicznych*. Każde urządzenie logiczne definiuje pewien fizyczny system plików. Na rysunku 21.8 widać podział struktury katalogowej na systemy plików, które są odwzorowywane na urządzenia logiczne będące częściami urządzeń fizycznych. We wcześniejszych systemach rozmiary i lokalizacja tych części były zakodowane w modułach sterujących urządzeniami, ale w wersji 4.3BSD są utrzymywane na dysku.

Podział urządzenia fizycznego na wiele systemów plików ma kilka zalet. Różne systemy plików mogą służyć do różnych celów. Chociaż większość stref dyskowych będzie używanych przez systemy plików, co najmniej jedną należy przeznaczyć na obszar wymiany dla oprogramowania pamięci wirtualnej. Polepsza się niezawodność systemu, gdyż uszkodzenia programowe są w zasadzie ograniczone do jednego systemu plików. Odpowiednio ustawiając w każdej ze stref parametry systemu plików (takie jak długości bloku i fragmentu), można zwiększyć wydajność. Stosowanie oddzielnych systemów plików chroni także przed zużyciem przez jeden program całej dostępnej przestrzeni na wielki plik, ponieważ pliki nie mogą być podzielone między różne systemy plików. Należy też dodać, że składowanie dysku odnosi się do strefy, więc łatwiej jest poszukiwać pliku na taśmie składowania, jeśli strefa jest mniejsza. Odtwarzanie całej strefy z taśmy również trwa krócej.



Rys. 21.8 Odwzorowanie logicznego systemu plików na urządzenia fizyczne

Faktyczna liczba systemów plików na jednym napędzie zmienia się w zależności od rozmiaru dysku i przeznaczenia systemu komputerowego jako całości. Jeden system plików – główny (ang. *root*)^{*} – jest zawsze dostępny. Inne mogą być *montowane*, tzn. integrowane z hierarchią katalogów głównego systemu plików.

Na zamontowanie systemu plików w jakimś i-węźle wskazuje bit w strukturze tego i-węzła. Odwołanie do takiego pliku powoduje przeszukanie tablicy montażu w celu znalezienia numeru zamontowanego urządzenia. Numer urządzenia służy do odnalezienia i-węzła katalogu głównego zamontowanego systemu plików, po czym używa się tego i-węzła. I na odwrót, jeśli element nazwy ścieżki ma postać „...” oraz przeszukiwanym katalogiem jest katalog główny zamontowanego systemu plików, to przeszukuje się tablicę

* Czyli „korzeniowy system plików”. – Przyp. tłum.

montaży w celu odnalezienia i-węzła, w którym jest on zamontowany, po czym używa się tego i-węzła.

Każdy system plików jest osobnym zasobem systemowym i reprezentuje zbiór plików. Pierwszy sektor na urządzeniu logicznym jest *blokiem rozruchowym*, w którym może się znajdować podstawowy program ładowający, którego można użyć do wywołania następnego programu ładowającego, rezydującego w obszarze następnych 7,5 KB. System wymaga tylko jednej strefy zawierającej dane bloku rozruchowego, niemniej jednak jego administrator może, za pomocą programów uprzywilejowanych, utworzyć kopie bloku rozruchowego, aby umożliwić uruchomienie systemu w przypadku uszkodzenia kopii podstawowej. Statyczne parametry systemu plików są zawarte w tzw. *superbloku*. Parametry te określają całkowity rozmiar systemu plików, rozmiary pełnych bloków danych i ich fragmentów oraz wybrane dane dotyczące zasad wykonywania przydziałów.

21.7.6 Implementacje

Interfejs użytkownika systemu plików jest prosty i dobrze określony, umożliwia więc wykonywanie zmian w implementacji systemu plików bez istotnego wpływu na użytkownika. System plików wersji 7 został zmieniony w stosunku do wersji 6, zmiany występują również między wersją 7 a wersją 4BSD. W wersji 7 rozmiar i-węzłów został podwojony, wzrosły maksymalne rozmiary plików i systemu plików, a także zmieniły się szczegóły obsługi list wolnych miejsc i informacji w superbloku. W tym czasie również funkcję *seek* (z 16-bitową odlegością) zastąpiono funkcją *lseek* (z odległością 32-bitową), aby umożliwić określanie odległości w większych plikach, lecz zmiany wiadoczone na zewnątrz jądra były niewielkie.

W wersji 4.0BSD powiększono rozmiar bloków w systemie plików z 512 do 1024 B. Choć powiększenie rozmiaru bloków spowodowało zwiększenie wewnętrznej fragmentacji na dysku, jednak podwoiła się przepustowość, głównie dzięki większej ilości danych wysyłanych za jednym razem. Ten oraz inne pomysły, a także wiele modułów obsługi urządzeń i programów wdrożono później w Systemie V.

Wersję 4.2BSD wzbogacono o szybki system plików (*Berkeley Fast File System*), szybszy i wyposażony w nowe właściwości. Dowiązania symboliczne wymagały wprowadzenia nowych funkcji systemowych. Długie nazwy plików spowodowały konieczność użycia nowych funkcji systemowych odnoszących się do katalogów, umożliwiających obchód złożonej obecnie, wewnętrznej struktury katalogowej. Dodano także wywołania *truncate*^{*}. Szybki

* Służące do obcinania pliku. – Przyp. tłum.

system plików odniósł sukces i obecnie można go odnaleźć w większości implementacji systemu UNIX. Jego wydajność osiągnięto dzięki jego rozplanowaniu i zasadom przydziału, które omówimy poniżej. W punkcie 11.2.4 omówiliśmy zmiany poczynione w systemie SunOS w celu dalszego zwiększenia przepustowości dysku.

21.7.7 Rozplanowanie i zasady przydziału

Jądro do identyfikacji pliku uzywa pary *<numer urządzenia logicznego, numer i-węzła>*. Numer urządzenia logicznego określa zastosowany system plików. I-węzły w systemie plików są ponumerowane kolejno. W systemie plików wersji 7 systemu wszystkie i-węzły znajdują się w tablicy występującej bezpośrednio po pojedynczym superbloku na początku urządzenia logicznego, a po nich występują bloki danych. *Numer i-węzła* jest więc po prostu indeksem do tej tablicy.

W systemie plików wersji 7 blok pliku może znajdować się na dysku gdziekolwiek między końcem tablicy i-węzłów a końcem systemu plików. Wolne bloki są powiązane w listę przechowywaną w superbloku. Bloki są dodawane na początku listy wolnych bloków i w miarę potrzeb również z jej początku zabierane do obsłużenia nowych plików lub rozszerzenia plików istniejących. Tak więc bloki pliku mogą znajdować się w dowolnych odległościach zarówno od i-węzła, jak i wzajemnie od siebie. Co więcej, im częściej system plików tego rodzaju jest używany, tym większy nieporządek pojawia się w rozmieszczeniu bloków pliku. Odwrócić ten proces można tylko przez ponowne zainicjowanie i zapamiętanie całego systemu plików, co nie jest łatwym zadaniem. Proces ten jest opisany w p. 11.6.2.

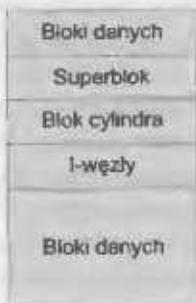
Druga trudność wiąże się z wątpliwą niezawodnością takiego systemu plików. Ze względu na szybkość działania superblok każdego z zamontowanych systemów plików jest przechowywany w pamięci operacyjnej. Pozwala to jądru na szybkie dostępy do superblok, zwłaszcza przy posługiwaniu się listą wolnych bloków. Co każde 30 s superblok jest przepisywany na dysk, aby zapewnić synchronizację między kopiami w pamięci i na dysku (za pomocą programu aktualizującego, użyciowego funkcji systemowej *sync*). Jednakże nierzadko się zdarza, że awaria systemu spowodowana błędami oprogramowania lub sprzętu niszczy przechowywany w pamięci superblok, zanim nastąpi uaktualnienie dysku. Wówczas lista wolnych bloków na dysku nie odpowiada dokładnie stanowi dysku. W celu jej zrekonstruowania należy wykonać długotrwałe sprawdzanie wszystkich bloków systemu plików. Zwróciły uwagę na to, że problem ten dotyczy też nowego systemu plików.

Implementacja systemu plików 4.2BSD jest całkowicie odmienna od zawartej w wersji 7. Powtórną implementację wykonano przede wszystkim w celu poprawienia wydajności i odporności – większość tych zmian jest niewidocznych na zewnątrz jądra. W tym samym czasie wprowadzono pewne inne zmiany, takie jak dowiązania symboliczne i długie nazwy plików (do 255 znaków), widoczne zarówno w funkcjach systemowych, jak i na poziomie użytkownika. Większość zmian wynikających z wdrożenia tych cech dotyczyła jednak nie jądra, lecz programów, w których z właściwości tych robiono użytk.

Zasadniczą zmianę wprowadzono w przydiale przestrzeni dyskowej. Podstawową nową koncepcją w systemie 4.2BSD jest *grupa cylindrów*. Grupa cylindrów ma umożliwić lokalizację bloków pliku. Każda grupa cylindrów zajmuje na dysku jeden lub więcej sąsiednich cylindrów, tak że dostęp w obrębie grupy cylindrów wymagał minimalnego ruchu głowic dyskowych. Każda grupa cylindrów ma superblok, blok opisu cylindrów, tablicę i-węzłów i pewną liczbę bloków danych (rys. 21.9).

Superblok jest identyczny w każdej grupie cylindrów, tak że w przypadku uszkodzenia dysku można go odzyskać z dowolnej z grup. Blok opisu cylindrów (ang. *cylinder block*) zawiera dynamiczne parametry konkretnej grupy cylindrów. Znajduje się w nim mapa bitowa wolnych bloków i fragmentów danych oraz mapa bitowa wolnych i-węzłów. Przechowuje się tutaj także dane statystyczne dotyczące ostatnich rezultatów stosowanych strategii przydzielu.

Informacja nagłówkowa dla grupy cylindrów (superblok, blok opisu cylindrów oraz i-węzły) nie zawsze znajduje się na początku grupy cylindrów. Jeśli tak było, to informacja nagłówkowa dla każdej grupy cylindrów mogłaby znajdować się na tej samej płycie dysku. Uszkodzenie jednej głowicy dyskowej mogłoby wówczas spowodować usunięcie wszystkich informacji. Dlatego każda grupa cylindrów ma informacje nagłówkowe umieszczone w innej odległości od początku grupy.



Rys. 21.9 Grupa cylindrów systemu 4.3BSD

Polecenie *ls*, służące do wyprowadzania katalogu, bardzo często musi czytać i-węzły wszystkich plików w katalogu, byłoby więc uzasadnione, aby takie i-węzły były ulokowane obok siebie na dysku. Z tego powodu i-węzeł pliku lokalizuje się zwykle w tej samej grupie cylindrów co i-węzeł macierzystego katalogu pliku. Nie wszystko jednak da się zlokalizować w ten sposób, więc i-węzeł nowego katalogu umieszcza się w innej grupie cylindrów niż jego katalog macierzysty. Na i-węzeł takiego nowego katalogu wybiera się grupę cylindrów, która ma największą liczbę nie wykorzystanych i-węzłów.

Aby zmniejszyć liczbę przesunięć głowicy dyskowej przy dostępie do bloków danych pliku, przydziela się bloki z obrębu tej samej grupy cylindrów tak często, jak to tylko jest możliwe. Ponieważ nie pozwala się, aby jeden plik zajął wszystkie bloki w grupie cylindrów, więc plikowi przekraczającemu pewien rozmiar (np. 2 MB) przydziela się dalsze bloki w innej grupie cylindrów. Nową grupę cylindrów wybiera się z tych, które mają więcej niż średnio wolnego miejsca. Jeśli plik nadal się powiększa, to następuje kolejna zmiana przydziału (po każdym megabajcie) do jeszcze innej grupy cylindrów. W ten sposób wszystkie bloki małego pliku mają szansę znaleźć się w tej samej grupie cylindrów, a liczba przemieszczeń głowicy dyskowej przy dostępie do wielkiego pliku jest nadal mała.

Są dwa poziomy procedur przydzielania bloków dyskowych. Procedury realizujące politykę globalną wybierają potrzebny blok dyskowy w sposób omówiony powyżej. Procedury polityki lokalnej na podstawie specyficznych informacji zapisanych w blokach opisu cylindrów starają się wybrać blok położony najbliżej zamawianego. Jeśli zamawiany blok nie jest używany, to będzie przekazany przez procedurę. W przeciwnym razie będzie przekazany blok rotacyjnie najbliższy zamawianemu na tym samym cylindrze lub blok z innego cylindra, lecz w tej samej grupie cylindrów. Jeśli w grupie cylindrów nie ma więcej bloków, to w celu znalezienia bloku oblicza się wartości funkcji haszującej drugiego stopnia w odniesieniu do pozostałych grup cylindrów. Gdy i to nie przyniesie rezultatu, wówczas szuka się bloku krok za krokiem. Jeśli w systemie plików jest wystarczająco dużo wolnej przestrzeni (zazwyczaj 10%), to bloki na ogół odnajduje się w pożądanych miejscach, więc nie są potrzebne ani funkcje haszującą, ani przeszukiwanie wszystkich bloków po kolej, a wydajność systemu nie maleje w miarę użytkowania.

Dzięki zwiększonej wydajności szybkiego systemu plików FFS typowe dyski są obecnie wykorzystywane na poziomie 30% ich technicznej przepustowości. Ta liczba oznacza istotną poprawę, zważywszy, że system plików w wersji 7 systemu operacyjnego zużywał tylko 3% szerokości pasma przesyłania danych.

W systemie 4.3BSD Tahoe wprowadzono nowy szybki system plików – Fat Fast File System, który przy tworzeniu systemu plików pozwala określić liczbę i-węzłów przypadających na grupę cylindrów, liczbę cylindrów w grupie cylindrów oraz liczbę rozróżnialnych pozycji obrotowych. System 4.3BSD określał te dane na podstawie parametrów sprzętowych dysku.

21.8 ■ System wejścia-wyjścia

Jednym z zadań systemu operacyjnego jest ukrywanie osobliwości i specyfiki urządzeń sprzętowych przed okiem użytkownika. Na przykład system plików przedstawia prostą i zwartą koncepcję przechowywania informacji (plik), niezależną od stosowanego sprzętu dyskowego. W systemie UNIX większość właściwości urządzeń wejścia-wyjścia jest ukryta także przed samym jądrem, a to za sprawą systemu wejścia-wyjścia. *System wejścia-wyjścia* składa się z zespołu podręcznych buforów, ogólnego kodu sterującego pracą urządzeń i modułów sterujących konkretnymi urządzeniami sprzętowymi. Specyfikę danego urządzenia zna tylko jego moduł sterujący. Główne części systemu wejścia-wyjścia są przedstawione na rys. 21.10.

W systemie 4.3BSD są trzy główne rodzaje wejścia-wyjścia: urządzenia blokowe, urządzenia znakowe oraz interfejs gniazd. Interfejs gniazd wraz z jego protokołami i interfejsami sieciowymi omówimy w p. 21.9.1.

Urządzenia blokowe obejmują dyski i taśmy. Wyróżniającą je cechą jest możliwość bezpośredniego ich adresowania blokami o stałej długości, zwykle 512 B. Moduł obsługi urządzenia blokowego jest potrzebny do odizolowania szczegółów związanych ze ścieżkami, cylindrami itd. od reszty jądra. Urządzenia blokowe są dostępne bezpośrednio za pomocą odpowiednich plików specjalnych (takich jak `/dev/rp0`), lecz częściej korzysta się z nich pośrednio, za pomocą systemu plików. W każdym przypadku transmisje są buforowane przez pamięć podręczną buforów blokowych, która ma zasadnicze znaczenie dla wydajności.

Interfejs systemowych odwołań do jądra					
Gniazdo	Plik zwykły	Opracowany interfejs blokowy	Surowy interfejs blokowy	Surowy interfejs terminalu	Terminal opracowany
Protokoły	System plików				Reżim linii
Interfejs sieciowy	Moduł sterujący urządzeniem blokowym			Moduł sterujący urządzeniem znakowym	
Serw!					

Rys. 21.10 Struktura wejścia-wyjścia w jądrze systemu 4.3BSD

Do urządzeń znakowych należą terminale, i drukarki wierszowe, lecz także prawie wszystkie inne urządzenia (z wyjątkiem interfejsów sieciowych), które nie buforują podręcznie bloków. Na przykład `/dev/mem` jest interfejsem do fizycznej pamięci głównej, a `/dev/null` jest nieskończonym ujściem danych i nieskończonym źródłem znaczników końca pliku. Pewne urządzenia, takie jak szybkie interfejsy graficzne, mogą mieć własne bufore, lub mogą zawsze wykonywać operacje wejścia-wyjścia wprost w obszarze danych użytkownika. Ponieważ nie używają one podręcznych buforów bloków, są sklasyfikowane jako urządzenia znakowe.

W terminalach i urządzeniach podobnych do terminali znajdują zastosowanie *C-listy* (ang. *C-lists, character lists*), które są buforami mniejszymi niż te, które zawiera pamięć podręczna buforów blokowych.

Urządzenia blokowe i znakowe są dwiema podstawowymi klasami urządzeń. Dostęp do modułów sterujących urządzeniami odbywa się poprzez jedną z dwóch tablic punktów wejściowych. Jedna tabela służy urządzeniom blokowym, druga jest przeznaczona dla urządzeń znakowych. Urządzenia są rozróżniane na podstawie ich klas (blokowe lub znakowe) oraz numerów urządzeń. Numer urządzenia składa się z dwóch części. Główny (ang. *major*) numer urządzenia jest używany do indeksowania tablicy urządzeń znakowych lub blokowych w celu odnalezienia wpisów dotyczących odpowiedniego modułu sterującego. Pomocniczy (ang. *minor*) numer urządzenia jest interpretowany przez moduł obsługi urządzenia jako na przykład logiczna strefa dysku lub linia terminalu.

Moduł obsługi urządzenia łączy z resztą jądra tylko jego punkty wejściowe zapisane w tablicy odpowiadającej jego klasie oraz używane przez niego wspólne systemy buforowania. To odseparowanie jest ważne ze względu na przenośność, jak również konfigurowanie systemu.

21.8.1 Podręczne buforowanie bloków

Urządzenia blokowe korzystają z podręcznego buforowania bloków. Pamięć podręczna buforów blokowych mieści pewną liczbę nagłówków buforów, z których każdy wskazuje na fragment pamięci fizycznej oraz numer urządzenia i numer bloku na urządzeniu. Nagłówki buforów dla bloków nie będących aktualnie w użyciu są przechowywane na kilku powiązanych listach, z których każda jest przezraczona na:

- bufore, używane ostatnio, ustawione na liście w porządku od najmłodszego do najstarszego (lista LRU);
- bufore, dawno nie używane lub z unieważnioną zawartością (lista wieku – AGE);
- puste bufore, nie związane z żadną pamięcią fizyczną.

W celu usprawnienia wyszukiwania buforów na listach stosuje się funkcje haszujące według numerów urządzeń i bloków.

Kiedy jest potrzebny blok z jakiegoś urządzenia (czytanie), wtedy przegląda się pamięć podręczna. Jeśli blok zostanie w niej znaleziony, to używa się go bez potrzeby wykonywania operacji wejścia-wyjścia. Jeśli nie ma go w pamięci podręcznej, to wybiera się bufor z listy AGE lub z listy LRU (jeśli lista AGE jest pusta). Wówczas aktualnia się przyporządkowany buforowi numer urządzenia i bloku, w razie konieczności znajduje się dla niego miejsce w pamięci i przesyła do niego dane z urządzenia. Jeśli nie ma wolnych buforów, to bufor określony przez porządek LRU zostaje odesłany do swojego urządzenia (jeśli został zmieniony), po czym używa się go ponownie.

Przy pisaniu, jeśli poszukiwany blok jest już w podręcznym buforze, nowe dane są umieszczane w tym buforze (zastępując jego poprzednią zawartość), w nagłówku bufora zaznacza się, że bufor został zmieniony i nie trzeba wykonywać natychmiast operacji wejścia-wyjścia. Dane będą zapisane w urządzeniu, kiedy bufor będzie potrzebny do innych celów. Jeśli bloku nie odnaleziono w pamięci podręcznej, to wybiera się pusty bufor (jak przy czytaniu) i przesyła dane do tego bufora.

Okresowo wymusza się zapisanie na dysku zabrudzonych (czyli zmienionych) buforów pamięci podręcznej, aby zminimalizować niespójności systemu plików po ewentualnej awarii.

Liczba danych w buforze jest w systemie 4.3BSD zmienna, aż do pewnego maksimum dotyczącego wszystkich systemów plików – zwykle 8 KB*. Minimalny ich rozmiar równa się rozmiarowi grona przy stronowaniu (ang. *paging-cluster*) i na ogół wynosi 1024 B. Bufory zaczynają się i kończą na granicach grona stron, a dowolne grono stron może być za każdym razem odwzorowane tylko na jeden bufor, podobnie jak w przypadku bloków dyskowych – każdy z nich może być za każdym razem odwzorowany tylko na jeden bufor. Lista pustych buforów zawiera nagłówki buforów używanych wówczas, gdy fizyczny, osmikilobajtowy blok pamięci jest podzielony na wiele mniejszych bloków. Bloki takie muszą mieć nagłówki i są odzyskiwane z listy pustych buforów.

Ilość danych w buforze może się zwiększać w miarę zapisywania przez proces użytkownika kolejnych danych po tych, które już znajdują się w buforze. Wzrost ilości danych spowoduje przydzielenie większego bufora, wystarczającego dużego, aby je wszystkie pomieścić. Następnie zostaną do niego skopiowane dane przebywające w buforze na początku, a po nich nowa porcja

* W systemie 4.4BSD maksymalna długość bufora w pamięci wirtualnej wynosi 64 KB.
– Przyp. tłum.

danych. Jeśli zaczyna brakować miejsca w buforze, to z kolejki pustych buforów zabiera się nowy bufor i umieszcza w nim nadmiarowe strony; dane z poprzedniego bufora zapisuje się na dysku i zwalnia bufor.

Niektóre urządzenia, takie jak taśmy magnetyczne, wymagają zapisywania bloków w pewnym porządku. W odniesieniu do takich urządzeń istnieją więc możliwości wymuszania synchronicznego pisania z buforów, z zachowaniem właściwego porządku. Bloki katalogów są również zapisywane synchronicznie, aby nie dopuścić do powstania niespójności w przypadku awarii. Wyobraźmy sobie chaos, do jakiego doszłoby, gdyby w katalogu wykonano wiele zmian, lecz nie zdolano wykonać odpowiednich aktualizacji samych wpisów katalogowych.

Rozmiar pamięci podrzędnej buforów może mieć decydujący wpływ na wydajność systemu, ponieważ dostateczna jej wielkość warunkuje wysoki procent odnajdywania w niej danych i zmniejszenie liczby rzeczywistych transmisji wejścia-wyjścia.

Interesujące jest współdziałanie między pamięcią podrzęczną buforów, systemem plików i modułami obsługi dysków. Dane zapisywane do pliku dyskowego są gromadzone w pamięci podrzęcznej, a moduł obsługi dysku sortuje je i tworzy kolejkę wyjściową według adresów dyskowych. Dzięki temu sterownik dysku może minimalizować ruchy głowic dyskowych i zapisywać dane w chwilach optymalnych ze względu na obroty dysku. Jeżeli tylko nie jest potrzebne pisanie synchroniczne, to proces mający coś do zapisania na dysku po prostu pisze do bufora podrzędnego, a system asynchronicznie, w dogodnej chwili zapisze dane z bufora na dysku. Dla procesu użytkownika pisanie jest czynnością bardzo szybką. Przy czytaniu danych z pliku dyskowego system blokowego wejścia-wyjścia wykonuje operacje czytania z pewnym wyprzedzeniem, jednak czytanie jest znacznie mniej asynchroniczne niż pisanie. Wskutek tego wyjście na dysk za pośrednictwem systemu plików jest często – wbrew wyobrażeniom – szybsze niż wejście w przypadku dużych porcji przesyłanych danych.

21.8.2 Surowe interfejsy urządzeń

Prawie każde urządzenie blokowe ma też interfejs znakowy. Takie interfejsy są nazywane *interfejsami surowego wejścia-wyjścia* (ang. *raw device interface*). Różnią się one od interfejsów blokowych tym, że pomija się w nich buforowanie podrzęczne.

Każdy moduł sterujący dyskiem utrzymuje kolejkę nie obsłużonych do końca operacji wejścia-wyjścia. Każdy rekord tej kolejki zawiera informację o tym, czy dotyczy pisania czy czytania, adres w pamięci głównej używany podczas przesyłania (zazwyczaj numer 512-bajtowego bloku), adres urządzenia

nia biorącego udział w przesyłaniu oraz rozmiar przesyłanego bloku danych (w sektorach). Odwzorowanie informacji pobranej z bufora bloku na informacje wymagane w tej kolejce nie nastręcza trudności.

Niemal równie prosto można odwzorować fragment pamięci głównej, odpowiadający części wirtualnej przestrzeni adresowej procesu użytkownika. Odwzorowania takie wykonuje na przykład surowy interfejs dyskowy. Jest zatem dozwolone niebuforowane przesyłanie danych, kierowane wprost do (lub z) wirtualnej przestrzeni adresowej użytkownika. Rozmiary transmisji są ograniczone przez urządzenia fizyczne – niektóre z nich wymagają parzystej liczby bajtów.

Jądro wykonuje operacje wejścia-wyjścia związane z wymianą i stronowaniem, umieszczając po prostu stosowne zamówienie w kolejce do odpowiedniego urządzenia. Nie jest do tego potrzebny żaden specjalny moduł obsługi urządzenia stronicującego lub wymiany.

Implementacja systemu plików 4.2BSD została w istocie napisana i w większości przetestowana jako proces użytkownika, który korzystał z surowego interfejsu dyskowego, zanim jego kod przeniesiono do jądra. Za interesujący zwrot można uznać całkowite usunięcie systemu plików z systemu operacyjnego, czego dokonano w systemie Mach. Systemy plików można realizować jako zadania poziomu użytkowego.

21.8.3 C-listy

W modułach sterujących terminalami stosuje się system buforowania znaków, w którym w postaci listowej utrzymuje się małe bloki znaków (zwykle po 28 B). Istnieją procedury do ustawiania w kolejce i pobierania z kolejki znaków na takich listach. Chociaż wszystkie wolne bufore znakowe znajdują się na jednej liście, większość korzystających z nich modułów obsługi urządzeń ogranicza liczbę znaków, które mogą znaleźć się jednorazowo w kolejce do danej linii terminala.

Systemowa funkcja **write** odniesiona do terminalu powoduje wstawienie znaków na liście do tego urządzenia. Następuje pierwsze przesłanie, a dalsze przesłania i pobrania znaków z kolejki będą sterowane przerwaniami.

Wejście jest podobnie sterowane przerwaniami. Moduły obsługi terminali na ogół utrzymują *dwie* kolejki wejściowe – przejście od pierwszej (surowej) kolejki do drugiej kolejki (kanonicznej) następuje z chwilą, gdy procedura obsługi przerwania dołoży do kolejki surowej znak końca wiersza. Budzi się wtedy proces wykonujący czytanie z urządzenia i w fazie systemowej wykonuje konwersję: następuje umieszczenie znaków w kolejce kanonicznej, z której może je pobrać proces użytkownika za pomocą operacji czytania.

Jest także możliwe pominięcie przez moduł sterujący kolejki kanonicznej i przekazywanie znaków wprost z kolejki surowej. Ten tryb działania nazywa się *surowym* (ang. *raw mode*). W trybie surowym pracują edytory pełnoekranowe i inne programy, które muszą reagować na każde uderzenie klawisza.

21.9 ■ Komunikacja międzyprocesowa

Wiele zadań może być wykonanych przez wyizolowane procesy, lecz wiele innych wymaga komunikacji międzyprocesowej. Izolowane systemy obliczeniowe służyły przez długi czas w wielu zastosowaniach, ale znaczenie pracy sieciowej rośnie nieustannie. Wraz z powiększającym się obszarem zastosowań osobistych stacji roboczych dzielenie zasobów staje się coraz powszechniejsze. Komunikacja międzyprocesowa nie należała do tradycyjnie mocnych punktów systemu UNIX.

Większość systemów UNIX nie zezwalała na dzielenie pamięci, ponieważ sprzęt PDP-11 nie zachęcał do takich przedsięwzięć. System V umożliwia dzielenie pamięci. Zaplanowano ją również w systemie 4.2BSD, lecz nie zrealizowano z powodu ograniczeń czasowych. W systemie Solaris 2 pamięć może być użytkowana wspólnie, podobnie jak w wielu innych, współczesnych wersjach systemu UNIX. W każdym razie korzystanie z pamięci dzielonej jest kłopotliwe w środowisku sieciowym, gdyż dostęp do pamięci w sieci nigdy nie może być tak szybki jak kontakty z pamięcią lokalnej maszyny. Ażkolwiek można próbować dzielenia pamięci między dwiema oddzielnymi maszynami na zasadzie przezroczystego kopiowania danych przycz. to jednak traci się główną korzyść wynikającą z dzielenia pamięci, tzn. szybkość.

21.9.1 Gniazda

Potok (omówiony w p. 21.4.3) jest najbardziej typowym w systemie UNIX mechanizmem komunikacji międzyprocesowej. Potok umożliwia niezawodny, jednokierunkowy przepływ strumienia bajtów między dwoma procesami. Implementuje się go tradycyjnie w postaci zwykłego pliku, z kilkoma wyjątkami. Potok nie ma nazwy w systemie plików, lecz jest tworzony przez funkcję systemową *pipe*. Rozmiar potoku jest ustalony i proces, który próbuje pisać do pełnego potoku, ulega wstrzymaniu. Gdy wszystkie dane, uprzednio zapisane do potoku, zostaną przeczytane, wówczas pisanie będzie kontynuowane na początku pliku (potoki nie są prawdziwymi buforami cyklicznymi). Jedną z zalet małych rozmiarów potoku (zwykle 4096 B) jest rzadko występująca konieczność zapisywania danych z potoku na dysku: na ogół przechodzi się je w zwykłych buforach podręcznych.

W systemie 4.3BSD potoki są zrealizowane jako specjalny przypadek mechanizmu *gniazda*. Mechanizm gniazda pozwala utworzyć ogólny interfejs nie tylko dla potoków, które są lokalne w danej maszynie, lecz także do pracy w sieci. Nawet w przypadku tej samej maszyny z potoku mogą korzystać tylko takie dwa procesy, które są powiązane przez funkcję systemową *fork*. Mechanizm gniazd może być zastosowany do procesów nie powiązanych ze sobą.

Gniazdo jest punktem końcowym komunikacji. Gniazdo będące w użyciu ma zwykle związaną ze sobą *adres*. Specyfika tego adresu zależy od *domeny komunikacyjnej gniazda*. Domenę wyróżnia to, że procesy komunikujące się w tej samej domenie stosują taki sam *format adresu*. Przez pojedyncze gniazdo można się komunikować tylko w jednej domenie.

W systemie 4.3BSD są obecnie zaimplementowane trzy domeny: domena uniksowa (AF_UNIX), domena internetowa (AF_INET) oraz domena usług sieciowych (NS – Network Services) firmy XEROX (AF_NS). Format adresu w domenie uniksowej jest zwyczajną nazwą ścieżki w systemie plików, w rodzaju */alpha/beta/gamma*. Procesy komunikujące się w domenie internetowej stosują protokoły DARPA Internet (takie jak TCP/IP) oraz adresy sieci Internet, które składają się 32-bitowego numeru komputera macierzystego i 32-bitowego numeru portu (reprezentującego punkt spotkania procesów w komputerze sieciowym).

Istnieje kilka *typów gniazd*, które reprezentują klasy usług. Każdy typ gniazda może, lecz nie musi, być zrealizowany w dowolnej domenie komunikacyjnej. Jeśli jakiś typ jest zaimplementowany w danej domenie, to może go implementować jeden lub kilka protokołów, które mogą być wybierane przez użytkownika.

- **Gniazda strumieniowe:** Gniazda tego rodzaju umożliwiają niezawodne, duplexowe przekazywanie sekwencyjnych strumieni danych. Przy przesyłaniu żadne dane nie giną ani nie są podwajane, nie ma również ograniczeń na rekordy. Ten typ występuje w domenie internetowej, gdzie realizuje go protokół TCP. W domenie uniksowej potoki implementują się jako pary komunikujących się gniazd strumieniowych.
- **Gniazda pakietów sekwencyjnych:** Te gniazda tworzą strumienie danych na podobieństwo gniazd strumieniowych, z tym że stosuje się ograniczenia rekordów. Ten typ jest używany w protokole AF_NS usług sieciowych XEROX.
- **Gniazda datagramów:** Te gniazda przesyłają w każdym kierunku komunikaty o zmiennej długości. Nie ma jednak gwarancji, że komunikaty dotrą do celu w tym samym porządku, w którym zostały wysłane, lub ze

nie będą powtarzane, a nawet – że nadjejdą w ogóle. Jeśli natomiast jakiś datagram dotrze do celu, to zawarty w nim komunikat (rekord) będzie miał zachowaną długość. Ten typ gniazd jest zaimplementowany w domenie internetowej przez protokół UDP.

- **Gniazda niezawodnie dostarczanych komunikatów:** Gniazda tego rodzaju gwarantują dostarczenie przesyłanych komunikatów, które poza tym przypominają komunikaty przekazywane za pomocą gniazd datagramów. Obecnie nie implementuje się tego typu gniazd.
- **Gniazda surowe:** Te gniazda umożliwiają procesom bezpośredni dostęp do protokołów implementujących inne typy gniazd. Dostępne są nie tylko protokoły najwyższego poziomu, lecz także protokoły niższego poziomu. Na przykład w domenie internetowej jest możliwe osiągnięcie protokołu TCP oraz protokołów IP lub Ethernet, które leżą poniżej tego protokołu. Ta właściwość jest przydatna przy opracowywaniu nowych protokołów.

Z gniazdem jest związanych kilka specyficznych funkcji systemowych. Funkcja systemowa `socket` tworzy gniazdo. Jako argumenty przyjmuje określenie domeny komunikacyjnej, typ gniazda i protokół, który ma być użyty do danego typu gniazda. Wynikiem tej funkcji jest mała liczba całkowita, zwana *deskryptorem gniazda* (ang. *socket descriptor*), która należy do tej samej przestrzeni nazw, co deskryptory plików. Deskryptor gniazda indeksuje tablicę otwartych „plików” w *u-strukturze* w jądrze i ma przydzieloną strukturę pliku. Struktura pliku w systemie 4.3BSD może wskazywać na gniazdo zamiast na i-węzeł. W tym przypadku pewne informacje o gnieździe (jak typ gniazda, licznik komunikatów oraz dane w jego kolejkach wejściowych i wyjściowych) są pamiętane wprost w strukturze gniazda.

Aby inny proces mógł zaadresować gniazdo, musi ono mieć nazwę. Powiązanie nazwy z gniazdem następuje za pomocą funkcji systemowej `bind`, która pobiera deskryptor gniazda, wskaźnik do nazwy i długość nazwy traktowanej jako ciąg bajtów. Zawartość i długość ciągu bajtów zależy od formatu adresu. Do nawiązania połączenia służy funkcja systemowa `connect`. Składnia jego argumentów jest taka sama, jak składnia argumentów w wywołaniu `bind`. Deskryptor gniazda reprezentuje gniazdo lokalne, a adres dotyczy gniazda obcego, z którym próbuje się nawiązać połączenie.

Wiele procesów komunikujących się za pośrednictwem gniazd IPC (ang. *interprocess communication* – komunikacja międzyprocesowa) działa zgodnie z modelem klient-serwer. W modelu tym proces serwera świadczy usługi dla procesu klienta. Gdy usługa jest dostępna, wówczas proces serwera prowadzi nasłuch pod ogólnie znanym adresem, a proces klienta używa uprzednio opisanej funkcji `connect`, aby nawiązać kontakt z serwerem.

Proces serwera używa funkcji `socket` do utworzenia gniazda i funkcji `bind` do związania z gniazdem ogólnie znanego adresu jego usługi. Następnie wywołuje funkcję systemową `listen`, aby powiadomić jądro, że jest gotowy przyjmować połączenia z klientami, oraz by określić, ile połączeń będących w toku powinno znaleźć miejsce w kolejce organizowanej przez jądro, do czasu gdy usługodawca będzie mógł je obsługiwać. Na koniec serwer używa funkcji systemowej `accept` do przyjmowania indywidualnych połączeń. Zówno wywołania `listen`, jak i `accept` przyjmują jako argument deskryptor odpowiedniego gniazda. Funkcja `accept` przekazuje nowy deskryptor gniazda, odpowiadający nowemu połączeniu. Pierwotny deskryptor gniazda jest wciąż otwarty dla dalszych połączeń. Po operacji `accept` serwer na ogół stosuje funkcję `fork`, aby wytworzyć nowy proces w celu obsługi klienta. Pierwotny proces serwera kontynuuje nasłuchiwanie dalszych połączeń.

Istnieją też funkcje systemowe do określania parametrów połączenia oraz do przekazywania adresu obcego gniazda po wykonaniu funkcji `accept`.

Po nawiązaniu połączenia z gniazdem typu strumieniowego adresy obu punktów końcowych są znane i do przesyłania danych nie potrzeba już żadnych dodatkowych informacji adresujących. Do przesyłania danych można użyć zwykłych funkcji systemowych `read` i `write`.

Najprostszym sposobem zakończenia połączenia i likwidacji związanego z nim gniazda jest użycie funkcji systemowej `close`, odniesionej do deskryptora odpowiedniego gniazda. Można również zadysponować zakończenie jednego tylko kierunku łączności w połączeniu dwukierunkowym. Do tego celu można użyć funkcji systemowej `shutdown`.

Niektóre typy gniazd, takie jak gniazda datagramów, nie ustanawiają połączeń. Zamiast tego gniazda wymieniają datagramy, które muszą być adresowane indywidualnie. Do takiej łączności używa się funkcji systemowych `sendto` oraz `recvfrom`. Obie te funkcje przyjmują jako argumenty deskryptor gniazda, wskaźnik do bufora wraz z jego długością oraz wskaźnik i długość bufora docelowego. Adres bufora docelowego określa adres przesyłania w funkcji `sendto` i jest wypełniony adresem datagramu otrzymanego pośrednio za pomocą operacji `recvfrom`. Obie funkcje przekazują w wyniku liczbę faktycznie przesłanych danych.

Funkcja systemowa `select` może być użyta do rozdzielania przesyłanych danych między kilka deskryptorów plików i (lub) deskryptorów gniazd. Można się nią nawet posłużyć do umożliwienia jednemu procesowi serwera nasłuchiwania połączeń od wielu klientów zamawiających usługi. Dla każdego dokonanego połączenia powołuje się (`fork`) osobny proces. Przy każdej usłudze serwer wykonuje funkcje `socket`, `bind` oraz `listen`, a następnie funkcję `select` skierowaną do wszystkich deskryptorów gniazd. Gdy funkcja `select` wykaże aktywność jakieguz deskryptora, wówczas serwer wykonuje na tym

deskryptorz funkcję `accept` i wytwarza proces do obsługi nowego deskryptora, przekazanego przez funkcję `accept`, pozostawiając procesowi macierzy-stemu dalsze wykonywanie funkcji `select`.

21.9.2 Możliwości działań w sieci

Prawie wszystkie dzisiejsze systemy uniksowe udostępniają pakiet sieciowy UUCP, z którego korzysta się najczęściej za pośrednictwem linii telefonicznych. W ten sposób można mieć dostęp do sieci poczty UUCP oraz do sieci nowości USENET. Są to jednak prymitywne rozwiązania sieciowe, nie umożliwiające nawet zdalnych rejestracji, a tym bardziej zdalnego wywoły-wania procedur lub korzystania z rozproszonych systemów plików. Konstrukcje te są prawie w całości zaimplementowane jako procesy użytkowe i nie są częścią właściwego systemu operacyjnego.

System 4.3BSD umożliwia korzystanie z protokołów DARPA Internet o nazwach: UDP, TCP, IP i ICMP z użyciem wielu rozmaitych interfejsów sieciowych: Ethernet, pierścieni z żetonem i ARPANET. Podbudowa do re-alizacji tych udogodnień w jądrze została tak pomyślana, aby można było uwzględnić implementacje przyszłych protokołów; wszystkie protokoły są dostępne przez interfejs gniazd. Pierwszą wersję kodu w postaci pakietu nad-budowującego system 4.1BSD opracował Rob Gurwitz z firmy BBN.

Model wzorcowy połączeń w systemach otwartych OSI (ang. *Open System Interconnection*) opracowany przez ISO określa siedem warstw protokołów sieciowych i ścisłe metody komunikowania się między nimi. Komunikacja według zaimplementowanego protokołu może się odbywać tylko między równymi sobie jednostkami, porozumiewającymi się takim samym protokołem w tej samej warstwie, lub odnosić się^{*} za pomocą interfejsu międzyproto-kolowego do protokołu w warstwie leżącej bezpośrednio nad lub pod daną warstwą, w tym samym systemie. Model ISO pracy sieciowej zrealizowano w systemach 4.3BSD Reno i 4.4BSD.

Realizacja pracy sieciowej w systemie 4.3BSD oraz w pewnym stopniu koncepcja gniazda odpowiada w większym stopniu modelowi wzorcowemu ARPANET (ang. *ARPANET Reference Model – ARM*). Projekt sieci ARPANET w jego pierwotnej postaci służył jako poligon do sprawdzania wielu koncepcji sieciowych, takich jak przełączanie pakietów i uwarstwowanie protokołów. Prace nad siecią ARPANET zostały zarzucone w 1988 r., ponieważ zastosowany w niej sprzęt zestarzał się technicznie. Jej następczynie – sieci NSFNET i Internet – mają znacznie większe rozmiary i służą jako na-

* W sensie logicznym. – Przyp. tłum.

** W sensie fizycznym. – Przyp. tłum.

rzędzia komunikacji dla prowadzących badania specjalistów oraz jako poletko doświadczalne w badaniach nad bramami internetowymi^{*}. Projekt ARM poprzedził model ISO; badania nad siecią ARPANET w dużym stopniu zainspirowały twórców modelu sieci ISO.

Chociaż model ISO jest często interpretowany jako ograniczający liczbę protokołów komunikacyjnych w warstwie do jednego, projekt ARM dopuszcza występowanie w tej samej warstwie kilku protokołów. W sieci ARM są tylko cztery warstwy protokołów:

- **Proces-zastosowania:** Ta warstwa spełnia zadania warstwy zastosowań, prezentacji i sesji z modelu ISO. Na tym poziomie istnieją takie programy użytkowe, jak protokół przesyłania plików (FTP) i Telnet (zdalna rejestracja).
- **Komputer do komputera (ang. host-host):** Ta warstwa odpowiada warstwie transportowej modelu ISO i wierzchniej części jego warstwy sieciowej. W tej warstwie jest umieszczony zarówno protokół strowania przesyaniem (TCP), jak i protokół internetowy (IP), przy czym protokół TCP znajduje się powyżej protokołu IP. Protokół TCP odpowiada protokołowi transportowemu ISO, a protokół IP spełnia zadania adresowania należące do warstwy sieciowej ISO.
- **Interfejs sieciowy:** Ta warstwa obejmuje dolne piętro warstwy sieciowej modelu ISO i całą warstwę łączą danych. Stosowane tu protokoły zależą od fizycznego typu sieci. Sieć ARPANET używa protokołów IMP-Host, natomiast w sieci Ethernet obowiązuje protokół Ethernet.
- **Sprzęt sieciowy:** Projekt ARM dotyczy zasadniczo oprogramowania, nie ma więc w nim jawnie określonej warstwy sprzętu sieciowego; niemniej jednak każda rzeczywista sieć będzie używać sprzętu odpowiadającego warstwie sprzętowej ISO.

Sieciowa infrastruktura w systemie 4.3BSD jest jeszcze ogólniejsza od modelu ISO lub ARM, choć jest najbardziej zbliżona do tego ostatniego (rys. 21.11).

Procesy użytkowników komunikują się z protokołami sieciowymi (a więc i z innymi procesami na różnych maszynach) za pomocą gniazd, które odpowiadają warstwie sesji modelu ISO, gdyż to ona odpowiada za nawiązywanie i nadzorowanie łączności.

^{*} Szacuje się, że liczba użytkowników sieci Internet osiągnie na przełomie tysiącleci rząd 200 mln. Zatem jest to już sieć o zupełnie nowej jakości i możliwościach. – Przyp. tłum.

Model wzorcowy ISO	Model wzorcowy ARPANET	Warstwy systemu 4.2BSD	Przykłady warstw
Aplikacje	proces aplikacji	programy użytkownika i biblioteki	telnet
Prezentacja		gniazda	sock_stream
Sesje			TPC
Transport	komputer do komputera (ang. host-host)	protokół	IP
Sieć	interfejs sieciowy	interfejsy sieciowe	moduł sterujący Ethernet
Łącze danych			
Sprzęt	sprzęt sieciowy	sprzęt sieciowy	sterownik sieci lokalnej

Rys. 21.11 Modele wzorcowe sieci i ich warstwy

Gniazda są obsługiwane przez *protoły* – być może przez kilka protokołów umieszczonych jeden nad drugim. Protokół może umożliwiać korzystanie z usług takich, jak niezawodne dostarczanie, pomijania podwojonych transmisji, kontrolowanie przepływu lub adresowanie – zależnie od typu zastosowanego gniazda i usług wymaganych przez wyższe protokoły.

Protokół może się komunikować z innym protokołem lub z interfejsem sieciowym odpowiednim do sprzętu sieciowego. W ogólnych wytycznych jest mało ograniczeń w sprawie możliwości komunikowania się ze sobą różnych protokołów oraz co do liczby protokołów umieszczanych warstwowo jeden nad drugim. Proces użytkownika może za pomocą gniazda typu surowego mieć bezpośredni dostęp do każdej warstwy protokołów: od najwyższej – używanej jako oprogramowanie jakiegoś innego typu gniazda, takiego jak strumieniowe, aż po dolną warstwę surowego interfejsu sieciowego. Z tej możliwości korzystają procesy wyznaczające trasy, jest ona również pomocna w opracowywaniu nowych protokołów.

Istnieje tendencja, aby na jeden typ sterownika sieci przypadał jeden moduł obsługi interfejsu sieciowego. *Interfejs sieciowy* jest odpowiedzialny za obsługę specyficznych właściwości adresowania w sieci lokalnej. Takie postawienie sprawy uwalnia protokoły korzystające z tego interfejsu od zajmowania się tymi szczegółami.

Zadania interfejsu sieciowego zależą przede wszystkim od *sprzętu sieciowego* spełniającego niezbędne wymagania sieci, do której jest dołączany. Niektóre sieci mogą zapewniać na tym poziomie niezawodne przesyłanie, lecz wiele z nich tego nie robi. W niektórych sieciach istnieje możliwość adresowania komunikatów rozgłoszanych, jednak w wielu nie ma takiego udogodnienia.

W oprogramowaniu gniazd i ogólnego zaplecza pracy sieciowej stosuje się wspólny zbiór buforów pamięci – tzw. *mbufs*. Pod względem rozmiaru mieścią się one między duzymi buforami blokowego systemu wejścia-wyjścia a C-listami używanymi przez urządzenia znakowe. Bufor *mbuf* ma 128 B długości, z czego 112 B może zawierać dane, a reszta jest przeznaczona na wskaźniki łączące bufore *mbufs* w kolejki oraz na określarki obszaru faktycznie zajętego przez dane.

Dane są przekazywane między warstwami (gniazdo-protokół, protokół-protokół lub protokół-interfejs sieciowy) na ogół w buforach *mbufs*. Możliwość przekazywania danych w buforach eliminuje pewną liczbę operacji ich kopiowania, ale i tak często istnieje konieczność usuwania lub dołączania nagłówków protokołów. Z wielu powodów jest również wygodne i wydajne umieszczenie danych w obszarach, których rozmiar odpowiada stronie w rozumieniu zarządzania pamięcią. Umożliwia się zatem, aby dane z bufora *mbuf* mogły przebywać nie w samym buforze *mbuf*, lecz w dowolnym miejscu pamięci. W tym celu istnieje tablica stron buforów *mbufs*, a także pula stron przeznaczonych na użycie tych buforów.

21.10 ■ Podsumowanie

Do pierwszych zalet systemu UNIX zalicza się napisanie go w języku wysokiego poziomu i rozprowadzanie w postaci programu źródłowego. Od początku był to też system, który przy użyciu oszczędnych środków umożliwiał korzystanie z silnych, elementarnych operacji systemowych. Zalety te spowodowały spopularyzowanie systemu UNIX w instytucjach edukacyjnych, badawczych i rządowych, a w końcu – w kręgach handlowych. Popularność systemu UNIX doprowadziła początkowo do powstania jego wielu wariantów i ulepszeń. Naciski ze strony rynku spowodowały obecną tendencję do konsolidacji tych wersji. Jedną z najbardziej znanych i naśladowanych odmian systemu jest wersja 4.3BSD, opracowana w California University w Berkeley dla komputera VAX, a potem przeniesiona na komputery wielu innych typów.

UNIX realizuje system plików o drzewiastej strukturze katalogowej. Jądro systemu traktuje pliki jako pozbawione struktury ciągi bajtów. Dostęp bezpośredni i sekwencyjny do danych umożliwiają funkcje systemowe i biblioteki procedur.

Pliki są przechowywane w postaci zestawów bloków danych o stałym rozmiarze; na końcach plików mogą występować mniejsze od bloków fragmenty. Bloki danych są odnajdywane za pomocą wskaźników przechowywanych w i-węzłach plików. Wpisy katalogowe wskazują na i-węzły. Przestrzeń

dyskowa jest przydzielana w grupach cylindrów, aby zminimalizować ruch głowic i polepszyć wydajność.

UNIX jest systemem wieloprogramowym. Procesy mogą łatwo tworzyć nowe procesy z pomocą funkcji systemowej `fork`. Procesy mogą się komunikować za pośrednictwem potoków lub – w sposób bardziej ogólny – używając gniazd. Procesy można grupować w zadania sterowane za pomocą sygnałów.

Procesy są reprezentowane przez dwie struktury: strukturę procesu i strukturę użytkownika. Planowanie przydziału procesora opiera się na algorytmie priorytetowym z dynamicznie obliczanymi priorytetami, który w skrajnym przypadku sprawdza się do planowania rotacyjnego.

Zarządzanie pamięcią w systemie 4.3BSD polega na dokonywaniu wymian wspomaganych stronnicowaniem. Przy zastępowaniu stron proces-demon stronnicowania stosuje zmodyfikowany algorytm drugiej szansy, utrzymując liczbę wolnych ramek na poziomie wystarczającym do działania innych procesów.

Operacje wejścia-wyjścia dotyczące stron oraz plików korzystają z podręcznego buforowania bloków, co minimalizuje liczbę rzeczywistych operacji wejścia-wyjścia. W terminalach stosuje się odrębny system buforowania znaków.

Jedną z najistotniejszych cech systemu 4.3BSD jest organizacja pracy w sieci. Koncepcja gniazda pozwala utworzyć programowy mechanizm uzyskiwania dostępu do innych procesów, także poprzez sieć. Gniazda są interfejsem dla kilku zbiorów protokołów.

■ Ćwiczenia

- 21.1 Jakie są główne różnice między systemem UNIX 4.3BSD a wersją UNIX System V Release 3 (SYSVR3). Czy jeden z nich jest „lepszy” od drugiego? Odpowiedź uzasadnij.
- 21.2 Na czym polegały różnice między celami projektowymi z wczesnych faz rozwoju systemu UNIX a koncepcjami zastosowanymi w innych systemach operacyjnych?
- 21.3 Dlaczego obecnie jest dostępnych wiele wersji systemu UNIX? W jakim sensie jest to korzystne dla systemu UNIX? Jakie są tego ujemne strony?
- 21.4 Jakie są zalety i wady pisania systemu operacyjnego w języku wysokiego poziomu, takim jak C?
- 21.5 W jakich okolicznościach użycie ciągu funkcji systemowych: `fork` i `execve` jest najwłaściwsze? Kiedy lepiej użyć funkcji `vfork`?

- 21.6 Czy system 4.3BSD daje preferencje w przydzielaniu procesora zadaniom zależnym od wejścia-wyjścia czy od procesora? Z jakich powodów różni się on te kategorie i dlaczego jedna z nich ma pierwszeństwo przed drugą? Na podstawie czego system może rozpoznać, do której z tych kategorii można zaliczyć dany proces?
- 21.7 We wczesnych systemach UNIX stosowano do zarządzania pamięcią wymianę procesów, natomiast w wersji 4.3BSD używa się stronicowania i wymiany. Omów zalety i wady obu tych metod zarządzania pamięcią.
- 21.8 Opisz zmiany, które w systemie plików wykonuje jądro systemu 4.3BSD, gdy proces zamówi utworzenie nowego pliku */tmp/takitam* i zacznie go sekwencyjnie zapisywać, aż do osiągnięcia 20 KB.
- 21.9 Bloki katalogów w systemie 4.3BSD są zapisywane (na dysku) na bieżąco z wykonywanymi w nich zmianami. Rozważ, co mogłoby się stać, gdyby bloki te nie były zapisywane synchronicznie. Opisz stan systemu plików, jeśli do jego awarii doszło po usunięciu wszystkich plików w katalogu, lecz przed uaktualnieniem wpisu katalogowego na dysku.
- 21.10 Opisz postępowanie potrzebne do odtworzenia wykazu wolnych bloków po awarii w systemie 4.1BSD.
- 21.11 Jaki wpływ na wydajność systemu miałyby następujące zmiany wprowadzone do wersji 4.3BSD? Odpowiedź uzasadnij.
- (a) Połączenie pamięci podręcznej buforów z przestrzenią stronicowania procesów.
 - (b) Wykonywanie dyskowych operacji wejścia-wyjścia większymi porcjami.
 - (c) Zaimplementowanie pamięci dzielonej i używanie jej w celu przekazywania danych między procesami zamiast stosowania zdalnych wywołań procedur lub gniazd.
 - (d) Zastosowanie siedmiowarstwowego protokołu ISO zamiast modelu sieciowego ARM.
- 21.12 Jaki typ gniazda powinno się zastosować do realizacji programu przesyłania plików między komputerami? Jakim typem gniazda należałoby się posłużyć w programie, który okresowo sprawdza, czy inny komputer jest włączony do sieci? Odpowiedź uzasadnij.

Uwagi bibliograficzne

Najlepszym ogólnym opisem charakterystycznych cech systemu UNIX pozostaje wciąż artykuł Ritchiego i Thompsona [354]. Wiele z historii systemu UNIX opisał Ritchie w pracy [353]. Krytykę tego systemu przedstawili Blair i in. [44]. Dwie główne, współczesne wersje systemu UNIX to 4.3BSD oraz System V. Wewnętrzna budowę Systemu V przedstawia w szczegółach Bach w książce [20]. Autorytatywny opis projektu i realizacji systemu 4.3BSD zawiera książka Lefflera i in. [245].

Być może najlepszą książką dotyczącą ogólnych zasad programowania w systemie UNIX, zwłaszcza w zakresie używania powłoki i udogodnień takich jak programy *yacc* i *sed*, jest dzieło Kernighana i Pike'a [207]. Programowanie poziomu systemowego przedstawił Stevens w książce [407]. Inną interesującą pozycją jest książka Bourne'a [48]. Za podstawowy język programowania w systemie UNIX obrano język C; opisali go w książce [208] Kernighan i Ritchie. C jest także językiem implementacji systemu. Powłokę Bourne'a opisał Bourne [47]. Powłokę Korna opisał Korn [221].

Zestaw dokumentacji towarzyszącej systemom UNIX jest nazywany *UNIX Programmer's Manual* (UPM) i jest tradycyjnie wydawany w dwu tomach. Tom 1 zawiera krótkie omówienia wszystkich poleceń, funkcji systemowych i systemowego pakietu podprogramów. Jest on także dostępny bezpośrednio w komputerze pracującym pod systemem UNIX (polecanie *man*). Tom 2, pod tytułem *Supplementary Documents*, czyli uzupełnienia (zwyczek podzielony na tomy 2A i 2B ze względu na wygodę oprawy introligatorskiej), zawiera wybór niezbędnych artykułów o systemie oraz instrukcje obsługi poleceń lub pakietów, które są za obszerne do przedstawienia na jednej lub dwu stronach. Systemy opracowywane w Berkeley są opisane jeszcze w tomie 2C, który zawiera opis właściwości specyficznych dla wersji z Berkeley.

System plików wersji 7 systemu UNIX jest opisany przez Thompsona [426], a system plików 4.2BSD udokumentowali McKusick i in. [278]. Odporny na awarie system plików UNIX opisali Anyanwu i Marshall [12]. Podstawowym źródłem informacji o zarządzaniu procesami jest praca Thompsona [426]. System zarządzania pamięcią 3BSD opisali Babaoglu i Joy [19], a pewne ulepszenia zarządzania pamięcią w systemie 4.3BSD przedstawili McKusick i Karels [277]. System wejścia-wyjścia opisał Thompson [426].

Opis zagadnień bezpieczeństwa w systemie UNIX podali Grampp i Morris [153] oraz Wood i Kochan [440].

* W 1996 r. ukazała się w USA analogiczna książka poswięcona finalnej wersji systemu: 4.4BSD. – Przyp. tłum.

Dwa pozytyczne artykuły na temat komunikacji w systemie 4.2BSD są autorstwa Lefflera i in. [243] i [244] – oba zawarte w tomie 2C UPM.

Opis normy *ISO Reference Model* znajduje się w publikacji [196]. Opis modelu wzorcowego sieci ARPANET przedstawili Cerf i Cain [66]. Sieć Internet i jej protokoły są opisane w książce Comera [81] oraz w książkach Comera i Stevensa – [82] i [83]. W książce [338] Quartermanna zawarł ogólny przegląd stanu sieci.

Wiele pozytycznych artykułów o systemie UNIX zawierają dwa specjalne wydania czasopisma *The Bell System Technical Journal* ([430] i [425]). Inne artykuły na ten temat pojawiły się na różnych konferencjach organizacji USENIX i są dostępne w sprawozdaniach z tych konferencji, a także w czasopiśmie *Computer Systems* związanym z USENIX.

Kilku autorów opracowało podręczniki różnych odmian systemu UNIX, w tym: Holt [178] (system operacyjny Tunis), Comer [79] i [80] (system operacyjny Xinu) oraz Tanenbaum i Woodhull [418] (Minix).

Rozdział 22

SYSTEM LINUX

W rozdziale 21 omówiliśmy szczegółowo wewnętrzną organizację systemu 4.3BSD. System BSD jest tylko jednym z systemów uniksopodobnych. Innym systemem wykazującym podobieństwo do systemu UNIX jest Linux, zyskujący w ostatnich latach na popularności. W tym rozdziale dokonujemy przeglądu historii i rozwoju systemu Linux oraz prezentujemy interfejsy użytkownika i programisty tego systemu, w których można odnaleźć wiele z tradycyjnych rozwiązań systemu UNIX. Omawiamy również wewnętrzne metody, za pomocą których w systemie Linux zrealizowano te interfejsy. Zważywszy jednak, że Linux zaprojektowano z myślą o wykonywaniu pod jego kontrolą możliwe wielu standardowych aplikacji uniksowych, ma on wiele wspólnego z istniejącymi implementacjami systemu UNIX. Nie będziemy zatem powtarzać podstawowego opisu systemu UNIX podanego w poprzednim rozdziale.

22.1 ■ Historia

System Linux sprawia wrażenie podobne do wielu innych systemów uniksowych i rzeczywiście – zgodność z systemem UNIX była głównym celem przy jego projektowaniu. Jednakże Linux jest znacznie młodszy od większości pozostałych systemów uniksowych. Prace nad nim rozpoczęły się w 1991 r., kiedy to fiński student, Linus Torvalds, napisał i ochrzcił małe, lecz kompletnie jądro dla procesora 80386 – pierwszego prawdziwie 32-bitowego procesora w zborze jednostek centralnych firmy Intel zgodnych ze standardem PC.

We wczesnej fazie rozwoju Linuxa jego kod źródłowy udostępniono bezpłatnie w sieci Internet. W związku z tym jego historia potoczyła się dalej

w formie współpracy wielu użytkowników z całego świata, korespondujących niemal wyłącznie za pośrednictwem sieci Internet. Z jądra, które początkowo tylko częściowo realizowało mały podzbior usług systemu UNIX, Linux rozrosł się do postaci obejmującej wszystkie funkcje uniksowe*.

W początkowym okresie rozwój systemu Linux koncentrował się na rdzeniu, tj. centralnych elementach jądra systemu operacyjnego, czyli uprzywilejowanym egzekutorze zarządzającym wszystkimi zasobami systemu, współpracującym wprost ze sprzętem. Oczywiście opracowanie pełnego systemu operacyjnego wymaga znacznie więcej niż tylko jądra. Wygodnie jest rozmieścić między jądrem systemu Linux a systemem Linux. Jądro w Linuxie jest oprogramowaniem w całości oryginalnym, wykonanym od podstaw przez społeczność linuksową. System Linux, jakim go znamy dzisiaj, zawiera liczne elementy, z których jedne napisano od nowa, a drugie zapożyczono z innych projektów rozwojowych lub utworzono we współpracy z innymi zespołami.

Podstawowy system Linux jest standardowym środowiskiem dla aplikacji i programów użytkownika, które jednak nie narzuca żadnych standardowych sposobów korzystania z dostępnych funkcji, pojmowanych jako całość. W miarę dojrzewania projektu Linux powstała potrzeba zbudowania jeszcze jednej warstwy usług na szczycie systemu Linux. Pakiet dystrybucyjny Linuksa obejmuje wszystkie standardowe części systemu Linux wraz ze zbiorem narzędzi administracyjnych upraszczających wstępne instalowanie systemu i dalsze jego ulepszanie oraz umożliwiających instalowanie i demontowanie innych pakietów systemowych. Nowoczesny pakiet dystrybucyjny zazwyczaj zawiera także narzędzia do administrowania systemem plików, tworzenia kont użytkowników i zarządzania nimi, administrowania siecią itp.

22.1.1 Jądro systemu Linux

Pierwsza wersja jądra systemu Linux, która pojawiła się w obiegu, miała numer 0.01 i była datowana na 14 maja 1991 r. Nie mogła ona współpracować z siecią; działała tylko na procesorach zgodnych z modelem Intel 80386 i sprzęcie PC, a jej oprogramowanie urządzeń zewnętrznych było nader ograniczone. Podsystem pamięci wirtualnej był również bardzo elementarny i nie zawierał żadnych środków odwzorowywania plików w pamięci. Jednak nawet to wczesne wcielenie systemu udostępniało strony dzielone w trybie kopowania przy zapisie. Jako system plików dostarczany był jedynie system plików Minix, co wynikało z tego, że pierwsze jądra Linuksa były opracowywane na

* Nie pierwszy raz, gdy dostępność kodu źródłowego sprzyja popularyzacji i rozbudowie systemu operacyjnego. Tak samo zaczynał przed laty karierę oryginalny UNIX Thompsona – Przyp. tłum.

komputerach zaopatrzonych w system Minix*. Niemniej jednak jądro to realizowało właściwe procesy systemu UNIX w chronionych przestrzeniach adresowych.

Dopiero 14 marca 1994 r. ukazała się istotnie różna wersja systemu – Linux 1.0. Wielezyła ona trzy lata burzliwego rozwoju jądra Linuxa. Niewykluczone, że jedną z najistotniejszych nowych cech było usiociowienie systemu: wersja 1.0 zawierała możliwości wykonywania standardowych, unikso- wych protokołów sieciowych TCP/IP, a także zgodny z systemem BSD interfejs gniazd, umożliwiający programowanie sieciowe. Dodano oprogramowanie sterujące za pomocą protokołu IP urządzeniami w sieci Ethernet lub połączonymi liniami szeregowymi albo modemami za pomocą protokołów PPP lub SLIP.

Jądro 1.0 zawierało również nowy, znacznie ulepszony system plików, w którym usunięto ograniczenia pierwotnego systemu Minix, oraz dostarczało całej gamy sterowników SCSI do wysoko wydajnego dostępu do dysków. Rozbudowano także podsystem pamięci wirtualnej, umożliwiając stronicowanie z udziałem plików wymiany oraz odwzorowywanie dowolnych plików w pamięci (jednak w wersji 1.0 odwzorowywano w ten sposób tylko informacje do czytania).

W wydaniu tym uwzględniono repertuar dodatkowego wyposażenia sprzętowego. Mimo utrzymanego ograniczenia do platformy Intel-PC wyposażenie sprzętowe rozrosło się o urządzenia dysków elastycznych i pamięci CD-ROM, a także o karty dźwiękowe, rozmaite myszki i klawiatury międzynarodowe. Użytkownikom procesorów 80386 nie mającym koprocesora matematycznego 80387 umożliwiono emulację obliczeń zmiennopozycyjnych. Zrealizowano też komunikację międzyprocesową IPC na sposób znany z systemu UNIX System V, wraz z pamięcią dzieloną, semaforami i kolejkami komunikatów. W prosty sposób umożliwiono korzystanie z modułów jądra ładowanych i usuwanych dynamicznie.

Poczynając od tego miejsca, rozpoczęto prace nad serią jąder 1.1, a jednocześnie publikowano liczne poprawki błędów wykrytych w wersji 1.0. Według tego schematu zapoczątkowano standardową numerację jąder Linuxa: jądra z nieparzystymi numerami po kropce, w rodzaju 1.1, 1.3 lub 2.1, są wersjami rozwojowymi; małe numery parzyste oznaczają wersje stabilne, czyli jądra, które można uważać za produkty finalne. Uaktualnienia wersji stabilnych mogą występować tylko w przypadkach wymagających działań naprawczych, natomiast jądra rozwojowe mogą zawierać funkcje nowsze i stosunkowo mało przetestowane.

* Minix to akademicka odmiana systemu UNIX zaprojektowana do celów dydaktycznych pod kierunkiem A.S. Tanenbauma w Uniwersytecie Vrije (Amsterdam). — Przyp. tłum.

W marcu 1995 r. ukazało się jądro w wersji 1.2. W wydaniu tym przyrost funkcjonalności nie był reprezentowany w stopniu porównywalnym z wersją 1.0, jednakże zawierało ono możliwości działania na znacznie szerszym asortymencie sprzętu, wliczając w to nową architekturę szynową PCI. Dodano też na procesorze 80386 charakterystyczny dla komputerów PC tryb obsługi wirtualnego procesora 8086, aby umożliwić emulowanie systemu operacyjnego DOS komputerów osobistych. Stos protokołów sieciowych uaktualniono tak, aby można było korzystać z protokołu IPX. Dodano również pełniejszą implementację protokołu IP, zaopatrzoną w rozliczanie i zapory ogniowe.

W końcowym efekcie wersja 1.2 reprezentowała jedynie jądro Linuxa dla komputerów PC. Pakiet dystrybucyjny Linux 1.2 zawierał częściową realizację dla procesorów Sparc, Alpha i Mips, lecz pełna integracja tych architektur nastąpiła dopiero po wdrożeniu stabilnej wersji jądra 1.2.

W wersji Linux 1.2 skoncentrowano się na szerszym udostępnieniu sprzętu i pełniejszej implementacji istniejących funkcji. W tym czasie w trakcie opracowywania było już znacznie więcej nowych funkcji, lecz integrację nowego kodu z głównym kodem źródłowym jądra opóźniano aż do opublikowania jego stabilnej wersji 1.2. W rezultacie w serii rozwojowych wersji jądra 1.3 pojawiło się mnóstwo nowych własności.

Praca ta zaowocowała ostatecznie w czerwcu 1996 r. wersją systemu Linux 2.0. Wydanie opatrzone znacznie większym numerem wersji ze względu na dwie główne, nowe możliwości: dostępność wielu architektur, w tym – w pełni 64-bitowego, rodzimego portu Alpha, oraz dostępność dla architektur wieloprocesorowych. Pakiety dystrybucyjne systemu Linux oparte na wersji jądra 2.0 są także dostępne dla procesorów serii Motorola 68000 oraz dla systemów Sun Sparc. Pochodna wersja Linuxa, działająca na szczeblu mikrojądra systemu Mach, pracuje również na komputerach PC i w systemach PowerMac.

Na wersji 2.0 nie zaprzestano zmian. Istotnie ulepszono kod zarządzania pamięcią, dostarczając ujednoliconej pamięci podręcznej dla danych systemu plików, niezależnej od pamięci podręcznych urządzeń blokowych. W wyniku tej zmiany jądro wzbogaciło się o znacznie poszerzony system plików i zwiększoną wydajność pamięci wirtualnej. Podręczną pamięć plików po raz pierwszy rozciągnięto na sieciowe systemy plików, umożliwiono również odwzorowywanie w pamięci obszarów zapisywanych.

Jądro 2.0 zawierało także znacznie ulepszone w działaniu protokoły TCP/IP. Dodano do niego również pewną liczbę nowych protokołów sieciowych, takich jak Appletalk, amatorskie sieci radiowe AX.25 oraz standard ISDN. System uzupełniono też o możliwości montowania zdalnych woluminów sieci Netware oraz SMB (Microsoft LanManager).

Do innych poważnych ulepszeń w wersji 2.0 należało zorganizowanie wątków wewnętrz jądra, możliwość manipulowania zależnościami między

modułami ładowalnymi oraz automatyczne ładowanie modułów na żądanie. Nowy, standaryzowany interfejs konfiguracji pozwolił na istotne usprawnienie dynamicznego konfigurowania jądra w trakcie jego działania. Dodatkowe, luźne cechy, które wprowadzono w tej wersji, obejmują rozmiary systemu plików oraz zgodne ze standardem POSIX klasy planowania (szeregowania) procesów w czasie rzeczywistym.

22.1.2 System Linux

Jądro systemu Linux z wielu względów stanowi centralną część projektu Linux, jednak kompletny system operacyjny Linux tworzą inne składowe. Podczas gdy jądro Linux jest zestawione w całości z kodu napisanego od zera specjalnie dla projektu Linux, wiele z pomocniczego oprogramowania tworzącego system Linux nie należy wyłącznie do projektu Linux, lecz jest wspólne dla wielu systemów uniksowych. W szczególności, w systemie Linux znajduje zastosowanie wiele narzędzi opracowanych jako część systemu operacyjnego BSD z Berkeley, systemu X Window z MIT oraz oprogramowania projektu GNU kierowanego przez Free Software Foundation.

Ten podział narzędzi zadziałał obustronnie. Główne biblioteki systemu Linux wywodzą się z projektu GNU, jednak wspólnota linuksowa włożyła sporo wysiłku w ulepszanie tych bibliotek pod kątem uzupełnień, usuwania niewydajności i błędów. Inne składowe, takie jak kompilator GNU C – `gcc`, miały już odpowiednią jakość, aby można ich było używać w systemie Linux bezpośrednio. Narzędzia administrowania siecią wprowadzono do systemu Linux z kodu opracowanego pierwotnie dla systemu 4.3BSD, jednakże w ostatnich wersjach systemu BSD, jak na przykład w FreeBSD, zapozyczono w zamian kod z systemu Linux, taki jak: emulacja obliczeń zmiennopozycyjnych dla procesora Intel, biblioteka matematyczna oraz moduły sterujące urządzeń dźwiękowych dla komputerów PC.

System Linux jako całość jest utrzymywany przez luźną sieć twórców współpracujących za pośrednictwem Internetu, w której małe grupy lub poszczególne osoby sprawują pieczę nad całością konkretnych składowych. Niewielka liczba powszechnie dostępnych stanowisk archiwizujących `ftp` (internetowy protokół przesyłania danych) pełni de facto rolę magazynów standardów tych składowych. Społeczność systemu Linux utrzymuje również dokument pod nazwą *File System Hierarchy Standard* (standard hierarchii systemu plików) w celu zachowywania zgodności między różnymi składowymi systemu. Standard ów określa ogólny wygląd znormalizowanego systemu plików Linux, tj. nazwy katalogów, w których powinno się przechowywać pliki konfiguracyjne, biblioteki, systemowe pliki binarne oraz pliki wykorzystywane w czasie działania systemu.

22.1.3 Upowszechnianie systemu Linux

Teoretycznie system Linux może zainstalować każdy, sprowadzając najnowsze wersje niezbędnych części systemu ze stanowisk ftp, po czym je kompljując. We wczesnych czasach systemu Linux czynność tę często musiał wykonać sam użytkownik Linuxa. Jednak w miarę dojrzewania systemu poszczególne osoby i grupy dołożyły starań, aby zadanie to uczynić mniej bolesnym, dostarczając standardowego, zawczasu skompilowanego zbioru pakietów do łatwego instalowania.

Zbiory te, zwane *dystrybucjami* (ang. *distributions*), zawierają znacznie więcej niż tylko podstawowy system Linux. Na ogół wchodzą w ich skład specjalne narzędzia instalowania systemu i zarządzania nim, jak również uprzednio skompilowane i gotowe do instalowania pakiety licznych, typowych narzędzi systemu UNIX, takich jak serwery, przeglądarki WWW, oprogramowanie przetwarzania tekstu, edytory, a nawet gry.

Pierwsze dystrybucje po prostu rozpakowywały wszystkie pliki pakietów w odpowiednich miejscach. Współczesne dystrybucje wnoszą w ten proces zaawansowane zarządzanie pakietami. Dzisiejsze dystrybucje Linuxa zawierają bazę danych nadzorującą pakiety, która pozwala na bezbolesną instalację, rozbudowę i usuwanie pakietów.

Pierwszą uznaną za kompletną dystrybucję był zbiór pakietów Linuxa nazwany SLS. Działo się to we wczesnym okresie użytkowania systemu. Choć można ją było zainstalować jako jedną całość, dystrybucja SLS była pozbawiona narzędzi zarządzania pakietami, których od dystrybucji linukso-wych oczekuje się obecnie. Dystrybucja Slackware była już istotnym ulepszeniem, jeśli wziąć pod uwagę jej jakość całościowo (choć i ona miała słabe zarządzanie pakietami). Dystrybucja ta jest nadal jedną z najszerzej instalowanych przez wspólnotę użytkowników systemu Linux.

Od czasu wydania Slackware ukazało się dużo komercyjnych i niekomercyjnych dystrybucji systemu Linux. Do szczególnie popularnych zalicza się dystrybucję o nazwie *Red Hat*, upowszechnianą komercyjnie, oraz dystrybucję *Debian*, rozprowadzaną bezpłatnie przez społeczność linuksovą. Do innych komercyjnie upowszechnianych wersji systemu Linux należą: *Caldera*, *Craftworks* oraz *WorkGroup Solutions*. Duże zainteresowanie Linuxem w Niemczech zaowocowało licznymi, specjalnymi dystrybucjami niemieckojęzycznymi, m.in. wersjami pochodząymi z firm *SuSE* i *Unifox*. Obecnie w obiegu znajduje się zbyt wiele dystrybucji Linuxa, aby przedstawić tu ich pełną listę. Rozmaitość dostępnych dystrybucji nie przeszkadza ich wzajemnej zgodności. Większość dystrybucji stosuje lub przynajmniej rozpoznaje pakiety plików formatu *RPM*, a rozpowszechniane w tym formacie aplikacje komercyjne można instalować i wykonywać w dowolnej dystrybucji akceptującej pliki *RPM*.

22.1.4 Licencje systemu Linux

Jądro systemu Linux jest upowszechniane według praw znanych jako GNU General Public Licence (GPL), określonych przez konsorcjum Free Software Foundation. System Linux nie jest oprogramowaniem typu public domain. Określenie *public domain* oznacza zrzeczenie się przez autorów praw autorskich do upowszechniania oprogramowania, natomiast prawa autorskie do upowszechniania kodu Linuksa są wciąż w rękach różnych autorów kodu. Jednakże Linux jest oprogramowaniem w wolnym obiegu (ang. *free software*) w tym sensie, że jego użytkownikom wolno go kopiować, zmieniać i stosować w dowolny sposób oraz rozdawać własne kopie bez ograniczeń.

Głównymi przesłankami wynikającymi ze sformułowań licencji systemu Linux jest zakaz prywatyzacji jego pochodnych produktów przez kogokolwiek korzystającego z Linuksa lub wytwarzającego własną pochodną systemu (tj. zachowującą standardy jego odmianę). Oprogramowania rozpowszechnianego na zasadach licencji GPL nie wolno redystrybuować w formie czysto binarnej. Jeżeli ktoś puszcza w obieg oprogramowanie zawierające jakkolwiek część podlegającą licencji GPL, to zgodnie z prawami licencji musi udostępnić wraz z każdą dystrybucją binarną jej kod źródłowy. (Ograniczenie to nie zakazuje wytwarzania – lub nawet sprzedawania – wyłącznie binarnych dystrybucji oprogramowania, jeżeli tylko każdy, kto otrzymuje kopię binarne, będzie miał szansę uzyskania również kodu źródłowego za rozsądną opłatą dystrybucyjną).

22.2 ■ Podstawy projektu

W ogólnym założeniu Linux przypomina każdą inną, tradycyjną, nie opartą na mikrojadrze realizację systemu UNIX. Jest to system wielodostępny i wielozadaniowy, z pełnym zestawem narzędzi zgodnych z systemem UNIX. System plików Linuks pasuje do tradycyjnej semantyki uniksowej, zrealizowano też w pełni sieciowy standard systemu UNIX. Na wewnętrzne szczegóły projektu Linuks wywarły duży wpływ kolejne etapy jego powstawania.

Chociaż Linuks działa na wielu rodzajach komputerów, we wczesnym okresie opracowywano go wyłącznie dla architektur PC. Znaczną część wczesnych prac wykonali pojedynczy entuzjaści, a nie stateczne ciała badawcze lub rozwojowe. toteż od samego początku w systemie Linuks usiłowano upakować tak wiele funkcji, jak tylko na to pozwalały ograniczone zasoby. Dziś Linuks może z powodzeniem działać na maszynie wieloprocesorowej z setkami megabajtów pamięci operacyjnej i wieloma gigabajtami przestrzeni

dyskowej, a jednocześnie wciąż jest w stanie pracować poprawnie z pamięcią RAM mniejszą niż 4 MB.

W miarę wzrostu mocy obliczeniowej komputerów PC oraz spadku cen pamięci operacyjnej i dysków twardych pierwotne, minimalistyczne jądra Linuksa powiększono o więcej funkcji systemu UNIX. Szybkość i wydajność są nadal ważnymi celami projektowymi, jednak wiele niedawnych i bieżących prac na systemem Linux skoncentrowano na drugim ważnym celu projektowym – standaryzacji. Jedną z kar, jaką przychodzi płacić za zróżnicowanie aktualnie dostępnych implementacji systemu UNIX, jest to, że kod źródłowy napisany w jednym stylu niekoniecznie daje się skompilować lub nie zawsze działa poprawnie w innych warunkach. Nawet jeżeli w dwóch różnych systemach UNIX występują te same funkcje systemowe, niekonieczne zachowują się w identyczny sposób. Standardy POSIX są zbiorem specyfikacji różnych aspektów sprawowania się systemu operacyjnego. Istnieją dokumenty standardu POSIX dotyczące wspólnych właściwości systemu operacyjnego, a także rozszerzeń, jak wątki procesowe czy działania w czasie rzeczywistym. System Linux zaprojektowano tak, aby pozostawał w wyraźnej zgodzie z istotnymi opisami standardu POSIX. Oficjalne certyfikaty komitetu normalizacyjnego POSIX zdobyły co najmniej dwie dystrybucje systemu Linux.

Linux, reprezentując standardowe interfejsy zarówno dla programisty, jak i dla użytkownika, nie powinien stwarzać wielu niespodzianek komukolwiek zaznajomionemu z systemem UNIX. Nie zajmujemy się w szczegółach owymi linuksowymi interfejsami. Punkty poświęcone interfejsowi programisty (p. 21.3) i interfejsowi użytkownika (p. 21.4) systemu 4.3BSD w rozdz. 21 w równym stopniu odnoszą się do systemu Linux. Jednakże, na mocy ustaleń zastępczych, interfejs programisty w systemie Linux odpowiada semantycy systemu SVR4 UNIX, a nie zachowaniu systemu BSD. Do realizacji semantyki BSD w miejscach, gdzie oba sposoby działania istotnie się różnią, służy oddzielnny zbiór bibliotek.

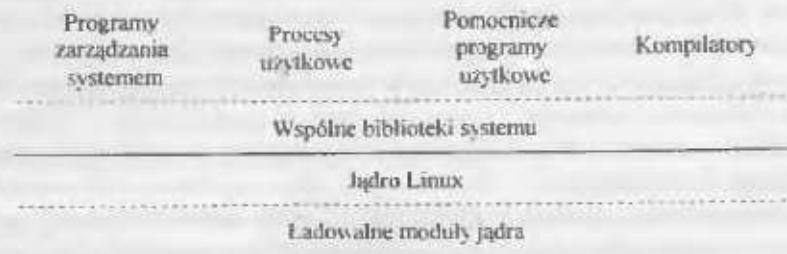
W świecie uniksowym istnieje wiele innych standardów, lecz pełne uzgodnienie z nimi systemu Linux jest spowalniałe niekiedy dlatego, że często są one osiągalne tylko odważnie, więc uzyskanie certyfikatów na zgodność systemu operacyjnego z licznymi standardami wymaga sporych wydatków. Jednakże wspieranie szerokiego repertuaru aplikacji jest istotne dla każdego systemu operacyjnego, toteż realizacja standardów jest ważnym celem rozwojowym systemu Linuks – nawet wówczas, gdy nie osiąga ona formalnego świadectwa poprawności. Oprócz podstawowego standardu POSIX system Linuks dostarcza teraz rozszerzeń dotyczących wątków POSIX oraz podzbioru rozszerzeń normy POSIX związanych ze sterowaniem procesami w czasie rzeczywistym.

22.2.1 Składowe systemu Linux

System Linux składa się z trzech głównych fragmentów kodu, zgodnych z większością tradycyjnych implementacji systemu UNIX. Są to:

- **Jądro:** Jest odpowiedzialne za realizację wszystkich istotnych abstrakcji systemu operacyjnego, łącznie z takimi elementami, jak pamięć wirtualna i procesy.
- **Biblioteki systemowe:** Definiują standardowy zbiór funkcji, za pomocą których aplikacje mogą współdziałać z jądrem i które realizują wiele właściwości systemu operacyjnego nie wymagających pełnych przywilejów kodu jądra.
- **Pomoc systemowe:** Są programami, które wykonują osobne, specjalizowane zadania administracyjne. Z niektórych pomocy systemowych można korzystać tylko jeden raz w celu zapoczątkowania i skonfigurowania pewnych elementów systemu; inne (w terminologii uniksowej nazywane *demonami*) mogą działać nieustannie, obsługując takie zadania, jak odpowiadanie na sygnały nadchodzące z sieci, przyjmowanie z terminali zamówień na rozpoczęcie sesji lub aktualnianie plików z dziennikami zdarzeń systemowych.

Na rysunku 22.1 widać różne części, z których składa się system Linux. Najważniejsza linia podziału biegnie tu między jądrem a wszystkim innym. Cały kod jądra jest wykonywany w uprzywilejowanym trybie procesora, z pełnym dostępem do wszystkich fizycznych zasobów komputera. W systemie Linux ów uprzywilejowany tryb nazywa się *trybem jądra* (ang. *kernel mode*) i jest równoważny trybowi monitora opisanemu w p. 2.5.1. W jądro systemu Linux nie jest wbudowany żaden kod działający w trybie użytkownika. Zamiast tego wszelki kod wspierający system operacyjny, a nie wymagający wykonywania w trybie jądra, umieszczono w bibliotekach systemowych.



Rys. 22.1 Składowe systemu Linux

Mimo że w rozmaitych nowoczesnych systemach operacyjnych do wewnętrznych rozwiązań jądrowych zaadaptowano architekturę przekazywania komunikatów, Linux zachowuje historyczny model systemu UNIX – jądro jest utworzone w formie jednego, monolitycznego modułu binarnego. Uczyniono tak przede wszystkim ze względu na poprawienie wydajności. Ponieważ cały kod jądra i wszystkie struktury danych są trzymane w jednej przestrzeni adresowej, więc gdy proces wywołuje funkcję systemu operacyjnego albo gdy sprzęt zgłosi przerwanie, nie ma potrzeby przełączania kontekstu. W tej samej przestrzeni adresowej przebywa nie tylko kod centralnego planowania i pamięci wirtualnej, lecz także *cały* kod jądra ze wszystkimi modułami obsługi urządzeń, systemami plików i oprogramowaniem sieci.

To, że całe jądro korzysta z tego samego, wspólnego tygla, nie oznacza, iż nie ma w nim miejsca na modularność. W taki sam sposób, jak aplikacje użytkowe wydobywają i wprowadzają w trakcie działania potrzebne fragmenty kodu ze wspólnych bibliotek, jądro systemu Linux może dynamicznie ładować (i rozładowywać) moduły podczas pracy. Jądro nie musi obowiązkowo wieździć z góry, które moduły będą potrzebne – są to elementy ładowalne w pełni niezależnie.

Jądro Linux tworzy rdzeń systemu operacyjnego Linux. Dostarcza ono wszystkich funkcji niezbędnych do wykonywania procesów oraz usług systemowych umożliwiających rozsądny i chroniony dostęp do zasobów sprzętowych. Jądro realizuje całość cech wymaganych, aby zasłużyć na miano systemu operacyjnego. Jednak system operacyjny tworzony przez samo jądro Linux ma niewiele wspólnego z systemem UNIX. Brakuje mu wielu specjalnych właściwości systemu UNIX, a to, co w istocie oferuje, nie zawsze jest dostarczane w formacie oczekiwany przez aplikacje uniksowe. Interfejs systemu operacyjnego, widoczny dla wykonywanych aplikacji, nie jest bezpośrednio realizowany przez jądro. Zamiast tego programy użytkowe odwołują się do *bibliotek systemowych* (ang. *system libraries*), w których z kolei następują niezbędne odwołania do usług systemu operacyjnego.

Biblioteki systemowe dostarczają wielorakich działań. Na najprostszym poziomie pozwalają one aplikacjom zamawiać systemowe usługi jądra. Odwołanie do systemu wymaga przekazania sterowania z nieuprzywilejowanego trybu użytkownika do uprzywilejowanego trybu jądra. Szczegóły tego przejścia zmieniają się w zależności od rodzaju architektury. Biblioteki dopilnowują zbierania argumentów wywołań funkcji systemowych i w razie konieczności – nadawania tym argumentom specjalnej postaci, niezbędnej do wykonania wywołania.

Biblioteki mogą również dostarczać bardziej złożonych wersji podstawowych funkcji systemowych. Na przykład wszystkie funkcje buforowanych działań na plikach w języku C są zrealizowane w bibliotekach systemowych,

umożliwiając bardziej zaawansowane sterowanie plikowymi operacjami wejścia-wyjścia niż to zapewniają podstawowe funkcje systemowe jądra. Biblioteki zawierają także podprogramy, które nie mają odpowiedników w odwołaniach do systemu, takie jak algorytmy sortowania, funkcje matematyczne i operacje na napisach. Wszystkie funkcje niezbędne do wspierania działań aplikacji standardu UNIX lub POSIX są zaimplementowane tutaj, w bibliotekach systemowych.

System Linux zawiera ponadto szeroki repertuar programów działających w trybie użytkownika – zarówno pomocy systemowych, jak i narzędzi dla użytkowników. W skład pomocy systemowych wchodzą wszystkie programy niezbędne do rozpoczęcia pracy systemu, takie jak programy do konfigurowania urządzeń sieciowych lub do ładowania modułów jądra. Programy serwerów pracujące nieustannie też zalicza się do pomocy systemowych. Obsługują one rozpoczęcie sesji przez użytkowników, sygnały nadchodzące z sieci oraz kolejki do drukarek.

Nie wszystkie standardowe pomoce pełnią kluczowe funkcje administracyjne w systemie. Środowisko użytkownika systemu UNIX zawiera wielką liczbę standardowych pomocy do wykonywania prostych, codziennych prac, takich jak wyprowadzanie zawartości katalogów, przemieszczanie i usuwanie plików lub wyświetlanie zawartości pliku. Bardziej skomplikowane narzędzia mogą przetwarzać teksty, na przykład porządkować dane tekstowe lub poszukiwać w tekście wejściowym zadanych wzorców. Łącznie pomoce te tworzą standardowy zestaw narzędziowy, którego użytkownicy mogą się spodziewać w każdym systemie unikowym. Choć nie wykonują one zadnej funkcji systemu operacyjnego, pozostają ważną częścią podstawowego systemu Linux.

22.3 ■ Moduły jądra

Jądro systemu Linux potrafi ładować i usuwać dowolną część swego kodu stosownie do potrzeb. Owe ładowalne moduły jądra działają w uprzywilejowanym trybie jądra, w wyniku czego mają pełny dostęp do wszystkich możliwości maszyny, na której są wykonywane. Teoretycznie nie ma żadnych ograniczeń dotyczących tego, co wolno zrobić modułowi jądra. Moduł taki zazwyczaj steruje pracą urządzenia lub jest systemem plików albo protokołem sieciowym.

Moduły jądra są przydatne z kilku powodów. Kod źródłowy systemu Linux jest w wolnym obiegu, więc każdy, kto ma ochotę pisać kod jądra, może skompilować zmienione jądro i rozpocząć jego działanie od nowa, aby załadować nowe funkcje. Jednakże powtórna komplikacja, konsolidacja i ładowanie całego jądra jest niewygodnym przedsięwzięciem, kiedy opracowuje się

tylko jakiś nowy moduł sterujący. Przy użyciu modułów jądra, nie trzeba wytwarzać nowego jądra w celu testowania nowego modułu sterującego. Moduł sterujący można skompilować niezależnie i załadować do już działającego jądra. Po napisaniu nowego modułu sterującego, można go naturalnie rozpowszechniać właśnie w postaci modułu, dzięki czemu każdy inny użytkownik może z niego skorzystać bez przebudowywania jądra w swoim systemie.

Ostatnia uwaga powoduje jeszcze inny skutek. Ponieważ jądro Linuxa jest objęte licencją GPL, nie wolno do niego dodawać prywatnych składowych, chyba że obejmie je również licencja GPL, wobec czego ich kod źródłowy stanie się dostępny na żądanie. Interfejs modułów jądra umożliwia osobom postronnym pisanie i upowszechnianie na własnych zasadach modułów sterujących urządzeniami lub systemów plików, których nie można rozpowszechniać na zasadach licencji GPL.

Moduły jądra pozwalają wreszcie na zestawianie systemu Linux z minimalnym, standardowym jądrem bez wbudowywania w nie żadnych dodatkowych modułów sterujących urządzeń. Dowolny moduł sterujący urządzenia, którego potrzebuje użytkownik, może być załadowany jawnie podczas rozruchu systemu lub wprowadzony automatycznie na żądanie systemu oraz usunięty, gdy przestanie być potrzebny. Na przykład moduł sterujący pamięcią CD-ROM może być załadowany wówczas, gdy zamontowano ją w systemie plików, i usunięty z pamięci operacyjnej, gdy pamięć CD-ROM zostanie od systemu plików odłączona.

Moduły w systemie Linux są wspomagane przez trzy następujące składowe:

- **Zarządzanie modułami** (ang. *module management*): Umożliwia wprowadzanie modułów do pamięci i ich kontakt z resztą jądra.
- **Rejestracja modułów sterujących** (ang. *driver registration*): Pozwala modułom poinformować jądro o dostępności nowego modułu obsługi urządzenia.
- **Mechanizm rozwijywania konfliktów** (ang. *conflict resolution*): Umożliwia różnym modułom sterującym rezerwowanie zasobów sprzętowych i chroni te zasoby przed przypadkowym użyciem przez inny moduł sterujący.

22.3.1 Zarządzanie modułami

Załadowanie modułu wymaga więcej niż tylko wprowadzenia jego binarnej zawartości do pamięci operacyjnej jądra. System musi także zapewnić, że każde odwołanie modułu do symboli jądra lub jego punktów wejściowych

zostanie uaktualnione tak, aby wskazywało na właściwe komórki w przestrzeni adresowej jądra. System Linux radzi sobie z owym uaktualnianiem odwołań przez podzielenie zadania ładowania modułu na dwie osobne sekcje: zarządzanie częściami kodu modułu w pamięci jądra oraz manipulowanie symbolami, do których modułom wolno się odwoływać.

Linux utrzymuje w jądrze wewnętrzną tablicę symboli. Tablica ta nie zawiera pełnego zbioru symboli zdefiniowanych w jądrze podczas ostatniej komplikacji. W zamian symbol musi być jawnie wyeksportowany przez jądro. Zbiór wyeksportowanych symboli tworzy dobrze określony interfejs, za pomocą którego moduł może współpracować z jądrem.

Chociaż eksportowanie symboli z funkcji jądra wymaga jawnego zamówienia ze strony programisty, ich import do modułu nie powoduje specjalnego zachodu. Osoba pisząca moduł korzysta ze standardowej możliwości konsolidacji zewnętrznej w języku C. Wszystkie symbole zewnętrzne, do których występują w module odwołania, lecz nie mające w nim deklaracji, są oznaczane w ostatecznym module binarnym tworzonym przez kompilator jako nie ustalone. Podczas wprowadzania modułu do jądra pomocniczy program systemowy odnajduje w nim najpierw owe nie ustalone symbole. Dla każdego symbolu wymagającego określenia przeszukuje się tablicę symboli jądra, po czym ich właściwe w aktualnie wykonywanym jądrze adresy są umieszczane w kodzie modułu. Dopiero po tym zabiegu moduł jest przekazywany jądru do załadowania. Jeżeli pomocniczy program systemowy mimo przeszukania tablicy jądra nie ustali wartości jakiegoś odwołania w module, to moduł zostaje odrzucony.

Ladowanie modułu przebiega w dwu etapach. Najpierw program ładujący modułu (ang. *module-leader*) zgłasza w jądrze zapotrzebowanie na potrzebny dla modułu ciągły obszar wirtualnej pamięci jądra. Jądro zwraca adres przydzielonej pamięci, więc program ładujący może go użyć w celu przemieszczenia kodu maszynowego modułu we właściwe miejsce. Wówczas drugie wywołanie systemowe przekazuje moduł do jądra wraz z niezbędną tablicą symboli, którą chce cn eksportować. Moduł jest teraz słowo po słowie kopowany do uprzednio zarezerwowanego obszaru, a tablica symboli jądra jest aktualizowana za pomocą nowych symboli, tak aby inne, jeszcze nie załadowane moduły mogły ewentualnie zrobić z nich użytkę.

Ostatnią składową zarządzania modułem jest procedura zamawiania modułu (ang. *module requestor*). Jądro definiuje interfejs komunikacyjny, z którym może się połączyć program zarządzania modułem. Dzięki temu połączeniu jądro będzie informować proces zarządzający o każdym zapotrzebowaniu ze strony innego procesu na działania modułu sterującego, systemu plików lub usług sieciowych nie mających aktualnej reprezentacji w pamięci, dając zarządcy możliwość załadowania danej usługi. Pierwotne zamówienie na usługę za-

kończy się z chwilą załadowania modulu. Proces zarządzający odpytuje regularnie jądro, sprawdzając, czy załadowany moduł pozostaje w użyciu, a gdy moduł nie jest już potrzebny, powoduje jego usunięcie z pamięci.

22.3.2 Rejestrowanie modulu sterującego

Moduł po załadowaniu pozostaje po prostu odizolowanym obszarem pamięci do czasu, aż zapozna resztę jądra z funkcjami, którymi rozporządza. Jądro utrzymuje dynamiczne tablice wszystkich znanych modułów sterujących (programów obsługi urządzeń) i dostarcza zestawu procedur umożliwiających dodawanie do nich lub usuwanie z nich modułów w dowolnej chwili. Jądro zapewnia, że przy ładowaniu modułu zostanie wywołana jego procedura rozruchowa, jak również wywołuje procedurę czyszczącą przed usunięciem modułu. To właśnie te procedury odpowiadają za rejestrowanie funkcji modułu.

Istnieje wiele typów modułów sterujących, które moduł (jądra) może zarejestrować. Jeden moduł może rejestrować moduły sterujące określonego typu lub każdego z tych typów i w razie potrzeby może zarejestrować więcej niż jeden moduł sterujący. Moduł sterujący urządzenia może na przykład wymagać zarejestrowania dwu oddzielnych mechanizmów dostępu do urządzenia. Tablice rejestracyjne zawierają następujące elementy:

- **Moduły sterujące urządzeń:** Moduły te obejmują urządzenia znakowe (takie jak drukarki, terminali lub myszki), urządzenia blokowe (w tym wszystkie programy obsługi dysków) oraz urządzenia interfejsów sieciowych.
- **Systemy plików:** Za system plików można uważać każde oprogramowanie, które realizuje procedury wywołań wirtualnego systemu plików Linux. Może ono implementować format plików przechowywanych na dysku, lecz równie dobrze może to być sieciowy system plików, którego zawartość jest generowana na żądanie – na przykład taki jak linuksowy system plików o nazwie proc.
- **Protokoły sieciowe:** Moduł może realizować cały protokół sieciowy, na przykład w rodzaju IPX, lub po prostu nowy zbiór reguł filtrowania pakietów przez sieciową zaporę ogniomową.
- **Format binarny:** Format ten określa zasady rozpoznawania i ładowania pliku wykonywalnego nowego typu.

Ponadto moduł może zarejestrować nowy zbiór wejść w tablicach sysctl i proc, aby umożliwić jego konfigurowanie dynamiczne (p. 22.7.3).

22.3.3 Rozwiązywanie konfliktów

Komercyjne implementacje systemu UNIX są na ogół sprzedawane tak, aby pracowały na sprzęcie dostarczonym przez samego sprzedawcę. Zaletą korzystania z usług jednego dostawcy jest to, że ma on dobre rozumnianie co do możliwych konfiguracji sprzętu. Składniki sprzęt typu IBM PC występuje w nader licznych konfiguracjach i obsługuje w rozmaitym stopniu sterujące urządzenia, takie jak karty sieciowe, sterowniki SCSI czy adaptery wyświetlania obrazu. Problem zarządzania konfiguracją sprzętu staje się jeszcze poważniejszy w sytuacji, gdy programy obsługujące urządzenia są same dostarczane w formie modularnej, ponieważ aktualnie działający zbiór urządzeń podlega dynamicznym zmianom.

Linux wprowadza centralny mechanizm rozwiązywania konfliktów, pomagając rozstrzygać o dostępie do pewnych zasobów sprzętowych. Ma to zapewnić, co następuje:

- ochronę modułów przed rywalizacją o dostęp do zasobów sprzętowych;
- zapobieganie zaburzaniu działania istniejących modułów sterujących przez autosondy (ang. *autoprobes*), tj. testy inicjowane przez moduły sterujące urządzeń w celu automatycznego wykrywania konfiguracji sprzętu;
- rozwiązywanie konfliktów między wieloma modułami sterującymi usiłującymi skorzystać z tego samego sprzętu; na przykład moduł sterujący równoległą linią drukarki oraz sieciowy moduł sterujący PLIP (ang. *parallel-line IP*) mogą próbować użyć tego samego, równoległego portu drukarki.

Aby osiągnąć te cele, jądro utrzymuje wykazy przydzielonych zasobów sprzętowych. Liczba możliwych portów wejścia-wyjścia w komputerze PC (czyli adresów w jego sprzętowej przestrzeni adresów wejścia-wyjścia) oraz linii przerwań i kanałów DMA jest ograniczona. Gdy którykolwiek moduł sterujący urządzeniami zechce sięgnąć po taki zasób, wówczas zakłada się, że wpierw zarezerwuje go w bazie danych jądra. Wymaganie to pozwala administratorowi systemu w razie potrzeby dokładnie określić, które zasoby przydzielono któremu modułowi sterującemu w danej chwili.

Oczekuje się, że moduł będzie zawczasu stosował ten mechanizm w celu rezerwowania wszelkich potrzebnych mu do użytku zasobów sprzętowych. W przypadku odrzucenia rezerwacji z powodu braku zasobu lub jego zablokowania decyzja o dalszym postępowaniu należy do modułu. Może on uznać niepowodzenie działań wstępnych i zażądać swojego usunięcia lub kontynuować działanie za pomocą zastępczych zasobów sprzętowych.

22.4 ■ Zarządzanie procesami

Proces jest podstawowym kontekstem, w obrębie którego obsługuje się w systemie operacyjnym wszystkie zamawiane przez użytkownika czynności. W celu zapewnienia zgodności z innymi systemami uniksowymi system Linux musi koniecznie korzystać z modelu procesu podobnego do stosowanego w tych systemach. Jednak w kilku miejscach postępowanie systemu Linux odbiega tu nieco od unikowej normy. W tym punkcie dokonamy przeglądu tradycyjnego modelu procesu systemu UNIX z p. 21.3.2 i przedstawimy właściwy dla systemu Linux model wątków.

22.4.1 Model procesu oparty na funkcjach fork i exec

Podstawową zasadą zarządzania procesami w systemie UNIX jest podział na dwie odrębne operacje: tworzenie procesów i wykonywanie nowych programów. Nowy proces jest tworzony za pomocą funkcji systemowej `fork`, a wykonanie nowego programu następuje wskutek wywołania funkcji `execve`. Są to dwie wyraźnie odrębne funkcje. Można utworzyć nowy proces za pomocą `fork`, lecz nie spowodować wykonania nowego programu – nowy podproces będzie wtedy wykonywał po prostu ten sam program co pierwszy proces, tzn. proces macierzysty. I podobnie, wykonanie nowego programu nie wymaga uprzedniego utworzenia nowego procesu – dowolny proces może w każdej chwili wywołać funkcję `execve`. Aktualnie wykonywany program zostanie wówczas natychmiast zakończony, a w kontekście istniejącego procesu rozpoczęte wykonywanie nowego programu.

Zaletą tego modelu jest wielka prostota. Zamiast określić wszystkie szczegóły środowiska nowego programu w wywoaniu systemowym powodującym jego wykonanie, nowy program po prostu działa w istniejącym środowisku. Jeśli proces macierzysty ma zamiar zmienić środowisko, w którym ma pracować nowy program, to może się rozwidlić (ang. *fork*), po czym – wykonując wciąż ten sam program w procesie potomnym – może wykonać dowolne wywołania systemowe potrzebne mu do zmiany owego procesu potomnego przed ostatecznym wykonaniem nowego programu.

W systemie UNIX proces obejmuje zatem całość informacji, które system operacyjny musi utrzymywać, aby śledzić kontekst wykonania indywidualnego programu. W systemie Linux kontekst ten można podzielić na pewną liczbę określonych części. Mówiąc ogólnie, cechy procesu dzielą się na trzy grupy: tożsamość, jego środowisko i kontekst.

22.4.1.1 Tożsamość procesu

Na tożsamość procesu składają się przed wszystkim następujące elementy:

- **Identyfikator procesu** (ang. *process identifier* – PID): Każdy proces ma niepowtarzalny identyfikator. Aplikacja odwołująca się do systemu w celu sygnalizacji, zmiany lub zaczekania na inny proces określa procesy systemowi operacyjnemu za pomocą identyfikatorów PID. Dodatkowe identyfikatory kojarzą proces z grupą procesów (na ogół drzewem procesów rozwidlonych za pomocą poleceń wydanych przez tego samego użytkownika) oraz z sesją konwersacyjną.
- **Uwierzytelnienia** (ang. *credentials*): Z każdym procesem musi być skojarzony identyfikator użytkownika oraz jeden lub więcej identyfikatorów grup (grupy użytkowników, lecz nie procesów, są omówione w p. 10.4.2) określających prawa procesu do sięgania po zasoby systemowe i pliki.
- **Indywidualność** (ang. *personality*): Indywidualności procesów nie odnajdziemy w tradycyjnych systemach UNIX, za to w systemie Linux każdy proces ma przypisany identyfikator indywidualności, który może nieco zmieniać semantykę niektórych funkcji systemowych. Z indywidualności korzystają przed wszystkim biblioteki emulacji, aby powodować, że funkcje systemowe będą zgodne z pewnymi ich szczególnymi odmianami występującymi w systemach uniksowych.

Ograniczoną kontrolę nad większością z tych identyfikatorów sprawuje sam proces. Jeżeli proces chce utworzyć nową grupę lub rozpoczęć nową sesję, to może zmienić identyfikatory grupy procesów i sesji. Jego uwierzytelnienia mogą ulegać zmianom stosownie do odpowiednich wymogów bezpieczeństwa. Jednakże podstawowy identyfikator PID procesu pozostaje niezmieniony i jednoznacznie identyfikuje proces aż do jego zakończenia.

22.4.1.2 Środowisko procesu

Proces dziedziczy środowisko po procesie macierzystym; składa się ono z dwóch wektorów zakończonych znakiem pustym: wektora argumentów i wektora środowiska. Wektor argumentów to po prostu wykaz argumentów wpisanych przez użytkownika w wierszu wywołania programu. Na mocy konwencji zaczyna się on od nazwy samego programu. Wektor środowiska jest wykazem par „NAZWA=WARTOŚĆ”, kojarzących nazwane zmienne środowiskowe z dowolnymi wartościami tekstowymi. Środowiska nie przechowuje się w pamięci jądra, lecz pamięta w trybie użytkownika w prywatnej przestrzeni adresowej procesu jako pierwszą daną na szczytce jego stosu.

Wektorów argumentów i środowiska nie zmienia się w chwili tworzenia nowego procesu. Nowo utworzony proces potomny odziedziczy takie samo środowisko, jakim dysponuje jego twórca. Jednak przy rozpoczynaniu nowego programu jest tworzone całkowicie nowe środowisko. Wywołując funkcję execve, proces musi dostarczyć środowisko dla nowego programu. Zmienne tego środowiska są przekazywane nowemu programowi poprzez jądro, zastępując dotychczasowe środowisko procesu. Poza tym wektory środowiska i argumentów z wiersza wywołania programu są przez jądro pozostawiane własnymu losowi – ich interpretacja należy w całości do działających w trybie użytkownika operacji bibliotecznych i aplikacji.

Przekazywanie zmiennych środowiskowych z jednego procesu do drugiego oraz ich dziedziczenie przez proces potomny dostarcza elastycznych środków przekazywania informacji do różnych składowych oprogramowania systemowego na poziomie użytkowym. Niektóre ważne zmienne środowiskowe mają konwencjonalne znaczenie dla powiązanych z nimi części oprogramowania systemowego. Na przykład wartość zmiennej TERM określa typ terminalu pośredniczącego w sesji konwersacyjnej użytkownika, więc wiele różnych programów korzysta z tej zmiennej, aby ustalić, w jaki sposób wykonywać operacje wyświetlania informacji na terminalu użytkownika, m.in. jak poruszać kursemorem lub jak przewijać porcję tekstu na ekranie. Zmienna LANG jest używana do określania, w jakim języku wyświetlać komunikaty systemowe programów umożliwiających wielojęzyczną komunikację.

Ważną cechą mechanizmu zmiennych środowiskowych jest to, że umożliwia on dostosowywanie systemu operacyjnego do specjalnych potrzeb przez zestawianie go z procesów, a nie konfigurowanie jako systemu rozpatrywanego całościowo. Użytkownicy mogą wybierać własne języki lub ulubione edytory niezależnie od siebie.

22.4.1.3 Kontekst procesu

Tożsamość procesu i jego środowisko są zazwyczaj określane podczas tworzenia procesu i nie zmieniają się aż do jego zakończenia. Procesowi wolno jednak w razie potrzeby zmienić pewne aspekty jego tożsamości lub środowiska. Z kolei kontekst procesu, jako stan wykonywanego procesu rozpatrywany w dowolnej chwili, zmienia się nieustannie.

- **Kontekst planowania:** Najważniejszą częścią kontekstu procesu jest jego kontekst planowania, czyli informacje potrzebne planiście (programowi szeregującemu) do zawieszania i wznowiania procesu. Należą do nich przechowane kopie wszystkich rejestrów procesu. Rejestry zmiennopozycyjne przechowuje się oddzielnie i odtwarza tylko wtedy, kiedy są potrzebne, więc proces nie korzystający z arytmetyki zmiennopozycyjnej

nie ponosi kosztów przechowywania tego stanu. Kontekst planowania zawiera również informacje o priorytecie planowania oraz o wszelkich nie obsłużonych sygnatach, oczekujących na dostarczenie do procesu. Zasadniczą częścią kontekstu planowania jest jądrowy stos procesu – osobny obszar pamięci jądra, zarezerwowany do wyłącznego użytku przez kod w trybie jądra. Ze stosu tego korzystają zarówno funkcje systemowe, jak i przerwania występujące podczas wykonywania procesu.

- **Rozliczanie:** Jądro utrzymuje informacje o zasobach zużywanych na bieżąco przez każdy z procesów, jak również o ich łącznej ilości skonsumentowanej przez proces w całym dotychczasowym jego przebiegu.
- **Tablica plików:** Tablica plików zawiera wskaźniki do jądrowych struktur plików. Wykonując systemowe operacje wejścia-wyjścia, procesy odwołują się do plików za pomocą indeksów do tej tablicy.
- **Kontekst systemu plików:** Podeczas gdy tablica plików ujmuje istniejące, otwarte pliki, kontekst systemu plików odnosi się do zamówień na otwarcie nowych plików. Przechowuje się w tym celu informacje o bieżącym katalogu głównym oraz o katalogach używanych zastępczo do poszukiwania nowych plików.
- **Tablica obsługi sygnałów:** Systemy uniksowe są w stanie dostarczać procesowi asynchroniczne sygnały jako odpowiedzi na rozmaite zdarzenia zewnętrzne. Tablica obsługi sygnałów określa procedurę w przestrzeni adresowej procesu, która ma być wywołana po nadaniu konkretnego sygnału.
- **Kontekst pamięci wirtualnej:** Kontekst pamięci wirtualnej opisuje całą zawartość prywatnej przestrzeni adresowej procesu. Przedstawimy go w p. 22.6.

22.4.2 Procesy i wątki

Większość nowoczesnych systemów operacyjnych umożliwia stosowanie zarówno procesów, jak i wątków. Aczkolwiek dokładne określenie różnic między tymi dwoma pojęciami zmienia się w zależności od implementacji, można jednak zdefiniować główną różnicę: *procesy* reprezentują wykonywanie poszczególnych programów, *wątki* zaś reprezentują oddzielne, współbieżne konteksty wykonywania w ramach jednego procesu wykonującego dany program.

Każde dwa osobne procesy mają własne, niezależne przestrzenie adresowe – nawet gdy korzystają ze wspólnej pamięci w celu dzielenia części (lecz nie całości) zawartości ich pamięci wirtualnej. Przeciwnie, dwa wątki tego

samego procesu będą dzielić *tą samą* przestrzeń adresową (a nie tylko podobne przestrzenie adresowe – każda zmiana w wyglądzie pamięci wirtualnej wykonana przez jeden z wątków będzie natychmiast widoczna dla innych wątków w procesie, ponieważ w rzeczywistości istnieje tylko jedna przestrzeń adresowa, w której są wykonywane wszystkie wątki).

Wątki można zrealizować na kilka różnych sposobów. Wątek może być zaimplementowany w jądrze systemu operacyjnego w postaci obiektu należącego do procesu lub może być jednostką w pełni niezależną. Wątek może być zrealizowany poza jądrem – wątki mogą być w pełni urzeczywistniane przez aplikację lub kod biblioteczny z pomocą dostarczanych przez jądro przerwań zegarowych.

Jądro systemu Linux w prosty sposób radzi sobie z różnicą między procesami a wątkami – stosuje dokładnie tę samą wewnętrzną reprezentację dla każdego z nich. Wątek jest po prostu nowym procesem, któremu przychodzi dzielić ze swoim rodzicem tę samą przestrzeń adresową. Rozróżnienia między procesem i wątkiem dokonuje się tylko podczas tworzenia wątku, wykonywanego za pomocą systemowej funkcji `clone`. Podczas gdy funkcja `fork` tworzy nowy proces z własnym, całkowicie nowym kontekstem, wywołanie `clone` tworzy proces z odrębną tożsamością, lecz któremu pozwala się dzielić struktury danych jego rodzica.

Rozróżnienie to jest możliwe, ponieważ Linux nie przechowuje całego kontekstu w głównej strukturze danych, pamiętając go w zamian w postaci niezależnych podkontekstów. Kontekst systemu plików, tablica deskryptorów plików, tablica obsługi sygnałów i kontekst pamięci wirtualnej są przechowywane w oddzielnych strukturach danych. Struktura danych procesu zawiera po prostu wskaźniki do tamtych struktur, więc dany podkontekst może być łatwo dzielony przez dowolną liczbę procesów za pomocą odpowiednich do niego wskazań.

Funkcja systemowa `clone`, tworząc nowy proces, przyjmuje argument określający, które podkonteksty należy skopiować, a które mają być wspólne. Nowy proces zawsze otrzymuje nową tożsamość i nowy kontekst planowania; stosownie do podanych parametrów może jednak utworzyć nowy podkontekst struktur danych, których wartości zostaną skopiowane z procesu macierzystego, lub korzystać z tego samego podkontekstu struktur danych co jego rodzic. Funkcja systemowa `fork` nie jest niczym więcej niż specjalnym przypadkiem funkcji `clone`, kopującym wszystkie podkonteksty i nie powodującym dzielenia żadnego z nich. Użycie funkcji `clone` pozwala na dokładniejsze określenie w aplikacji, co ma być przez dwa wątki użytkowane wspólnie.

Grupy robocze POSIX zdefiniowały interfejs programowy, zawarty w standardzie POSIX.1c, umożliwiający aplikacjom działania wielowątkowe. Biblioteki systemu Linux dostarczają dwóch osobnych mechanizmów, które

realizują ten standard różnymi sposobami. W programie użytkowym można wybierać między pakietem wątków działających w linuksowym trybie użytkownika a pakietem wątków zrealizowanych w jądrze. Biblioteka wątków poziomu użytkownika pozwala na unikanie kosztów planowania wątków przez jądro i wywołań jądra przy ich współpracy, ale ogranicza ją to, że wszystkie wątki działają w jednym procesie. W bibliotece wątków zrealizowanych przez jądro robi się użytek z funkcji systemowej `clone`, za pomocą której jest realizowany ten sam interfejs programowy, lecz dzięki tworzeniu wielu kontekstów planowania ma ona tę zaletę, że umożliwia aplikacjom jednocześnie wykonywanie wątków na wielu procesorach systemu wieloprocesorowego. Pozwala ona również na jednocześnie wykonywanie przez wiele wątków systemowych funkcji jądra.

22.5 ■ Planowanie

Przez planowanie rozumie się przydzielanie czasu jednostki centralnej różnym zadaniom systemu operacyjnego. Na ogół rozpatrujemy planowanie jako ciąg działań i przerwań procesów, istnieje jednak pewien aspekt planowania istotny dla systemu Linux, mianowicie wykonywanie różnych zadań jądra. Wykonywanie zadań jądra obejmuje zarówno zadania zlecone przez działający proces, jak i zadania działające wewnętrznie na zamówienie modułu sterującego jakiegoś urządzenia.

22.5.1 Synchronizacja jądra

Sposób, w jaki jądro planuje własne operacje, jest zasadniczo różny od planowania procesów. Zamówienie na działanie w trybie jądra można wykonać dwoma sposobami. Wykonywany program może zamówić usługę systemu operacyjnego jawnie – za pomocą wywołania systemowego – lub niejawnie, na przykład wskutek braku strony. Może też wystąpić przerwanie pochodzące od modułu sterującego urządzenia, które spowoduje, że jednostka centralna zacznie wykonywać kod zdefiniowanej w jądrze procedury obsługi przerwania.

Problem, z którym ma do czynienia jądro, polega na tym, że wszystkie te rozmaite zadania mogą próbować sięgać po te same struktury danych. Jeśli podczas wykonywania procedury obsługi przerwania któryś z zadań jądra jest w trakcie kontaktowania się z pewną strukturą danych, to owa procedura obsługi nie może czytać ani zmieniać tych samych danych bez ryzyka ich uszkodzenia. Fakt ten wiąże się z koncepcją sekcji krytycznych, tj. fragmentów kodu korzystających ze wspólnych danych, których nie wolno wykonywać jednocześnie.

W rezultacie synchronizacja jądra jest znacznie bardziej skomplikowana niż zwykłe planowanie procesów. Potrzebne są regulacje pozwalające na wykonywanie sekcji krytycznych jądra w sposób wzajemnie nie kolidujący.

Pierwsza część rozwiązania tego problemu w systemie Linux polega na uczynieniu zwykłego kodu jądra niewywłaszczałnym. Po nadaniem do jądra przerwania od czasomierza wywołuje ono z reguły proces planisty, w związku z czym może zawiesić aktualnie wykonywany proces i wznowić działanie jakiegoś innego – jest to naturalny w systemie UNIX podział czasu. Jednakże gdy przerwanie zegarowe zostanie odebrane w czasie, gdy proces wykonyuje jądrową, systemową procedurę obsługi, nowe zaplanowanie nie następuje natychmiast. Zamiast tego ustawia się w jądrze znacznik `need_resched`, aby poinformować jądro o zapotrzebowaniu na pracę planisty po zakończeniu wywołania systemowego, kiedy to ma nastąpić przekazanie sterowania do trybu użytkownika.

Jeśli tylko rozpoczęcie się wykonywanie kawałka kodu jądra, to gwarantuje się, że będzie to jedyny aktywny fragment kodu jądra do czasu, aż wystąpi jedno z następujących zdarzeń:

- przerwanie;
- brak strony;
- wywołanie działania planisty przez sam kod jądra.

Trudności z przerwaniami dotyczą tylko sytuacji, gdy one same zawierają sekcje krytyczne. Przerwanie od czasomierza nigdy nie powoduje bezpośrednio zaplanowania procesu. Stanowi ono jedynie zamówienie na wykonanie planowania w późniejszym terminie. Tak więc żadne z nadchodzących przerwań nie może zaburzyć porządku wykonania nieprzerywalnego kodu jądra. Po zakończeniu obsługi przerwania podejmie się po prostu wykonywanie tego samego kodu jądra, który był wykonywany w chwili przejęcia przerwania.

Potencjalny problem stanowią braki stron. Jeśli procedura jądrowa próbuje czytać lub zapisywać pamięć użytkownika, to może się jej przytrafić brak strony wymagający obsługi ze strony wejścia-wyjścia, wobec czego aktywny proces zostanie zawieszony do czasu zakończenia operacji wejścia-wyjścia. Podobnie, gdy procedura obsługująca funkcję systemową wywołuje planistę podczas swojego pobytu w jądrze – bądź to przez jawnie odwołanie się do kodu planisty, bądź niejawnie, przez wywołanie funkcji oczekiwania na zakończenie operacji wejścia-wyjścia – wówczas proces zostanie zawieszony i nastąpi ponowne zaplanowanie procesora. Kiedy proces stanie się ponownie gotowy do działania, będzie kontynuował pracę w trybie jądra, rozpoczynając ją od instrukcji występującej po wywołaniu planisty.

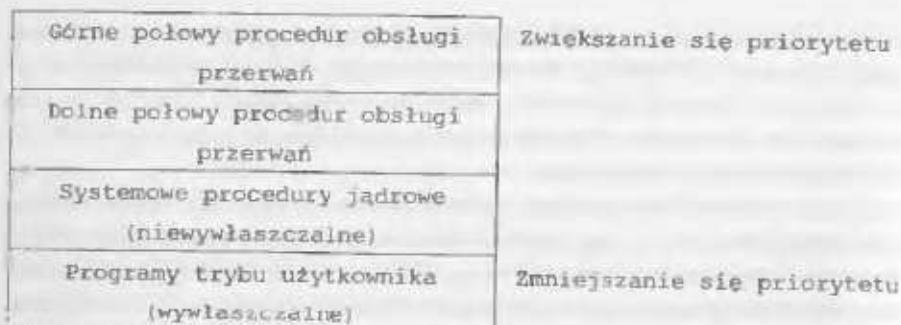
O kodzie jądra można zatem założyć, że nigdy nie zostanie on wywieszony przez inny proces i że nie trzeba stosować żadnych specjalnych środków ochrony sekcji krytycznych. Jedynym wymaganiem jest, aby sekcje krytyczne nie zawierały odwołań do pamięci użytkownika ani oczekiwania na koniec operacji wejścia-wyjścia.

Druga technika stosowana w systemie Linux do ochrony sekcji krytycznych dotyczy tych sekcji krytycznych, które występują w procedurach obsługi przerwań. Podstawowym narzędziem używanym tu do osiągnięcia odpowiedniego działania jest sprzęt nadzorujący przerwania procesora. Przez wprowadzenie zakazu przerwań w trakcie wykonywania sekcji krytycznej jądro gwarantuje, że jej wykonanie przebiegnie bez ryzyka wspólnie dostępu do dzielonych struktur danych.

Wylączanie przerwań pociąga za sobą pewne sankcje. Rozkazy włączania i wylaczania przerwań są w większości architektur sprzętowych kosztowne. Co więcej, dopóki przerwania pozostają wyłączone, dopóty wszystkie operacje wejścia-wyjścia podlegają zawieszeniu i każde urządzenie czekające na obsługę będzie oczekwać aż do przywrócenia wykrywania przerwań, co pogarsza wydajność. W jądrze systemu Linux zastosowano architekturę synchronizacji umożliwiającą wykonywanie długich sekcji krytycznych bez wylaczania przerwań na czas trwania całej sekcji. Możliwość ta jest szczególnie użyteczna w oprogramowaniu sieci. Przerwanie w module sterującym urządzenia sieciowego może sygnalizowaćadejscie całego pakietu sieciowego, co może powodować konieczność wykonania w procedurze obsługi przerwania dużej liczby instrukcji dekodujących, określających trasę i wysyłających pakiet w dalszą drogę.

Linux implementuje tę architekturę przez wyodrębnianie w procedurach obsługi przerwań dwu części: górnej i dolnej połowy. *Góra połowa* (ang. *top half*) jest zwykłą procedurą obsługi przerwania; jej wykonanie przebiega z włączonymi rekurencyjnymi przerwaniami (przerwania o wyższym priorytecie mogą tę procedurę przerywać, lecz przerwania o tym samym lub niższym priorytecie są zablokowane). *Dolna połowa* (ang. *bottom half*) procedury obsługiowej przebiega z włączonymi wszystkimi przerwaniami i jest nadzorowana przez miniaturowego planistę, który zapewnia, że dolne połowy nigdy nie przerywają się wzajemnie. Planista dolnej połowy jest wywoływany automatycznie przy każdym wychodzeniu z procedury obsługi przerwania.

Oddzielenie takie oznacza, że dowolnie złożone przetwarzanie, które ma być wykonane w odpowiedzi na przerwanie, może być dokonane przez jądro bez obawy, że ono samo zostanie przerwane. Jeżeli podczas wykonywania dolnej połowy wystąpi inne przerwanie, to przerwanie to może zażądać wykonania tej samej dolnej połowy, lecz nastąpi to dopiero po zakończeniu bieżącego przebiegu. Kazde wykonanie dolnej połowy może być przerwane przez górną połowę, lecz nigdy nie może go przerwać analogiczna dolna połowa.



Rys. 22.2 Poziomy ochrony przed przerwaniami

Architektura górnej i dolnej połowy jest uzupełniana przez mechanizm wyłączania wybranych dolnych połów podczas wykonywania zwykłego, pierwszoplanowego kodu jądra. Za pomocą tej metody można łatwo zaprogramować sekcje krytyczne w jądrze: sekcje krytyczne procedur obsługi przerwań mogą być zakodowane jako dolne połowy, a gdy jądro, działając pierwszoplanowo, zechce wejść do sekcji krytycznej, wówczas może wydać niezbędne zakazy działania dowolnych z dolnych połów, aby uchronić się przed przerwaniem mu przez jakikolwiek inną sekcję krytyczną. Pod koniec sekcji krytycznej jądro może zezwolić na działanie dolnych połów i podjąć wykonywanie dowolnych zadań określonych przez dolne połowy spośród ustawionych w kolejce przez górne połowy procedur obsługi przerwań podczas wykonywania sekcji krytycznej.

Na rysunku 22.2 są wymienione różne poziomy ochrony przerwań w jądrze. Każdy poziom może być przerwany przez kod wykonywany na wyższym poziomie, ale nigdy nie będzie przerwany przez kod wykonywany na tym samym lub niższym poziomie (poza kodem w trybie użytkownika – procesy użytkowe mogą być zawsze wyłączane przez inne procesy w chwilach określonych przerwaniami zegarowymi służącymi podziałowi czasu).

22.5.2 Planowanie procesów

Gdy jądro osiąga punkt, w którym ma nastąpić kolejne zaplanowanie – czy to z powodu wystąpienia przerwania sygnaлизującego potrzebę zaplanowania, czy wskutek zablokowania się wykonywanego w nim procesu w oczekiwaniu na pewien sygnał pobudzający – musi zadecydować, który proces ma być wykonywany w dalszym ciągu. Linux ma dwa oddzielne algorytmy planowania procesów. Jednym z nich jest algorytm z podziałem czasu do sprawiedliwego planowania z wyłączaniami działania wielu procesów, a drugi za-

projektowano z myślą o zadaniach czasu rzeczywistego, w których priorytety bezwzględne są ważniejsze niż sprawiedliwość.

W skład tożsamości każdego procesu wchodzi klasa planowania definiująca, który z tych algorytmów ma być zastosowany do procesu. Klasy planowania występujące w systemie Linux są zdefiniowane w rozszerzeniach standardu POSIX dotyczących obliczeń w czasie rzeczywistym (norma POSIX.4 – obecnie znana pod nazwą POSIX.1b).

Dla procesów z podziałem czasu system Linux używa algorytmu priorytetowego, opartego na kredytowaniu. Każdy proces otrzymuje pewną wielkość kredytu na korzystanie z procesora. Kiedy należy wybrać nowe zadanie do wykonywania, wtedy wybiera się proces z największym kredytem. Przy każdym przerwaniu pochodząącym od czasomierza aktualnie wykonywanego proces traci jednostkę kredytu; gdy jego kredyt spadnie do zera, zostaje zawieszony i następuje wybranie innego procesu.

Jesli żaden z procesów zdolnych do działania nie ma kredytu, to Linux wykonuje operację wtórnego kredytowania, udzielając kredytu każdemu procesowi w systemie (a nie tylko procesom gotowym do działania) według następującego wzoru:

$$\text{kredyt} = \frac{\text{kredyt}}{2} + \text{priorytet}$$

Algorytm ten dąży do zmieszania dwu czynników: historii procesu i jego priorytetu. Po zastosowaniu algorytmu zostaje zachowana połowa kredytu, którym proces rozporządzał w wyniku poprzedniego kredytowania, co odzwierciedla w jakiś sposób historię ostatniego zachowania się procesu. Procesy, które działają cały czas, zużywają swoje kredyty gwałtownie, natomiast procesy spędżające dużą część czasu w zawieszeniu mogą akumulować kredyty wskutek wielu kolejnych kredytowań; ich kredyty stają się większe po nowym kredytowaniu. Taki system kredytowania automatycznie preferuje procesy interakcyjne, uzałożone od wejścia-wyjścia, dla których istotny jest czas odpowiedzi.

Użycie priorytetu procesu w obliczaniu nowego kredytu pozwala na dobre strojenie tego priorytetu. Drugoplanowe zadania wsadowe mogą otrzymywać niskie priorytety, będą więc automatycznie uzyskiwały mniej kredytów niż interakcyjne zadania użytkowników i będą otrzymywać mniejszy procent czasu procesora niż podobne zadania o wyższych priorytetach. Linux stosuje ten system priorytetów do implementacji standardowego w systemie UNIX mechanizmu priorytetów procesów opartego na działaniu polecenia *nice*.

Większą prostotę widać również w linuksowym planowaniu czasu rzeczywistego. Linux realizuje dwie klasy planowania w czasie rzeczywistym wymagane przez standard POSIX: algorytm „pierwszy na wejściu – pierwszy

na wyjściu" (FIFO) oraz algorytm rotacyjny (p. 5.3.1 i 5.3.4). W obu przypadkach oprócz klasy planowania każdy proces ma również priorytet. Jednak procesy planowane z podziałem czasu mimo różnych priorytetów zawsze mogą w pewnym stopniu wspólnie postępować do przodu. Planista czasu rzeczywistego wykonuje zawsze proces o największym priorytecie. Spośród procesów o równych priorytetach planista wybiera do wykonania proces, który oczekiwany najdłużej. Jedyną różnicą między planowaniem FIFO a rotacyjnym jest to, że procesy FIFO działają do końca lub do zablokowania, natomiast w algorytmie rotacyjnym proces zostanie po chwili wywłaszczyony i przesunięty na koniec kolejki planowania, tak więc procesy rotowane o równych priorytetach będą automatycznie dzieliły czas między siebie.

Zauważmy, że planowanie w czasie rzeczywistym w systemie Linux jest łagodne, a nie rygorystyczne. Planista oferuje ścisłe gwarancje odnosnie do względnych priorytetów procesów czasu rzeczywistego, lecz jądro nie daje żadnych gwarancji co do szybkości zaplanowania procesu czasu rzeczywistego od chwili, gdy stanie się on gotowy do działania. Pamiętajmy, że kod jądra Linux nigdy nie może być wywłaszczyony przez kod poziomu użytkownika. Po nadejściu przerwania budzącego proces czasu rzeczywistego w czasie gdy jądro wykonuje właśnie funkcję systemową na zamówienie innego procesu, proces czasu rzeczywistego musi zwyczajnie poczekać na zakończenie bieżącej funkcji systemowej lub jej zablokowanie.

22.5.3 Wieloprzetwarzanie symetryczne

Jądro Linux 2.0 było pierwszym stabilnym jądrem systemu Linux wspierającym *symetryczny sprzęt wieloprocesorowy* (ang. *symmetric multiprocessor – SMP*). Poszczególne procesy lub wątki mogą działać równolegle na oddzielnych procesorach. Jednak w celu spełnienia niewywłaszczeniowych wymagań synchronizacji jądra implementacja SMP w tym jądrze narzuca ograniczenie, że w danej chwili tylko jeden procesor może wykonywać kod w trybie jądra. Do wymuszenia tej zasady w symetrycznym wieloprocesorze znajduje zastosowanie jedna wirująca blokada w jądrze. Nie powoduje ona problemów w zadaniach wykonujących intensywne obliczenia, może jednak być wąskim gardłem dla zadań wymagających dużej aktywności jądra.

Podeczas pisania tej książki jednym z ważniejszych założen projektowych rozwojowego jądra Linux 2.1 było zwiększenie skalowalności implementacji SMP przez rozbicie jednej wirującej blokady w jądrze na wiele zamków, z których każdy chroni przed ponownym udostępnieniem tylko niewielki podzbiór jądrowych struktur danych. Dzięki zastosowaniu takich technik nowsze jądra rozwojowe umożliwiają naprawdę jednoczesne wykonywanie kodu w trybie jądra przez wiele procesorów.

22.6 ■ Zarządzanie pamięcią

Zarządzania pamięcią dotyczą w systemie Linux dwie składowe. Pierwsza z nich – system zarządzania pamięcią fizyczną – zajmuje się przydzielaniem i zwalnianiem stron, ich grup i małych bloków pamięci. Druga składowa obsługuje pamięć wirtualną, będącą odwzorowaniem pamięci operacyjnej na przestrzeń adresową wykonywanych procesów.

Opisemy obie te składowe, a następnie przeanalizujemy mechanizm, za pomocą którego ładowalne części nowego programu są wprowadzane do wirtualnej pamięci procesu jako efekt działania systemowej funkcji `exec`.

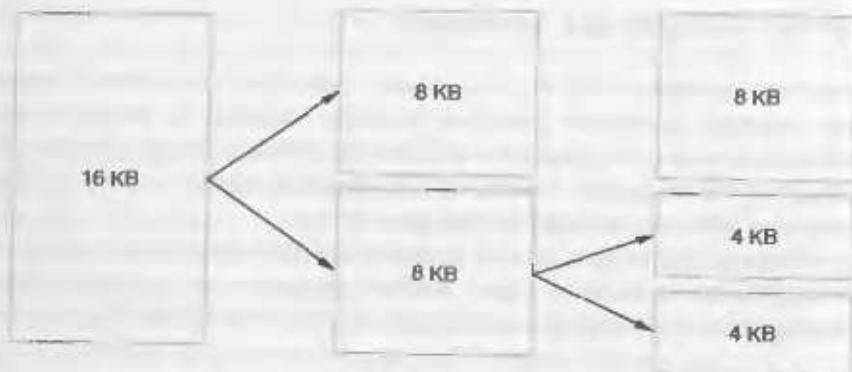
22.6.1 Zarządzanie pamięcią fizyczną

Zarządcą podstawowej pamięci fizycznej w jądrze systemu Linux jest dyspozytor stron (ang. *page allocator*)^{*}. Odpowiada on za przydział i zwalnianie wszystkich fizycznych stron i jest w stanie przydzielać na zamówienie partie stron fizycznie ze sobą sąsiadujących. Dyspozytor ten stosuje algorytm *sąsiednich stert* (ang. *buddy-heap*) w celu rejestrowania dostępnych stron fizycznych^{**}. Dyspozytor sąsiednich stert łączy w pary przyległe do siebie jednostki przydziału – stąd bierze się jego nazwa. Każdy obszar zdatnej do przydziału pamięci ma przyległego partnera, czyli sąsiada, a ilekroć dwa sąsiadujące obszary są zwalniane, łączy się je w większy obszar. Taki większy obszar też ma swojego sąsiada, z którym może utworzyć jeszcze większy obszar itd. Z kolei gdy małe zamówienie na pamięć nie może być zrealizowane przez przydział istniejącego, małego obszaru wolnego, wówczas w celu zrealizowania zamówienia większy wolny obszar zostanie podzielony na dwa sąsiednie. Do przechowywania informacji o wolnych obszarach pamięci każdego z dopuszczalnych rozmiarów stosuje się osobne listy z dowiązaniami. W systemie Linux najmniejszą możliwą do przydzielenia w ten sposób jednostką jest pojedyncza strona fizyczna. Na rysunku 22.3 widać przykład przydziału za pomocą sąsiednich stert. Przydziela się obszar wielkości 4 KB, jednak najmniejszy z dostępnych obszarów ma 16 KB. Obszar podlega więc rekurencyjnemu podziałowi, aż uzyska się fragment pożąданiej wielkości.

Ostatecznie wszystkie przydziały pamięci w jądrze systemu Linux są zarezerwowane statycznie przez moduły sterujące w postaci ciągłych obszarów pamięci, przydzielanych w czasie rozruchu systemu lub dynamicznie – za pomocą dyspozytora stron. Jednakże funkcje jądra nie muszą koniecznie używać

* Termin angielski akcentuje przydział stron, lecz zaniedbuje operację ich zwalniania przez opisaną procedurę systemową – Przyp. tłum.

** Czyli ramek (ang. *frames*). – Przyp. tłum.



Rys. 22.3 Podział pamięci w algorytmie sąsiednich ster

podstawowego dyspozytora do rezerwowania pamięci. Istnieje kilka specjalizowanych podsystemów zarządzania pamięcią. Korzystają one z usług dyspozytora stron do realizacji własnej puli pamięci. Do najważniejszych podsystemów pamięci należy system pamięci wirtualnej opisany w p. 22.6.2, dyspozytor obszarów zmiennej długości, czyli funkcja `kmalloc`, oraz dwie trwałe pamięci podrzeczne jądra: podrzczna pamięć buforów i podrzczna pamięć stron.

Wiele części systemu operacyjnego Linux zamawia przydziały całych stron, lecz często są też potrzebne mniejsze bloki pamięci. Jądro dostarcza dodatkowego dyspozytora do realizowania zamówień dowolnej wielkości w sytuacjach, gdy wielkość zamówienia nie jest znana z góry i może wynosić zaledwie kilka bajtów zamiast całej strony. Podobnie jak funkcja `malloc` języka C, usługa `kmalloc` przydziela na żądanie całe strony, a następnie dzieli je na mniejsze kawałki. Jądro utrzymuje zbiór wykazów stron używanych przez usługę `kmalloc`, przy czym wszystkie strony na danym wykazie są podzielone na kawałki określonego rozmiaru. Podczas przydzielania pamięci aktualnia się stan odpowiedniego wykazu, zabierając z niego pierwszy dostępny kawałek strony lub przydzielając nową stronę i ją dzieląc.

Zarówno dyspozytor stron, jak i usługa `kmalloc` są zabezpieczone przed przerwaniami. Funkcja potrzebująca przydziału pamięci przekazuje priorytet zamówienia funkcji przydzielającej. Procedury przerwań stosują niepodzielny priorytet, który zapewnia, że zamówienie zostanie zrealizowane albo – przy braku wystarczającej ilości pamięci – natychmiast zostanie odrzucone. W porównaniu z tym zwykle procesy użytkowe, zamawiające przydział pamięci, rozpoczynają od próby znalezienia pamięci do zwolnienia, co może powodować ich unieruchomianie do czasu uzyskania pamięci. Priorytetu przydziału można również użyć do określenia, że pamięć jest potrzebna dla operacji

DMA, do użytku w architekturach takich jak PC, w których pewne zamówienia DMA nie są realizowane na wszystkich stronach pamięci fizycznej.

Obszary pamięci administrowane przez system `kmalloc` są przydzielane na stałe aż do ich jawnego zwolnienia. W przypadkach braku pamięci system `kmalloc` nie może obszarów tych przemieszczać ani odzyskiwać.

Trzy pozostałe główne podsystemy sprawujące własny zarząd nad stronami fizycznymi są ze sobą blisko spokrewnione. Są to: podręczna pamięć buforów, podręczna pamięć stron oraz system pamięci wirtualnej. Podręczna pamięć buforów jest główną pamięcią podręczną jądra stosowaną w kontakcie z urządzeniami blokowymi, takimi jak napędy dysków i stanowi podstawowy mechanizm za pomocą którego są wykonywane operacje wejścia-wyjścia tych urządzeń. Podręczna pamięć stron przechowuje całe strony z zawartościami plików i nie ogranicza się do urządzeń blokowych. Może ona również przechowywać dane z sieci i jest używana zarówno przez rdzenne, dyskowe systemy plików Linux, jak i przez sieciowy system plików NFS. System pamięci wirtualnej zarządza zawartością przestrzeni adresowej każdego procesu.

Wszystkie te trzy systemy blisko ze sobą współpracują. Przeczytanie strony danych do podręcznej pamięci stron wymaga chwilowego przejścia przez podręczną pamięć buforów. Strony w podręcznej pamięci stron mogą być także odwzorowywane w systemie pamięci wirtualnej, jeżeli proces odwzorował plik w swojej przestrzeni adresowej. Jądro utrzymuje licznik odwołań do każdej ze stron pamięci fizycznej, toteż strony dzielone przez dwa lub więcej z tych podsystemów mogą być zwalniane dopiero wtedy, kiedy nie są już potrzebne w żadnym z nich.

22.6.2 Pamięć wirtualna

System pamięci wirtualnej Linux odpowiada za opiekę nad przestrzenią adresową widoczną dla każdego z procesów. Tworzy on strony pamięci wirtualnej na żądanie i zarządza sprowadzaniem ich z dysku lub ich wyniesieniem z powrotem na dysk w razie potrzeby. W systemie Linux zarządcą pamięci wirtualnej utrzymuje dwa osobne obrazy przestrzeni adresowej procesu: zbiór oddzielnich obszarów i zbiór stron.

Pierwszy obraz przestrzeni adresowej jest obrazem logicznym, odzwierciedlającym rozkazy, za pomocą których system pamięci wirtualnej ukształtował wygląd przestrzeni adresowej. Na tym obrazie przestrzeń adresowa składa się ze zbioru nie zachodzących na siebie obszarów reprezentujących ciągle podzbiory przyległych stron przestrzeni adresowej. Każdy obszar jest opisany wewnętrznie za pomocą jednej struktury `vm_area_struct`, która określa jego cechy, łącznie z procesowymi prawami czytania, pisania i wykonywania obszaru oraz informacjami dotyczącymi wszystkich plików skojarzonych z nim.

rzonych z obszarem. Obszary każdej przestrzeni adresowej są połączone w zrównoważone drzewo binarne w celu umożliwienia szybkiego przeszukiwania obszaru odpowiadającego dowolnemu adresowi wirtualnemu.

Jądro utrzymuje także drugi, fizyczny obraz każdej przestrzeni adresowej. Obraz ten jest pamiętany w sprzętowej tablicy stron procesu. Wpis w tablicy stron określa dokładnie bieżące położenie każdej strony pamięci wirtualnej niezależnie od tego, czy znajduje się ona na dysku, czy w pamięci operacyjnej. Obraz fizyczny jest zarządzany przez zbiór procedur wywoływanych przez procedury obsługi programowych przerwań jądra, ilekroć proces próbuje uzyskać dostęp do strony, której nie ma w danej chwili w tablicy stron. Każda struktura `vm_area_struct` w opisie przestrzeni adresowej zawiera pole wskazujące tablicę funkcji realizujących podstawowe działania administracyjne na stronach dla dowolnego, określonego obszaru pamięci wirtualnej. Wszystkie zamówienia na czytanie lub pisanie niedostępnej strony są w końcu kierowane do odpowiedniej procedury z tablicy `w vm_area_struct`, zatem procedury centralnego zarządzania pamięcią nie muszą znać szczegółów zarządzania wszelkimi możliwymi typami obszarów pamięci.

22.6.2.1 Obszary pamięci wirtualnej

System Linux realizuje różne typy obszarów pamięci wirtualnej. Pierwszym elementem charakteryzującym typ pamięci wirtualnej jest pamięć pomocnicza obszaru. Pamięć ta określa, skąd pochodzą strony obszaru. Większość obszarów ma jako swoje zaplecze pliki lub nie ma żadnego zaplecza. Obszar nie mający żadnego zaplecza jest najprostszym typem pamięci wirtualnej. Obszar taki reprezentuje pamięć o „zerowych wymaganiach” (ang. *demund-zero memory*). Jeśli proces próbuje czytać stronę w takim obszarze, to dostarcza mu się po prostu stronę pamięci wypełnioną zerami.

Obszar mający jako zaplecze plik działa jak odniesienie do części tego pliku. Gdy tylko proces próbuje dostępu do strony w takim obszarze, w tablicy stron umieszcza się adres strony z podręcznej pamięci stron jądra, odpowiadającej właściwej odległości w pliku. Ta sama strona pamięci fizycznej jest używana zarówno przez podręczną pamięć stron, jak i przez tablice stron procesu, więc wszystkie zmiany wykonane w pliku przez system są natychmiast widoczne w każdym procesie mającym odwzorowany ów plik w swojej przestrzeni adresowej. Ten sam obszar jednego pliku może być odwzorowany przez dowolnie wiele procesów, więc w rezultacie będą one korzystać w tym celu z tej samej strony pamięci fizycznej.

Obszar pamięci wirtualnej jest również zdefiniowany przez sposób jego reagowania na pisanie. Odwzorowanie obszaru w przestrzeni adresowej procesu może być *prywatne* lub *wspólne*. Jeśli proces zapisuje obszar odwzorowany prywatnie, to procedura stronicująca wykrywa, że w celu zachowania

lokalności zmian w procesie niezbędne będzie kopiowanie przy zapisie. Z drugiej strony, zapisywanie obszaru wspólnego powoduje uaktualnienie obiektu odwzorowanego w tym obszarze, więc zmiana będzie widoczna natychmiast we wszystkich procesach, które mają ten obiekt odwzorowany.

22.6.2.2 Czas istnienia wirtualnej przestrzeni adresowej

Jądro utworzy nową wirtualną przestrzeń adresową w dokładnie dwu sytuacjach: gdy proces rozpoczyna wykonanie nowego programu za pomocą funkcji systemowej `exec` oraz przy tworzeniu nowego procesu za pomocą funkcji systemowej `fork`. Pierwszy przypadek jest łatwy. Kiedy następuje wykonanie nowego programu, wtedy proces otrzymuje nową, całkowicie pustą wirtualną przestrzeń adresową. Zapelnienie tej przestrzeni adresowej obszarami pamięci wirtualnej należy do procedur ładujących program.

W drugim przypadku tworzenie nowego procesu za pomocą funkcji `fork` obejmuje utworzenie pełnej kopii wirtualnej przestrzeni adresowej istniejącego programu. Jądro kopiuje deskryptory `vm_area_struct` procesu macierzystego, a następnie tworzy nowy zbiór tablic stron dla procesu potomnego. Tablice stron procesu macierzystego są kopowane wprost do tablic procesu potomnego ze zwiększeniem licznika odwołań każdej strony. Tak więc po operacji `fork` procesy macierzysty i potomny dzielą w swoich przestrzeniach adresowych te same fizyczne strony pamięci.

Specjalny przypadek występuje wówczas, gdy operacja kopiowania natyka się na obszar pamięci wirtualnej, który jest odwzorowany prywatnie. Wszelkie strony, na których proces macierzysty pisał w takim obszarze, są prywatne, toteż następne zmiany na tych stronach, wykonywane przez proces macierzysty lub potomny, nie powinny uaktualniać stron w przestrzeni adresowej drugiego z nich. Przy kopiowaniu wpisów tablicy stron takiego obszaru określa się je jako zdatne tylko do czytania i oznacza jako kopowane przy zapisie. Dopóki strony te nie zostaną zmienione przez żaden z procesów, dopóty oba procesy dzielą tę samą stronę pamięci fizycznej. Jednak gdy któryś proces spróbuje zmienić stronę kopowaną przy zapisie, wówczas sprawdza się licznik odwołań do tej strony. Jeśli strona wciąż podlega dzieleniu, to proces kopiuje jej zawartość na całkini nową stronę pamięci fizycznej i używa jej kopii zamiast oryginału. Mechanizm taki zapewnia, że strony z danymi prywatnymi będą przez procesy użytkowane wspólnie, gdzie tylko jest to możliwe, a kopie będą wykonywane tylko w razie bezwzględnej konieczności.

22.6.2.3 Wymiana i stronicowanie

Ważnym zadaniem dla systemu pamięci wirtualnej jest przemieszczanie stron pamięci z pamięci operacyjnej na dysk, kiedy występuje zapotrzebowanie na tę pierwszą. Wczesne systemy uniksowe dokonywały takich przemieszczeń

przez jednorazową wymianę zawartości całych procesów, lecz nowoczesne systemy UNIX w większym stopniu korzystają ze stronicowania, przemieszczając między pamięcią operacyjną a dyskiem poszczególne strony pamięci wirtualnej. Linux nie implementuje wymiany całego procesu – korzysta wyłącznie z nowszego mechanizmu stronicowania.

System stronicowania można podzielić na dwie części. Po pierwsze, istnieje algorytm postępowania (ang. *policy algorithm*) decydujący o tym, które strony zapisywać na dysku i kiedy. Po drugie, istnieje mechanizm stronicowania (ang. *paging mechanism*), który realizuje przesłania i który sprowadza strony danych z powrotem do pamięci fizycznej, kiedy staną się znów potrzebne.

Zastosowana w Linuxie polityka wysyłania stron na dysk jest zmodyfikowaną wersją standardowego algorytmu zegarowego (drugiej szansy), opisanego w p. 9.5.4.2. W systemie Linux używa się wieloprzebiegowego zegara, przy czym każda strona ma swój *wiek*, korygowany w każdym przebiegu zegara. Ów wiek jest, mówiąc scisłej, miarą żywotności strony, tj. stopnia jej aktywności w ostatnim czasie. Strony często odwiedzane otrzymują większą wartość wieku, natomiast wiek rzadko odwiedzanych stron będzie w każdym przebiegu dążył do zera. Takie oszacowanie wieku pozwala procedurze stronicującej wybierać do wyrzucenia strony według kryterium najrzadszego ich używania (LFU).

Mechanizm stronicowania umożliwia stronicowanie zarówno z użyciem urządzeń i stref wymiany, jak i zwykłych plików, choć wymiana z użyciem pliku jest znacznie wolniejsza z powodu dodatkowych kosztów narzuconych przez system plików. Przydział bloków na urządzeniach wymiany odbywa się według mapy bitowej używanych bloków, stale przechowywanej w pamięci operacyjnej. Aby dążyć do zapisywania stron w ciągły sposób w sąsiednich blokach dysku, co poprawia wydajność, dyspozytor stron stosuje algorytm najbliższego dopasowania (ang. *next fit*). Fakt wysyłania strony na dysk dyspozytor odnotowuje, wykorzystując właściwość tablic stron nowoczesnych procesorów: we wpisie tablicy stron ustawia się bit nieebecnej strony, co pozwala na zapelnienie reszty wpisu tablicy stron za pomocą indeksu identyfikującego miejsce, w którym ją zapisano.

22.6.2.4 Wirtualna pamięć jądra

Stał, zależny od architektury obszar wirtualnej przestrzeni adresowej każdego procesu zostaje zarezerwowany przez system Linux do jego własnego użytku. Wpisy tablicy stron wskazujące na strony jądra oznacza się jako chronione, wskutek czego strony takie nie są widoczne ani modyfikowalne, gdy procesor działa w trybie użytkownika. Owa strefa wirtualnej pamięci jądra zawiera dwa obszary. Pierwsza część jest obszarem statycznym, zawierającym

cym odwołania tablicy stron do każdej dostępnej, fizycznej strony pamięci w systemie, co upraszcza tłumaczenie adresów fizycznych na wirtualne w czasie wykonywania kodu jądra. W tym obszarze przebywa rdzeń jądra oraz wszystkie strony przydzielone przez zwykłą procedurę stronicującą.

Reszta obszaru przestrzeni adresowej zarezerwowanego dla jądra nie jest zarezerwowana w żadnym konkretnym celu. Wpisy tablicy stron odnoszące się do tego przedziału adresów mogą być zmieniane przez jądro, aby stosownie do potrzeb wskazywać dowolne inne fragmenty pamięci. Procesy mogą korzystać z tej pamięci wirtualnej za pomocą pary funkcji zawartych w jądrze. Funkcja `vmalloc` przydziela dowolną liczbę stron pamięci i odwzorowuje je w jeden obszar wirtualnej pamięci jądra, pozwalając na przydział wielkich, ciągłych porcji pamięci nawet wtedy, kiedy do zrealizowania tego zamówienia nie ma wystarczającej liczby przylegających do siebie stron fizycznych. Funkcja `vremap` odwzorowuje ciąg adresów wirtualnych w celu wskazania strefy pamięci używanej przez moduł sterujący urządzeniem do odwzorowywanych w pamięci operacji wejścia-wyjścia.

22.6.3 Wykonywanie i ładowanie programów użytkownika

Wykonanie programów użytkownika w systemie Linux następuje przez wywołanie funkcji systemowej `exec`. Nakazuje ona jądro wykonać w ramach bieżącego procesu nowy program, zapisując całkowicie na nowo bieżący kontekst wykonywania przez początkowy kontekst nowego programu. Pierwszą rzeczą, jaką musi wykonać ta usługa systemowa, jest sprawdzenie, że proces wywołujący ma prawo do wykonywania danego pliku. Po tej weryfikacji jądro wywoła procedurę ładującą w celu rozpoczęcia wykonywania programu. Program ładujący niekoniecznie musi wprowadzać zawartość pliku z programem do pamięci fizycznej, lecz przynajmniej ustala odwzorowanie programu w pamięci wirtualnej.

W systemie Linux nie ma pojedynczej procedury ładowania nowego programu. Zamiast tego Linux operuje tablicą możliwych funkcji ładujących, dając każdej z nich szansę spróbowania umieszczenia w pamięci danego pliku podczas wykonywania systemowej funkcji `exec`. Początkową przyczyną zastosowania owej tablicy funkcji ładujących była zmiana standardowego formatu linuksowych plików binarnych między jądrami o wydaniach 1.0 i 1.2. W starszych jądrach systemu Linux rozpoznawano format a.out plików binarnych – stosunkowo prosty format, popularny w starszych systemach UNIX. Nowsze systemy Linux stosują nowocześniejszy format ELF, dostarczany obecnie przez większość bieżących implementacji systemu UNIX. Binarna postać standardu ELF może być bez przeszkód dla działania procedur ładujących uzupełniana o nowe sekcje (np. w celu dodania specjalnej informacji

uruchomieniowej). Zezwalając na zarejestrowanie wielu procedur ładowających, Linux może łatwo dostarczać w ramach jednego systemu zarówno format ELF, jak i a.out.

W punktach 22.6.3.1 i 22.6.3.2 koncentrujemy się wyłącznie na ładowaniu i wykonywaniu plików binarnych formatu ELF. Procedura ładowania plików binarnych a.out jest prostsza, lecz podobna w działaniu.

22.6.3.1 Odwzorowywanie programów w pamięci

Wprowadzanie pliku binarnego do pamięci fizycznej nie jest w systemie Linux wykonywane przez program ładowający obrazów binarnych (ang. *binary loader*). Zamiast tego następuje odwzorowywanie stron pliku binarnego w obszary pamięci wirtualnej. Załadowanie strony do pamięci fizycznej jest powodowane dopiero po nieudanej próbie odwołania do niej przez program.

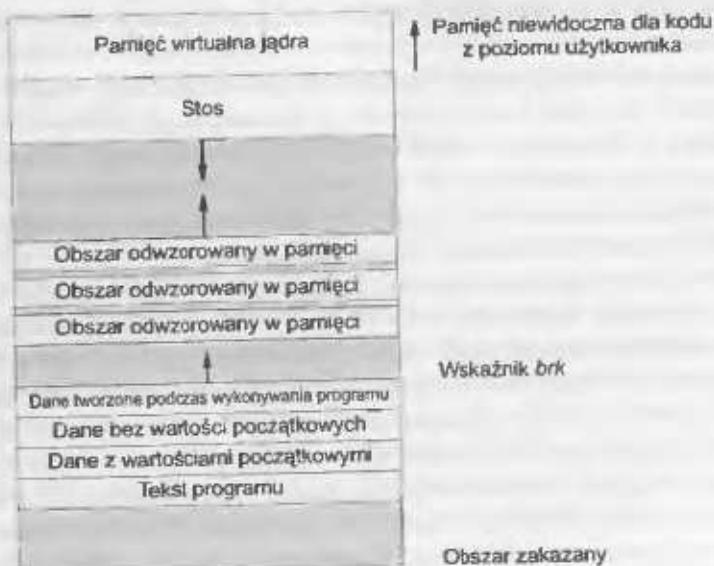
Do obowiązków jądrowych procedur ładowających należy ustalenie początkowego odwzorowania pamięci. Plik binarny formatu ELF składa się z nagłówka, po którym występuje kilka sekcji rozpoczętych się i kończących się na granicach stron. Praca programu ładowającego formatu ELF polega na czytaniu nagłówka i odwzorowywaniu sekcji pliku na osobne obszary pamięci wirtualnej.

Na rysunku 22.4 widać typowy wygląd obszarów pamięci określonych przez program ładowający ELF. W obszarze zarezerwowanym na jednym z końców przestrzeni adresowej jest umieszczone jądro; jego uprzywilejowany obszar pamięci wirtualnej jest niedostępny dla zwykłych programów z poziomu użytkownika. Reszta pamięci wirtualnej jest dostępna dla aplikacji, które mogą używać jądrowych funkcji odwzorowywania pamięci do tworzenia obszarów z odwzorowaniami fragmentów pliku lub w celu pomieszczenia danych aplikacji.

Zadaniem programu ładowającego jest ustalenie początkowego odwzorowania pamięci w celu umożliwienia rozpoczęcia wykonywania programu. Do obszarów, które wymagają określenia wartości początkowych, należą: stos i tekst programu oraz obszar danych.

Stos powstaje u szczytu pamięci wirtualnej poziomu użytkownika. Rośnie on w dół, w kierunku adresów o mniejszych numerach i zawiera kopie argumentów oraz zmienne środowiskowe podane programowi w funkcji systemowej exec. Inne obszary są tworzone bliżej dolnego końca pamięci wirtualnej. Sekcje pliku binarnego zawierające tekst programu lub dane przeznaczone tylko do czytania są odwzorowywane w pamięci jako obszary chronione przez zapisem. Dalej są odwzorowywane zapisywane dane o nadanych wartościach początkowych. Na końcu, w prywatnym obszarze o zerowych

* Czyli kod binarny programu – Przyp. tłum.



Rys. 22.4 Wygląd pamięci w programach formatu ELF

wymaganiach odwzorowuje się wszelkie dane, których wartości początkowe nie zostały określone.

Bezpośrednio poza tymi stałymi rozmiarami obszarami znajduje się obszar o zmiennej długości, który stosownie do potrzeb może być przez programy rozszerzany w celu utrzymywania danych przydzielanych podczas wykonywania. Każdy proces ma wskaźnik o nazwie `brk`, który wskazuje aktualną granicę tego obszaru. Obszar wskazywany przez `brk` może być przez proces wydłużany lub skracany za pomocą pojedynczego wywołania systemowego.

Po ustaleniu odwzorowania program ładujący określa w procesie wstępny stan rejestru programowego licznika rozkazów za pomocą wartości początkowej zapamiętanej w nagłówku ELF i proces może kandydować do procesora.

22.6.3.2 Łączenie statyczne i dynamiczne

Z chwilą załadowania i rozpoczęcia wykonywania programu cała niezbędna zawartość pliku binarnego znajduje się już w wirtualnej przestrzeni adresowej procesu. Jednakże większość programów chce również wykonywać funkcje z bibliotek systemowych, więc funkcje te także muszą być załadowane. W najprostszym przypadku funkcje takie zostają osadzone wprost w binarnym pliku wykonywalnym podczas budowania przez programistę aplikacji. Program taki zostaje *statycznie* połączony ze swoimi bibliotekami, a statycz-

nie połączone (skonsolidowane) pliki wykonywalne mogą podjąć działanie natychmiast po załadowaniu.

Podstawową wadą łączenia statycznego jest to, że każdy wygenerowany program musi zawierać kopię dokładnie tych samych, popularnych systemowych funkcji bibliotecznych. Pod względem ekonomicznego wykorzystania zarówno pamięci operacyjnej, jak i dyskowej byłoby znacznie wydajniej ładować biblioteki systemowe do pamięci w tylko jednym egzemplarzu. Postępowanie takie umożliwia łączenie dynamiczne.

System Linux realizuje łączenie dynamiczne w trybie użytkownika za pomocą specjalnej biblioteki konsolidatora. Każdy program łączony dynamicznie zawiera małą, statycznie dodawaną funkcję, wywoływaną na początku programu. Owa statyczna funkcja odwzorowuje po prostu bibliotekę konsolidacji (łączenia modułów programu) w pamięci i powoduje wykonanie jej kodu. Biblioteka konsolidacji czyta wykaz bibliotek dynamicznych wymaganych przez program oraz zawartych w nich, a potrzebnych mu zmiennych i funkcji, analizując informacje zawarte w częściach formatu binarnego ELF. Odwzorowuje ona potem te biblioteki pośrodku pamięci wirtualnej, kierując odwołania programowe do zawartych w nich symboli. Miejsce odwzorowania tych wspólnych bibliotek w pamięci nie ma specjalnego znaczenia – są one kompilowane w kodzie niezależnym od położenia (ang. *position-independent code* – PIC), który można wykonywać w pamięci pod każdym adresem.

22.7 ■ Systemy plików

Linux zachowuje model systemu plików standardu UNIX. W systemie UNIX plik nie musi być obiektem przechowywanym na dysku lub ładowanym za pomocą sieci z odległego serwera plików. Plik systemu UNIX może byćвшystkim, co jest w stanie obsługiwać strumień danych wejściowych lub wyjściowych. Moduły sterujące urządzeń mogą sprawiać wrażenie plików. Z punktu widzenia użytkownika do plików są również podobne kanały komunikacji międzyprocesowej lub połączenia sieciowe.

Jądro systemu Linux obsługuje wszystkie te różnorodne typy plików, ukrywając szczegóły ich implementacji za warstwą programową, czyli wirtualnym systemem plików (VFS).

22.7.1 Wirtualny system plików

Wirtualny system plików Linuksa (ang. *virtual file system* – VFS) zaprojektowano na zasadach obiektowych. Ma on dwie składowe: zbiór definicji określających, jak powinien wyglądać obiekt o nazwie plik, oraz warstwę opro-

gramowania do działań na takich obiektach. W systemie VFS zdefiniowano trzy podstawowe typy obiektów: struktury *obiektu i-węzła* oraz *obiektu pliku*, reprezentujące poszczególne pliki, oraz *obiekt systemu plików*, reprezentujący cały system plików.

Dla każdego z tych trzech typów obiektów system VFS definiuje zbiór operacji, które muszą być zrealizowane dla ich struktur. Każdy z obiektów wymienionych typów zawiera wskaźnik do tablicy funkcji. Tablica funkcji jest wykazem adresów rzeczywistych funkcji implementujących działania na konkretnym obiekcie. W ten sposób warstwa oprogramowania VFS może wykonywać operację na jednym z obiektów, wywołując odpowiednią funkcję z tablicy funkcji obiektu bez konieczności uprzedniego dokładnego poznawania, z jakim rodzajem obiektu ma do czynienia. System VFS nie wie, bądź nie dba o to, czy obiekt jest plikiem zapamiętanym w sieci, czy na dysku, gniazdem sieciowym czy katalogiem – odpowiednia dla niego operacja „czytaj dane” będzie zawsze w tym samym miejscu w jego tablicy funkcji, a warstwa oprogramowania VFS wywoła tę funkcję, nie wnikając w rzeczywisty sposób czytania danych.

Obiekt systemu plików reprezentuje połączony zbiór plików, który tworzy zamkniętą w sobie hierarchię katalogów. Jądro systemu operacyjnego utrzymuje dla każdego zamontowanego urządzenia dyskowego jeden reprezentujący system plików obiekt systemu plików, jak również czyni tak dla każdego systemu plików aktualnie podłączonego przez sieć. Podstawowym obowiązkiem obiektu systemu plików jest udostępnianie i-węzłów. System VFS identyfikuje każdy i-węzeł za pomocą niepowtarzalnej pary (system plików, numer i-węzła) i odnajduje i-węzeł odpowiadający danemu numerowi i-węzła, za pomocą skierowanej do obiektu systemu plików prośby o zwrócenie i-węzła o podanym numerze.

Obiekty i-węzłów oraz plików są mechanizmami dostępu do plików. Obiekt i-węzła reprezentuje plik jako całość, a obiekt pliku reprezentuje punkt dostępu do danych wewnętrz pliku. Proces nie może sięgnąć do zawartości danych i-węzła bez wcześniejszego uzyskania obiektu pliku wskazującego na i-węzeł. Obiekt pliku przechowuje informacje o aktualnym miejscu czytania lub zapisywania pliku przez proces, co umożliwia sekwencyjne operacje wejścia-wyjścia na pliku. W obiekcie tym pamięta się również, czy proces prosił o prawo zapisywania pliku przy jego otwieraniu, jak również przechowuje informacje o czynnościach procesu w razie konieczności wykonywania czytania z wyprzedzeniem (tj. sprowadzania danych pliku do pamięci zaraz po zapotrzebowaniem ich przez proces, w celu poprawienia wydajności).

Obiekty plików zazwyczaj należą do pojedynczych procesów, nie jest tak natomiast w przypadku obiektów i-węzłów. Nawet jeżeli plik przestaje być używany przez jakikolwiek proces, system VFS może wciąż przechowywać

jego i-węzle w pamięci podręcznej w celu zwiększenia wydajności w przypadku ponownego użycia pliku w najbliższej przyszłości. Wszystkie przechowywane podręcznie dane plikowe są łączone w listę w obiekcie i-węzła pliku. I-węzeł utrzymuje też standardowe informacje o pliku, takie jak nazwa właściciela, rozmiar pliku i czas jego ostatniej modyfikacji.

Pliki katalogowe są obsługiwane nieco inaczej niż pozostałe pliki. Programowy interfejs systemu UNIX określa pewną liczbę operacji na katalogach, takich jak tworzenie, usuwanie i przemianowywanie pliku w katalogu. W odróżnieniu od czytania i zapisywania danych, kiedy ten plik musi najpierw być otworzony, systemowe wywołania operacji na katalogach nie wymagają od użytkownika otwierania interesujących go plików. Dlatego system VFS definiuje operacje katalogowe w obiekcie i-węzła, a nie w obiekcie pliku.

22.7.2 System plików Linux Ext2fs

Standardowy, dyskowy system plików stosowany w systemie Linux swoją nazwę – *ext2fs* – nosi z przyczyn historycznych. Pierwotnie Linux był zaprogramowany z systemem plików zgodnym z systemem Minix, aby ułatwić wymianę danych z ćwicznym systemem Minix, jednak ten system plików był poważnie ograniczony przez 14-znakowe nazwy plików i maksymalny rozmiar wynoszący 64 MB. System plików Minix zastąpiono nowym systemem, który oznaczono mianem rozszerzonego systemu plików (ang. *extended file system – extfs*). Jego późniejsze przeprojektowanie mające na celu polepszenie wydajności i skalowalności oraz dodanie kilku brakujących cech zaowocowało nazwą *ext2fs* (ang. *second extended file system*).

System ext2fs ma wiele wspólnego z szybkim systemem plików BSD (ang. *Fast File System – ffs*) opisany w p. 21.7.7. Stosuje on podobny mechanizm odnajdywania bloków danych należących do konkretnego pliku, przechowując wskaźniki bloków danych w blokach pośrednich, spiętrzanych w całym systemie aż do trzeciego poziomu. Analogicznie do systemu ffs pliki katalogowe są pamiętane na dysku tak jak zwykłe pliki, chociaż ich zawartość jest interpretowana odmiennie. Każdy blok w pliku katalogowym składa się z powiązanej listy wpisów, z których każdy zawiera długość wpisu, nazwę pliku i numer i-węzła, do którego dany wpis się odwołuje.

Główna różnica między systemami ext2fs i ffs dotyczy sposobu przydziału dysku. W systemie ffs dysk jest przydzielany plikom blokami wielkości 8 KB, z dzieleniem bloków na porcje 1 KB w celu pamiętania małych plików lub niepełnych bloków na końcach plików. W porównaniu z tym system ext2fs nie używa w ogóle bloków cząstkowych, natomiast dokonuje wszystkich przydziałów za pomocą mniejszych jednostek. Blok standardowy w systemie ext2fs ma wielkość 1 KB, choć stosuje się także bloki o wielkości 2 KB i 4 KB.

Aby utrzymywać wysoką wydajność, system operacyjny powinien starać się wykonywać operacje wejścia-wyjścia wielkimi porcjami, gdy tylko jest to możliwe, łącząc zamówienia na fizycznie przyległe jednostki wejścia-wyjścia w grona. Operowanie gronami zmniejsza przypadające na jedno zamówienie koszty wynikające z pracy modułów sterujących urządzeń, dysków i sprzętu nadzorującego działanie dysku. Zamówienie wielkości 1 KB jest za małe, aby utrzymywać dobrą wydajność, toteż system ext2fs stosuje politykę przydziału zmierzającą do umieszczania logicznie sąsiadujących bloków pliku w fizycznie przylegających blokach dyskowych, dzięki czemu może on w jednej operacji przedkładać zamówienia wejścia-wyjścia dla kilku bloków.

Zasady dokonywania przydziałów w systemie ext2fs można podzielić na dwie części. Tak jak system ffs, system plików ext2fs jest podzielony na wiele *grup bloków*. W systemie ffs zastosowano podobną koncepcję *grupy cylindrów*, gdzie każda grupa odpowiada jednemu cylindrowi dysku fizycznego. Jednak napęd dysku o nowoczesnej technologii pakuje sektory na dysku z różnymi gęstościami, co powoduje różne rozmiary cylindrów w zależności od odległości głowicy dysku od jego środka, dlatego grupy cylindrów o stałych rozmiarach niekoniecznie pasują do geometrii dysku.

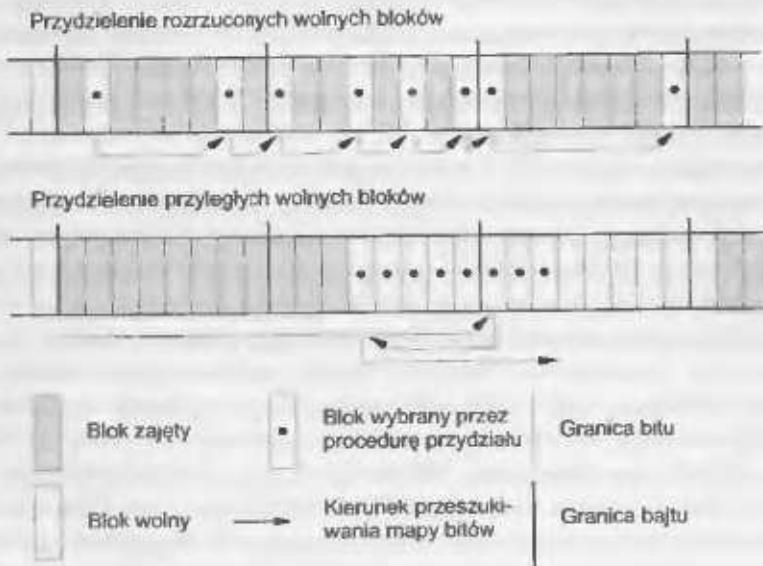
Przydzielając miejsce dla pliku, system ext2fs musi najpierw wybrać dla niego grupę bloków. System usuwa przydzielić blokom danych tę grupę bloków, do której przydzielono i-węzeł pliku. Do przydzielania i-węzłów plików nie będących katalogami system wybiera tę grupę bloków, w której mieści się macierzysty katalog pliku. Pliki katalogowe nie są przechowywane razem, lecz są rozproszone w dostępnych grupach bloków. Te zasady przyjęto po to, aby powiązane ze sobą informacje trzymać w tej samej grupie bloków, a także by rozłożyć obciążenie dysku między jego grupy bloków w celu zmniejszenia fragmentacji obszarów dyskowych.

Wewnętrz grupy bloków system ext2fs próbuje w miarę możliwości dokonywać przydziałów ciągłych fizycznie, dając do zmniejszania fragmentacji. Utrzymuje on mapę bitową wszystkich wolnych bloków w grupie. Podczas przydzielania pierwszego bloku nowego pliku system rozpoczyna szukanie wolnego bloku od początku grupy bloków; przy rozszerzaniu pliku poszukiwanie są kontynuowane od bloku przydzielonego plikowi ostatnio. Szukanie odbywa się dwuetapowo. Najpierw system szuka całego wolnego bajta w mapie bitowej. Jeżeli to się nie powiedzie, to następuje szukanie pojedynczego wolnego bitu. Szukanie wolnych bajtów ma na celu przydzielanie dysku porcjami co najmniej ośmioblokowymi, gdy tylko jest to możliwe.

Po zlokalizowaniu wolnego bloku poszukiwanie jest kontynuowane wstecz aż do napotkania bloku przydzielonego. W przypadku znalezienia w mapie bitowej wolnego bajta to rozszerzanie wstecz zapobiega przed pozostawieniem przez system ext2fs dziury między ostatnim przydzielonym blo-

kiem z poprzedniego, niezerowego bajta a blokami z odnalezionej bajta zerowego. Gdy następny blok do przydzielenia zostanie za pomocą takiego bitowego lub bajtowego szukania odnaleziony, system ext2fs rozszerza przydział w przód co najwyżej do ośmiu bloków i wstępnie przydziela te dodatkowe bloki do pliku. Wstępny przydział pozwala zmniejszać fragmentację powstającą podczas naprzemiennej zapisywania różnych plików, redukuje również koszt procesora zużywany na przydział dysku dzięki jednoczesnemu przydzielaniu wielu bloków. Przy zamykaniu pliku wstępnie przydzielone bloki są oznaczane w mapie bitowej jako wolne przestrzenie.

Na rysunku 22.5 są pokazane zasady przydziału. Każdy rządek przedstawia ciąg ustawionych bądź wyzerowanych bitów w mapie bitowej przydziałów, reprezentujących zajęte bądź wolne bloki na dysku. W pierwszym przypadku, jeżeli jesteśmy w stanie znaleźć jakieś wolne bloki wystarczająco blisko miejsca, od którego zaczyna się szukanie, to przydzielamy je, nie zważając na ich rozproszenie. Fragmentacja jest częściowo kompensowana przez wzajemną bliskość bloków, więc prawdopodobnie da się je wszystkie przeczytać bez żadnych przemieszczeń na dysku, a przydzielanie ich wszystkich do jednego pliku jest w dłuższym okresie lepsze niż przydzielanie wyizolowanych bloków do różnych plików, skoro na dysku brakuje już wielkich, wolnych obszarów. W drugim przypadku nie znaleźliśmy od razu wolnego bloku tuż obok, dlatego zaczęliśmy szukać dalej w mapie bitowej całego wolnego bajta.



Rys. 22.5 Zasady przydziału bloków w systemie ext2fs

Gdybyśmy przydzieliли go w całości, to przed nim powstałaby nie wykorzystana wolna przestrzeń, toteż przed dokonaniem przydziału wracamy, aby wy pełnić tę lukę za pomocą przydziału wcześniejszych bloków i dopiero potem postępujemy z przydziałem do przodu w celu osiągnięcia jego zastępczej wielkości ośmiu bloków.

22.7.3 System plików Linux-*proc*

Linuksowy system VFS jest na tyle elastyczny, że pozwala na implementację systemu plików, który zamiast trwale przechowywać dane, po prostu dostarcza interfejsu do pewnych innych działań. *Procesowy system plików Linuxa* (ang. *process file system*), znany jako system plików *proc*, jest przykładem systemu plików, którego dane w rzeczywistości nie są nigdzie pamiętane, natomiast oblicza się je na żądanie, stosownie do wydawanych przez użytkownika zamówień na operacje wejścia-wyjścia.

System plików *proc* nie występuje wyłącznie w systemie Linux. W systemie SVR4 UNIX wprowadzono system plików *proc* jako wydajny interfejs wspierający sprawdzanie poprawności procesów jądra. Każdy z podkatalogów tego systemu odpowiadał nie katalogowi na jakimś dysku, lecz procesowi aktywnemu w aktualnym systemie. Wykaz zawartości takiego systemu plików ujawnia po jednym katalogu na proces, przy czym nazwa katalogu jest dziesiętną reprezentacją ASCII niepowtarzalnego identyfikatora procesu (PID).

Linux implementuje system plików *proc*, rozszerzając go znacznie przez dodanie pewnej liczby specjalnych katalogów i plików tekstowych w głównym katalogu systemu plików. Te nowe wpisy odpowiadają różnym statystykom dotyczącym jądra i skojarzonych z nim, załadowanych modułów sterujących. System plików *proc* umożliwia programom dostęp do tych informacji jako do zwykłych plików tekstowych, podczas gdy standardowe środowisko użytkownika systemu UNIX dostarczało je do procesu za pomocą silnych narzędzi. Na przykład w przeszłości tradycyjne uniksowe polecenie *ps*, służące do wyprowadzania wykazu stanów wszystkich wykonywanych procesów, implementowano jako uprzywilejowany proces czytający stan procesu wprost z pamięci wirtualnej jądra. Pod nadzorem systemu Linux polecenie to jest zrealizowane w całości w postaci programu nieuprzywilejowanego, który po prostu dokonuje rozbioru i formatowania informacji pobranych z systemu *proc*.

System plików *proc* musi realizować dwie rzeczy: strukturę katalogów oraz docieranie do zawartości pliku. Mając za punkt wyjścia to, że system plików UNIX jest zdefiniowany jako zbiór i-węzłów plików i katalogów, identyfikowanych za pomocą numerów i-węzłów, system plików *proc* musi definiować niepowtarzalny i trwały numer i-węzła dla każdego katalogu i skojarzonych z nim plików. Po dokonaniu takiego odwzorowania system mo-

że korzystać z numerów i-węzłów w celu rozpoznawania operacji potrzebnej wówczas, gdy użytkownik próbuje czytać dane z jakiegoś i-węzła pliku lub odnaleźć coś w i-węźle katalogu. Podczas czytania danych z któregoś z tych plików, system plików proc będzie gromadził odpowiednie informacje, nadawał im formę tekstową i umieszczał w buforze zamawiającego je procesu.

Podeczas odwzorowania numeru i-węzła na informacje jakiegoś typu dzieli się go na dwa pola. W systemie Linux identyfikator procesu (PID) ma 16 bitów, natomiast numer i-węzła jest 32-bitowy. Górnego 16 bitów numeru i-węzła interpretuje się jako PID, a pozostałe bity określają zamawiany rodzaj informacji o procesie.

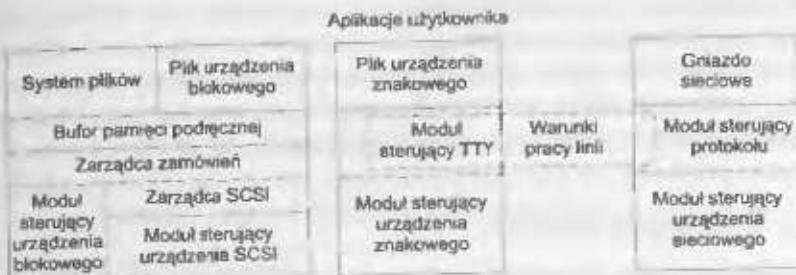
Zerowy identyfikator PID nie jest dozwolony, dlatego przyjęto, że zero-wa wartość PID w numerze i-węzła oznacza, iż dany i-węzeł zawiera informacje globalne, a nie specyficzne dla procesu. W systemie proc istnieją osobne, globalne pliki z informacjami takimi, jak: wersja jądra, ilość wolnej pamięci, statystyka wydajności oraz aktualnie działające moduły sterujące.

Nie wszystkie numery i-węzłów w podanym przedziale są zajęte. Jądro może dynamicznie przydzielać nowe odwzorowania i-węzłów systemu proc, utrzymując mapę bitową przydzielonych numerów i-węzłów. Utrzymuje ono również drzewistą strukturę zarejestrowanych, globalnych wpisów systemu plików proc. Każdy wpis zawiera numer i-węzła pliku, nazwę pliku, prawa dostępu oraz specjalne funkcje używane do tworzenia zawartości pliku. Moduły sterujące urządzeń mogą w każdej chwili rejestrować się i wyrejestrowywać we wpisach tego drzewa, a specjalna część drzewa (zaczynająca się od katalogu /proc/sys) jest zarezerwowana na zmienne jądra. Pliki w tym drzewie są realizowane za pomocą zbioru wspólnych manipulatorów, pozwalających zarówno na czytanie, jak i na zapisywanie zmiennych jądra, więc administrator systemu może łatwo stroić wartości parametrów jądra, wpisując dziesiętnie nowe, potrzebne wartości w kodzie ASCII do odpowiednich plików.

Aby umożliwić wydajny dostęp do tych zmiennych z wnętrza aplikacji, poddrzewo /proc/sys jest osiągalne za pomocą specjalnego wywołania systemowego sysctl, które czyta i zapisuje te same zmienne binarnie zamiast tekstowo, bez kosztów związanych z użyciem systemu plików. Wywołanie sysctl nie jest dodatkowym udogodnieniem – czyta ono po prostu dynamiczne drzewo wpisów systemu proc, aby rozstrzygać, do których zmiennych odwołuje się aplikacja.

22.8 ■ Wejście i wyjście

Z punktu widzenia użytkownika system wejścia-wyjścia jest w systemie Linux bardzo podobny do zrealizowanego w dowolnym systemie UNIX. Tak więc w maksymalnym stopniu nadaje się wszystkim modułom sterującym



Rys. 22.6 Struktura blokowa modułów sterujących urządzeń

urządzeń wygląd zwykły plików. Użytkownik może otwierać kanał dostępu do urządzenia w ten sam sposób, jak otwiera dowolny inny plik – urządzenia mogą wyglądać jak obiekty w systemie plików. Administrator systemu może tworzyć w systemie plików specjalne pliki, które zawierają odwołania do określonego modułu sterującego urządzenia, a użytkownik, otwierając taki plik, może odwoływać się do urządzenia za pomocą operacji czytania i pisania. Administrator może ustanawiać prawa dostępu do każdego z urządzeń za pomocą zwykłego systemu ochrony plików, służącego do określania praw dostępu do plików.

Linux dzieli wszystkie urządzenia na trzy klasy: *urządzenia blokowe*, *urządzenia znakowe* i *urządzenia sieciowe*. Na rysunku 22.6 widać ogólną strukturę systemu modułów sterujących urządzeń. Do blokowych należą wszelkie te urządzenia, które pozwalały na swobodny dostęp do całkowicie niezależnych, o stałym rozmiarze bloków danych, w czym mieszczą się dyski twarde i elastyczne oraz pamięć CD-ROM. Urządzenia blokowe są na ogół używane do przechowywania systemów plików, zezwala się jednak i na bezpośredni do nich dostęp, dzięki czemu programy mogą na nich tworzyć i naprawiać systemy plików. W razie potrzeby bezpośredni dostęp do urządzeń blokowych mogą też mieć aplikacje, na przykład baza danych może przedkładać ponad ogólnego użytku system plików własny, odpowiednio dostosowany sposób rozmieszczenia danych na dysku.

Urządzenia znakowe obejmują wiele innych urządzeń, a głównym tutaj wyjątkiem są urządzenia sieciowe. Urządzenia znakowe nie muszą mieć wszystkich własności regularnych plików. Na przykład urządzenie głośnika pozwala na zapisywanie w nim danych, lecz nie umożliwia odczytywania z niego danych z powrotem. Podobnie – odnajdywaniem określonego miejsca w pliku może rozporządzać urządzenie taśmy magnetycznej, lecz nie miałoby to sensu w przypadku urządzenia wskazującego, takiego jak myszka.

* Systemowe – Przyp. tłum.

Z urządzeniami sieciowymi postępuje się inaczej niż z blokowymi lub znakowymi. Użytkownicy nie mogą bezpośrednio przekazywać danych do urządzeń sieciowych, lecz zamiast tego muszą się komunikować pośrednio, za pomocą otwierania połączenia z podsystemem sieciowym jądra. Interfejsowi do urządzeń sieciowych został poświęcony oddzielny punkt 22.10.

22.8.1 Urządzenia blokowe

Urządzenia blokowe dostarczają podstawowego interfejsu do wszystkich dysków w systemie. Szczególnego znaczenia w przypadku dysków nabiera wydajność, a system urządzeń blokowych powinien dostarczać rozwiązań zapewniających, że dostęp do dysku jest tak szybki, jak tylko jest to możliwe. Osiaga się to przy użyciu dwu składowych systemu: podręcznej pamięci buforów i zarządcy zamówień.

22.8.1.1 Podręczna pamięć buforów

Podręczna pamięć buforów systemu Linux służy dwu głównym celom. Działa ona zarówno jako pula buforów dla operacji wejścia-wyjścia będących w toku, jak i w charakterze pamięci podręcznej wyników zakończonych operacji wejścia-wyjścia. Podręczna pamięć buforów składa się z dwóch części. Po pierwsze, są to same bufory, czyli zbiór stron przyczepianych wprost z puli pamięci operacyjnej jądra, którego rozmiar jest ustalany dynamicznie. Każda strona jest podzielona na pewną liczbę buforów o równych wymiarach. Po drugie, istnieje odpowiedni zbiór deskryptorów buforów, tj. ich nagłówków (**buffer_heads**) – po jednym na każdy bufor w pamięci podręcznej.

Nagłówki buforów zawierają całość utrzymywanych przez jądro informacji o buforach. Podstawową informacją jest tożsamość bufora. Każdy bufor jest identyfikowany przez trójkę: urządzenie blokowe, do którego bufor należy, położenie danych wewnętrz danego urządzenia blokowego oraz rozmiar bufora. Informacje o buforach są również przechowywane na kilku listach. Istnieją osobne listy buforów czystych, zabrudzonych oraz zablokowanych, jak również lista wolnych buforów. Na listę wolnych buforem trafiają wskutek umieszczania ich tam przez system plików (np. przy usuwaniu pliku) lub jako wynik działania funkcji `refill_freelist` (uzupełnij listę wolnych buforów), wywoływanej gdy jądro potrzebuje więcej buforów. Jądro zapenia listę wolnych buforów przez rozszerzanie puli buforów lub przez wtórnego obrót istniejącymi buforami – w zależności od dostępności wystarczającej ilości wolnej pamięci. Na koniec, każdy bufor, którego nie ma na liście do wynajęcia, jest indeksowany za pomocą funkcji haszowania według jego numeru urządzenia i bloku i dodawany do odpowiedniej listy przeszukiwanej przy udziale tej funkcji.

Zarządzanie buforami jądra automatycznie powoduje zapisywanie zabrudzonych buforów z powrotem na dysk. Asystują przy tym dwa drugoplanowe demony. Jeden z nich po prostu budzi się w regularnych odstępach czasu i zamawia zapisywanie na dysku wszystkich danych, które pozostają zabrudzone przez czas dłuższy od określonego. Drugi demon jest wątkiem jądrowym, budzonym zawsze wtedy, kiedy funkcja `refill_freelist` wykryje, że zbyt duża część podręcznej pamięci buforów jest zabrudzona.

22.8.1.2 Zarządcza zamówień

Zarządcza zamówień jest warstwą oprogramowania, która administruje czytaniem buforów z modułu obsługi urządzenia blokowego i zapisywaniem ich zawartości w tym module. System zamówień opiera się na funkcji `ll_rw_block`, która wykonuje niskopoziomowe czytanie i zapisywanie urządzeń blokowych. Funkcja ta pobiera jako parametry wykaz `buffer_heads` deskryptorów buforów oraz znacznik czytania lub pisania i inicjuje operacje wejścia-wyjścia dla wszystkich tych buforów. Nie czeka ona na zakończenie operacji wejścia-wyjścia.

Pozostałe do wykonania zamówienia wejścia-wyjścia są zapisywane w strukturach `request`. Struktura `request` reprezentuje nie zrealizowane na wyjściu lub wejściu zamówienie na ciągły obszar sektorów jednego z urządzeń blokowych. Ponieważ w przesyłaniu może brać udział więcej niż jeden bufor, więc struktura `request` zawiera wskaźnik do pierwszego z nich na powiązanej liście `buffer_heads`, którego używa się do przesyłania.

Dla każdego modułu obsługi urządzenia blokowego utrzymuje się osobną listę zamówień, których realizacje planuje się według algorytmu jednokierunkowej windy (C-SCAN), korzystającego z porządku wstawiania i usuwania zamówień na listach przypisanych do urządzeń. Listy zamówień są porządkowane według wzrastających numerów początkowych sektorów. Zamówienie przyjęte do wykonania przez moduł obsługi urządzenia blokowego nie jest usuwane z listy. Usuwa się je dopiero po zakończeniu operacji wejścia-wyjścia, kiedy to moduł obsługi podejmuje obsługę następnego zamówienia z listy, nawet pomimo to, że przed aktualnie obsługiwany zamówieniem pojawiło się na liście nowe zamówienie.

W miarę napływu nowych zamówień wejścia-wyjścia zarządcza zamówień próbuje łączyć je na listach przypisanych poszczególnym urządzeniom. Przekazanie jednego, dużego zamówienia zamiast wielu mniejszych jest często znacznie wydajniejsze z punktu widzenia kosztów korzystania z urządzeń. Wszystkie nagłówki buforów (struktury `buffer_heads`) w takim zespołolonym zamówieniu ulegają zablokowaniu już w czasie wykonywania początkowego zamówienia wejścia-wyjścia. Podczas gdy zamówienie jest przetwarzane przez moduł sterujący urządzeniem, odblokowuje się po jednym poszczególnie

nagłówki buforów zespolonego zamówienia. Nie ma potrzeby, aby proces czekający na jeden bufor oczekwał na odblokowanie wszystkich pozostałych buforów w zamówieniu. Jest to szczególnie ważny czynnik, gdyż pozwala na sprawne wykonywanie czytania z wyprzedzeniem.

Zauważmy na koniec, że zarządcy zamówień pozwala na całkowite omijanie bufora przez zamówienia wejścia-wyjścia. Niskopoziomowa funkcja `brw_page` stronicowanego wejścia-wyjścia tworzy tymczasowy zbiór nagłówków buforów w celu zaetykietowania zawartości strony pamięci, aby można było przedkładać zamówienia wejścia-wyjścia zarządcy zamówień. Jednakże te czasowo istniejące nagłówki buforów nie są w podręcznej pamięci buforów łączone w listę i z chwilą zakończenia operacji wejścia-wyjścia w ostatnim buforze na stronie następuje odblokowanie całej strony i usunięcie struktur `buffer_heads`.

Mechanizm omijania pamięci podręcznej jest stosowany w jądrze zawsze wtedy, kiedy proces wywołujący niezależnie dokonuje przechowania danych w pamięci podręcznej lub wówczas, gdy wiadomo, że dane nie nadają się do dalszego podręcznego przechowywania. Pamięć podręczna stron zapewnia swoje strony w ten sposób, aby uniknąć niepożądanej sytuacji, w której te same dane są przechowywane w podręcznej pamięci stron oraz w podręcznej pamięci buforów. System pamięci wirtualnej również obchodzi pamięć podręczną podczas wykonywania operacji wejścia-wyjścia w celu wymiany informacji na urządzeniach.

22.8.2 Urządzenia znakowe

Modułem sterującym urządzenia znakowego może być moduł obsługi niemal dowolnego urządzenia, które nie oferuje swobodnego dostępu do bloków danych o stałej wielkości. Wszelkie moduły sterujące urządzeń znakowych zarejestrowane w jądrze Linux muszą też rejestrować zbiór funkcji realizujących różnorodne plikowe działania wejścia-wyjścia, które dany moduł sterujący potrafi wykonywać. Jądro prawie wcale nie przetwarza wstępnie zamówienia na czytanie lub pisanie pliku na urządzeniu znakowym. Zamiast tego przekazuje po prostu zamówienie do wytypowanego urządzenia, pozostawiając mu obsługę zamówienia.

Głównym wyjątkiem od tej reguły jest specjalny podzbior modułów sterujących urządzeń znakowych, oprogramowujących terminal. Standardowy interfejs z takimi modułami sterującymi jądro obsługuje za pomocą zbioru struktur `tty_struct`. W każdej z tych struktur jest miejsce na bufore i dane sterujące przepływem znaków z terminalu według organizacji linii.

Organizacja linii (ang. *line discipline*) oznacza sposób interpretowania informacji pochodzących z urządzenia terminalu. Najpopularniejszą z organizacji linii jest tryb `tty`, w którym strumień danych z terminalu jest sklejany

w standardowe strumienie wejściowe i wyjściowe procesów wykonywanych przez użytkownika. Zadanie to komplikuje fakt, że jednocześnie może być wykonywany więcej niż jeden taki proces, więc tryb **tty** odpowiada za łączenie i rozłączanie wejścia i wyjścia terminalu z różnymi dołączonymi do niego procesami, stosownie do ich zawieszania lub budzenia przez użytkownika.

Są zrealizowane również inne organizacje linii, nie dotyczące wejścia-wyjścia procesu użytkowego. Protokoły sieciowe PPP i SLIP określają sposoby kodowania połączeń sieciowych za pomocą takich urządzeń końcowych jak linia szeregowa. Protokoły te są zrealizowane w systemie Linux w postaci modułów sterujących, które z jednej strony sprawiają dla systemu terminali wrażenie organizacji linii, a z drugiej strony dla systemu łączności sieciowej objawiają się jak moduły sterujące urządzeń sieciowych. Po ustaleniu dla urządzenia terminala którejś z takich organizacji linii, wszelkie dane pojawiające się na tym terminalu będą kierowane wprost do odpowiedniego modułu sterującego urządzenia sieciowego.

22.9 ■ Komunikacja międzyprocesowa

Środowisko systemu UNIX umożliwia procesom różnorodne sposoby wzajemnej łączności. Komunikacja może być po prostu kwestią powiadomienia drugiego procesu o wystąpieniu jakiegoś zdarzenia lub może obejmować przesyłanie danych od jednego procesu do innego.

22.9.1 Synchronizacja i sygnały

Standardowym mechanizmem systemu UNIX służącym do powiadamiania procesu o wystąpieniu zdarzenia jest *sygnał*. Sygnały mogą być przesyłane między dowolnymi procesami (z ograniczeniami dotyczącymi sygnałów wysyłanych do procesów należących do innego użytkownika), jednak liczba dostępnych sygnałów jest skończona i nie mogą one przenosić informacji – proces staje wyłącznie wobec faktu, że sygnał wystąpił. Sygnały nie muszą być generowane przez inne procesy. Jądro w swoim wnętrzu również tworzy sygnały. Może ono na przykład wysłać sygnał do procesu usługowego, gdy w kanale sieciowym pojawią się dane, bądź do procesu macierzystego, gdy proces potomny zakončzy działanie lub w przypadku wyczerpania czasu odliczanego przez czasomierz.

Jądro systemu Linux nie korzysta wewnętrznie z sygnałów w celu komunikowania się z procesami wykonywanymi w trybie jądra. Jeśli proces działający w trybie jądra oczekuje na wystąpienie zdarzenia, to zazwyczaj nie posłuży się sygnałami w celu zorientowania się o tym zdarzeniu. Zamiast tego komunikacja

w jądrze związana z pojawiającymi się asynchronicznymi zdarzeniami będzie się odbywać za pomocą planowania stanów i struktur `wait_queue`. Mechanizmy te umożliwiają procesom wykonywanym w trybie jądra na wzajemne informowanie się o stosownych zdarzeniach, jak również pozwalały na generowanie zdarzeń przez moduły sterujące urządzeń lub przez system współpracy z siecią. Kiedy proces zechce poczekać na wystąpienie jakiegoś zdarzenia, włącza się do kolejki oczekujących (`wait_queue`), skojarzonej z tym zdarzeniem, i informuje planistę, że od tego momentu nie nadaje się do wykonania. Wystąpienie zdarzenia powoduje obudzenie wszystkich procesów w kolejce `wait_queue`. Procedura taka pozwala wielu procesom oczekiwającym na jedno zdarzenie. Na przykład, jeżeli kilka procesów próbuje czytać plik z dysku, to wszystkie one zostaną obudzone po pomyślnym wczytaniu danych do pamięci.

Chociaż sygnały zawsze były głównym mechanizmem komunikowania asynchronicznych zdarzeń między procesami, Linux ma także zaimplementowany mechanizm semafora z systemu UNIX wydania V. Proces może czekać pod semaforem równie łatwo, jak to czyni w związku z sygnałem, ale semafory mają tę zaletę, że można je dzielić w dużych ilościach między wiele niezależnych procesów, a działania na licznych semaforach mogą być wykonywane w sposób niepodzielny. Do synchronizowania procesów komunikujących się za pomocą semaforów system Linux stosuje wewnętrznie swój standardowy mechanizm kolejki `wait_queue`.

22.9.2 Przekazywanie danych między procesami

Linux oferuje kilka mechanizmów przekazywania danych między procesami. Standardowy, uniksowy mechanizm potoku (`pipe`) umożliwia procesowi połomnemu dziedziczenie kanału komunikacyjnego do swojego przodka. Dane zapisane na jednym końcu potoku mogą być czytane na drugim. Potoki w systemie Linux wyglądają jak pewien typ i-węzła z oprogramowaniem wirtualnego systemu plików, a każdy potok ma dwie kolejki `wait_queue` do synchronizowania czytelnika i pisarza. System UNIX zawiera także definicje zestawu udogodnień sieciowych, których można używać do przesyłania strumieni danych zarówno do lokalnych, jak i do odległych procesów. Działania sieciowe są omówione w p. 22.10.

Istnieją również dwa inne sposoby dzielenia danych przez procesy. Pierwszy z nich – zastosowanie pamięci dzielonej – jest najszybszym środkiem przekazywania wielkich lub małych ilości danych. Wszelkie dane zapisane przez proces w obszarze pamięci dzielonej mogą być natychmiast czytane przez inny proces, który ma ten obszar odwzorowany w swojej przestrzeni adresowej. Główna wada pamięci dzielonej polega na tym, że sama w sobie nie daje żadnej synchronizacji. Proces nie może spytać systemu operacyjnego,

czy fragment pamięci dzielonej został zapisany, ani nie może zawiesić swego działania do czasu wystąpienia takiego zapisu. Pamięć dzielona nabiera szczególnej użyteczności w połączeniu z innym mechanizmem komunikacji międzyprocesowej, który zapewni brakującą synchronizację.

Obszar pamięci dzielonej w systemie Linux jest obiektem trwałym, który może być tworzony i usuwany przez procesy. Obiekt taki jest traktowany tak, jakby to była mała, niezależna przestrzeń adresowa. Algorytmy stronicowania systemu Linux mogą typować strony pamięci dzielonej do usuwania na dysk, zupełnie tak jak to robią ze stronami danych procesu. Obiekt pamięci dzielonej zachowuje się jak pamięć pomocnicza dla obszarów pamięci dzielonej, na podobieństwo pliku pełniącego funkcję pamięci pomocniczej dla obszarów pamięci odwzorowanych w pamięci operacyjnej. Gdy następuje odwzorowanie pliku w obszarze wirtualnej przestrzeni adresowej, wówczas wszelkie pojawiające się braki stron powodują odwzorowanie w pamięci wirtualnej odpowiednich stron pliku. Analogicznie, braki bezpośrednich odwzorowań stron pamięci dzielonej powodują odwzorowanie stron z trwałego obiektu pamięci dzielonej. Podobnie jak w przypadku plików, obiekty pamięci dzielonej utrzymują swoją zawartość nawet wtedy, gdy aktualnie żaden proces nie odwzorowuje ich w pamięci wirtualnej.

22.10 ■ Struktura sieci

Działania sieciowe są kluczowym obszarem funkcjonalności systemu Linux. Dostarcza on nie tylko standardowych protokołów internetowych, stosowanych w większości usług komunikacyjnych między systemami uniksowymi, lecz również realizuje pewną liczbę protokołów właściwych dla innych, nieuniksowych systemów operacyjnych. W szczególności, ponieważ Linux był początkowo zaimplementowany na komputerach PC, a nie na dużych stacjach roboczych ani na systemach klasy serwerów, zawiera on wiele protokołów powszechnie używanych w sieciach komputerów PC, a więc takich, jak Appletalk lub IPX.

Praca w sieci jest wewnętrznie w jądrze Linuxa implementowana za pomocą trzech warstw oprogramowania:

- interfejsu gniazd;
- modułów obsługi protokołów;
- modułów sterujących urządzeniami sieciowymi.

Aplikacje użytkownika dokonują wszelkich zamówień na usługi sieciowe za pomocą interfejsu gniazd. Jest on zaprojektowany tak, aby sprawiał wrażenie

warstwy gniazd systemu 4.3BSD, toteż dowolne programy zaprojektowane do korzystania z gniazd Berkeley będą działać w systemie Linux bez jakichkolwiek zmian kodu źródłowego. Ten interfejs jest opisany w p. 21.9.1. Jedną z ważnych cech interfejsu gniazd BSD jest jego ogólność, wystarczająca do reprezentowania adresów sieciowych w szerokim zakresie różnych protokołów sieciowych. Ten pojedynczy interfejs jest używany w Linuxie do dostępu nie tylko do protokołów zrealizowanych w standardowych systemach BSD, lecz również do wszystkich protokołów dostarczanych przez system.

Następną warstwą oprogramowania jest stos protokołów, przypominający pod względem organizacji ramy systemu BSD. Kiedy do warstwy tej nadchodzą dane z sieci – czy to z gniazda aplikacji, czy z modułu sterującego urządzenia sieciowego – oczekuje się, że będą zawierały identyfikator określający, który protokół sieciowy reprezentują. W razie potrzeby protokoły mogą komunikować się między sobą. Na przykład w zbiorze protokołów sieci Internet istnieją osobne protokoły do zarządzania wyznaczaniem tras, sygnalizacji błędów i niezawodnej retransmisji utraconych danych.

Warstwa protokołów może ponawiać zapisywanie pakietu, tworzyć nowe pakiety, dzielić je lub z powrotem łączy w większe porcje, lub po prostu pojmując nadchodzące dane. Gdy wróscie zakończy ona przetwarzanie zbioru pakietów, przekazuje je dalej: w górę, do interfejsu gniazda, jeśli dane są przeznaczone do lokalnej łączności, bądź w dół, do modułu sterującego urządzenia, jeżeli pakiet musi być przesłany na dużą odległość. Decyzja co do wyboru gniazda lub urządzenia do przesłania pakietu należy do warstwy protokołów.

Cała łączność między warstwami stosu sieciowego odbywa się przez przekazywanie pojedynczych struktur `skbuff`. Struktura `skbuff` zawiera zbiór wskaźników do jednego, ciągłego obszaru pamięci, reprezentującego bufor, wewnętrz którego mogą być konstruowane pakiety sieciowe. Poprawne dane struktur `skbuff` nie muszą zaczynać się od początku bufora `skbuff` i nie muszą ciągnąć się do jego końca. Oprogramowanie sieciowe może dokładać danych do bufora `skbuff` tak długo, jak długo wynik będzie się jeszcze w nim mieścił, może też obcinać je u końca każdego pakietu. Pojemność ta jest szczególnie ważna w przypadku nowoczesnych mikroprocesorów, w których polepszona szybkość jednostki centralnej znacznie przewyższa wydolność pamięci operacyjnej – architektura `skbuff` pozwala na elastyczne manipulowanie nagłówkami pakietów i sumami kontrolnymi, bez uciekania się do zasadniczego kopiowania danych.

Najważniejszym zbiorem protokołów w systemie sieciowym Linuxa jest kompletny protokoły internetowe (IP). W skład tego kompletu wchodzi kilka oddzielnych protokołów. Kompletny protokoły IP dokonuje wyboru tras między różnymi komputerami macierzystymi położonymi w dowolnym miej-

scu sieci. Na szczeblu protokołu wyznaczania tras są zbudowane protokoły UDP, TCP oraz ICMP. Protokół UDP przenosi między komputerami w sieci dowolne datagramy, a protokół TCP realizuje między komputerami sieciowymi niezawodne połączenia z gwarancją uporządkowania dostaw pakietów i automatyczną retransmisją danych utraconych. Protokół ICMP służy do przenoszenia między komputerami sieciowymi rozmaitych komunikatów o błędach i zaistniałych warunkach.

Zakłada się, że pakiety (struktury `skbuff`) docierające do sieciowego oprogramowania stosu protokołów będą zaopatrzone w wewnętrzny identyfikator wskazujący, do którego protokołu należy dany pakiet. Moduły sterujące różnych urządzeń sieciowych w różny sposób kodują typ protokołu za pomocą właściwych sobie środków komunikacji, dlatego identyfikacja protokołu dla nadchodzących danych musi dokonywać się w module sterującym urządzenia. Aby odnaleźć odpowiedni protokół, moduł sterujący urządzenia korzysta z tablicy haszowania znanych identyfikatorów protokołów sieciowych, po czym przekazuje pakiet odszukanemu protokołowi. Do tablicy haszowania można dodawać nowe protokoły w postaci modułów ładowanych przez jądro.

Nadchodzące pakiety IP trafiają do modułu sterującego IP. Zadaniem tej warstwy jest wyznaczanie tras. Rozstrzyga ona o miejscu przeznaczenia pakietu i przekazuje go do odpowiedniego, wewnętrznego programu obsługi protokołu w celu dostarczenia lokalnego lub umieszcza z powrotem w kolejce do modułu sterującego wybranego urządzenia sieciowego w celu przesłania dalej, do innego komputera sieciowego. Wybór trasy jest dokonywany przez moduł sterujący IP na podstawie dwóch tablic: trwałej bazy informacji *przekierowujących* (ang. *forwarding information base* – FIB) oraz podręcznej pamięci ostatnich wyborów tras. Baza FIB przechowuje informacje o konfiguracji tras i może określać trasy na podstawie konkretnego adresu przeznaczenia lub według *szablonu* (ang. *wildcard*) reprezentującego wiele adresów docelowych. Baza FIB ma postać zbioru tablic haszowania, indeksowanych za pomocą adresu przeznaczenia. Na samym początku przeszukuje się tablice reprezentujące trasy określone najbardziej szczegółowo. Pomyślne wyniki poszukiwań w takiej tablicy są dodawane do podręcznej tablicy tras, która dośćnie pamięta tylko trasy o dokładnej specyfikacji. Nie przechowuje się podręcznych szablonów, więc przeszukania mogą się odbywać szybciej. Wpis w podręcznej pamięci tras traci ważność, jeśli nie natrafiono na niego w ciągu ustalonego czasu.

W rozmaitych fazach oprogramowanie IP przekazuje pakiety do osobnej części kodu w celu pokonania zapory ogniowej, tj. wybrózegó odfiltrowania pakietów, stosownie do dowolnych kryteriów, mających na ogół na celu bezpieczeństwo. Zarządcą zapory ogniowej utrzymuje pewną liczbę osobnych łańcuchów zapór ogniowych (ang. *firewall chains*) i zezwala na porównanie

struktury `skbuff` z dowolnym łańcuchem. Osobne łańcuchy są zarezerwowane dla osobnych celów: jednego z nich używa się dla pakietów wysyłanych w dalszą drogę, inny służy do kontroli pakietów przychodzących do danego komputera macierzystego, a jeszcze inny jest używany dla informacji wytworzonych w danym komputerze. Każdy łańcuch jest przechowywany w postaci uporządkowanej listy reguł, przy czym reguła określa jedną z wielu możliwych funkcji decyzyjnych zapory ogniodowej wraz z dowolnymi danymi do porównywania.

Moduł sterujący protokołu IP wykonuje jeszcze dwa inne zadania: dzieli i porownie łączy wielkie pakiety. Jeśli pakiet wyjściowy jest za duży na umieszczenie go w kolejce do urządzenia, to dzieli się go po prostu na mniejsze *fragmenty*, z których każdy ustawia się w kolejce do modułu sterującego. W komputerze odbiorczym fragmenty te należy ze sobą poskładać. Moduł sterujący protokołu IP utrzymuje obiekt `ipfrag` dla każdego fragmentu oczekującego na połączenie z innymi oraz obiekt `ipq` dla każdego zestawianego datagramu. Nadchodzące fragmenty są porównywane ze wszystkimi znanymi obiektami `ipq`. W przypadku znalezienia dopasowania fragment dodaje się do obiektu. W przeciwnym razie tworzy się nowy obiekt `ipq`. Po nadaniu ostatniego fragmentu obiektu `ipq` następuje skonstruowanie zupełnie nowej struktury `skbuff` w celu przechowania nowego pakietu, a pakiet zostaje skierowany z powrotem do modułu sterującego IP.

Pakiety dopasowane przez protokół IP są, jako adresowane do danego komputera, przekazywane do jednego z pozostałych modułów obsługi protokołów. Protokoły UDP i TCP korzystają ze wspólnych środków kojarzenia pakietów z gniazdami źródłowymi i docelowymi. Każda połączona para gniazd jest jednoznacznie identyfikowana przez swoje adresy: źródłowy i docelowy oraz przez numery portu źródłowego i portu przeznaczenia. Listy gniazd są powiązane z tablicą haszowania budowaną na podstawie tych czterech wartości adresowo-portowych, aby można było w nadchodzących pakietach poszukiwać gniazd. Protokół TCP ma do czynienia z zawodnymi połączeniami, więc utrzymuje uporządkowane listy nie potwierdzonych pakietów wyjściowych w celu retransmitowania ich po upływie umownego czasu oraz po to, by pakiety przychodzące poza kolejnością można było zaprezentować gniazdu po nadaniu brakujących danych.

22.11 ■ Bezpieczeństwo

Model bezpieczeństwa systemu Linux jest blisko spokrewniony z mechanizmem bezpieczeństwa typowego systemu UNIX. Zagadnienia bezpieczeństwa można sklasyfikować w dwu grupach.

- **Uwierzytelnianie:** Zapewnienie, że nikomu nie uda się uzyskać dostępu do systemu bez uprzedniego wykazania, że ma do tego prawo.
- **Kontrola dostępu:** Dostarczanie mechanizmu sprawdzania, czy użytkownik ma prawo dostępu do danego obiektu, oraz realizowanie stosownej ochrony obiektów przed dostępem.

22.11.1 Uwierzytelnianie

Uwierzytelnianie w systemie UNIX na ogół odbywa się za pomocą pliku haseł jawnie udostępnianego do czytania. Hasło użytkownika zostaje zmieszane z wartością losową („przyprawione”), a wynik jest kodowany za pomocą jednokierunkowej funkcji transformacji i zapamiętyany w pliku haseł. Użycie funkcji jednokierunkowej oznacza, że wydedukowanie oryginalnego hasła na podstawie pliku haseł jest niemożliwe; co najwyżej można usiłować to zrobić metodą prób i błędów. Gdy użytkownik przedkłada hasło systemowi, jest ono mieszane z taką samą wartością losową przechowywaną w pliku haseł i poddawane tej samej transformacji jednokierunkowej. Hasło zostaje przyjęte, jeśli wynik tego przetworzenia zgadza się z wartością pamiętaną w pliku haseł.

W historii implementacji tego mechanizmu w systemie UNIX odnotowano kilka problemów. Hasy były częstokroć ograniczane do ośmiu znaków, a liczba możliwych wartości losowych była tak mała, że napastnik mógł bez trudu zestawić słownik popularnych haseł z wszystkimi możliwymi wartościami losowymi, zyskując niezłą szansę na dopasowanie jednego lub więcej haseł w pliku haseł i w efekcie otrzymanie bezprawnego dostępu do kont ze złamany zabezpieczeniem. Wprowadzono zatem rozszerzenia mechanizmu haseł polegające na przetrzymywaniu zaśzyfrowanego hasła w tajemnicy, w pliku niedostępnym do czytania dla ogólnego użytkownika, pozwoleniu na stosowanie dłuższych haseł lub korzystaniu z bezpieczniejszych metod kodowania hasła. Inne mechanizmy uwierzytelniania wprowadzono w celu ograniczenia czasu, w którym użytkownikowi wolno łączyć się z systemem lub rozprowadzać informacje uwierzytelniające do wszystkich pokrewnych systemów w sieci.

Aby rozwiązać te zagadnienia, kraj dostawców systemu UNIX opracował nowy mechanizm bezpieczeństwa. System *dolążalnych modułów uwierzytelniania* (ang. *pluggable authentication modules* – PAM) korzysta ze wspólnej biblioteki, którą może się posłużyć dowolna składowa systemu potrzebująca uwierzytelniać użytkowników. Implementacja tego systemu jest dostępna w systemie Linux. System PAM umożliwia ładowanie modułów uwierzytelniania na żądanie, zgodnie z ustaleniami zawartymi w ogólnosystemowym pliku konfiguracyjnym. Jeżeli w późniejszym czasie następuje dodanie nowego mechanizmu uwierzytelniania, to można go dodać do pliku

konfiguracyjnego i wszystkie części systemu będą mogły natychmiast zrobić z niego użytk. Moduły PAM mogą określać sposoby uwierzytelniania, ograniczenia działań na kontach, funkcje kształtuowania sesji lub funkcje zmiany hasła (tak aby podczas zmieniania haseł przez użytkowników następowalo natychmiastowe uaktualnienie wszystkich niezbędnych mechanizmów uwierzytelniania).

22.11.2 Kontrolowanie dostępu

Nadzorowanie dostępu w systemach uniksowych, w tym – w systemie Linux, odbywa się za pomocą niepowtarzalnych identyfikatorów numerycznych. Identyfikator użytkownika (ang. *user identifier* – uid) określa jednego użytkownika lub jeden zbiór praw dostępu. Identyfikator grupy (ang. *group identifier* – gid) jest dodatkowym identyfikatorem, który można stosować do określania praw przynależnych więcej niż jednemu użytkownikowi.

Kontrolowanie dostępu odnosi się do rozmaitych obiektów w systemie. Każdy plik osiągalny w systemie jest chroniony za pomocą standardowego mechanizmu kontrolowania dostępu. Ponadto inne obiekty użytkownika wspólnie, takie jak sekcje pamięci dzielonej i semafory, korzystają z tego samego systemu dostępu.

Z każdym obiektem w systemie UNIX, do którego dostęp jest kontrolowany przez użytkownika i grupę, jest skojarzony jeden identyfikator uid i jeden identyfikator gid. Procesy użytkowników też mają pojedyncze identyfikatory uid, ale mogą mieć więcej identyfikatorów gid. Jeśli identyfikator użytkownika (uid) procesu zgadza się z identyfikatorem użytkownika obiektu, to proces ma prawa użytkownika, czyli *właściciela* (ang. owner) tego obiektu. W przeciwnym razie, jeżeli któryś grupowy identyfikator (gid) procesu pasuje do grupowego identyfikatora obiektu, to nadaje się prawa grupowe. W pozostałym przypadku proces ma do obiektu *prawa reszty świata* (ang. *world rights*).

Linux nadzoruje dostępy przez przypisywanie obiektom *masek ochronnych* (ang. *protection masks*), określających, które tryby dostępu: pisanie, czytanie lub wykonywanie są udzielane procesom z dostępem właściciela, grupy lub świata. Tak więc właściciel obiektu może mieć pełny dostęp do pliku, tj. prawo czytania, pisania i wykonywania. Inni użytkownicy, zrzeszeni w pewnej grupie, mogą mieć dostęp do czytania, a jednocześnie zakaz zapisywania, a wszyscy pozostali mogą nie mieć żadnych praw dostępu.

Jednym wyjątkiem jest uprzywilejowany identyfikator użytkownika root. Procesowi z identyfikatorem tego specjalnego użytkownika automatycznie udziela się dostępu do dowolnego obiektu w systemie, z obejściem zwykłej procedury sprawdzania dostępu. Procesom takim udziela się również

pozwolenia na wykonywanie uprzywilejowanych operacji, takich jak czytanie dowolnego miejsca pamięci operacyjnej lub otwieranie zarezerwowanych gniazd sieciowych. Mechanizm ten umożliwia jądro zapobiegać dostępem zwykłych użytkowników do tych zasobów – większość wewnętrznych zasobów jądra należy niejawnie do użytkownika o identyfikatorze **root**.

W systemie Linux zrealizowano standardowy, uniksowy mechanizm **setuid**, opisany w p. 21.3.2. Pozwala on programom na działanie z przywilejami różnymi od tych, które ma użytkownik wykonywanego programu. Na przykład program **lpr** (umieszczający zadania w kolejce drukowania) ma dostęp do systemowej kolejki drukowania, pomimo że użytkownik wykonujący ten program nie ma takiego dostępu. Implementacja mechanizmu **setuid** w systemie UNIX odróżnia rzeczywiste (ang. *real*) i skuteczne (ang. *effective*) identyfikatory użytkownika. Rzeczywistym identyfikatorem użytkownika jest ten, który ma wykonawcą programu, a identyfikator skuteczny jest identyfikatorem właściciela pliku.

W systemie Linux mechanizm ten rozszerzono na dwa sposoby. Po pierwsze, Linux implementuje mechanizm *chronionego identyfikatora użytkownika* (ang. *saved user-id*), określony w normie POSIX, który pozwala procesowi raz po razie pozbywać się i ponownie nabywać swój identyfikator skuteczny; z powodów podktowanych bezpieczeństwem program może chcieć wykonywać większość swoich działań w trybie bezpiecznym, zrzekając się przywilejów uzyskanych na mocy statusu **setuid**, lecz może zyczyć sobie wykonania wybranych operacji z wszystkimi swoimi przywilejami. Implementacje standardowe systemu UNIX osiągają tę zdolność tylko przez wymianę rzeczywistych i skutecznych identyfikatorów użytkownika: poprzedni identyfikator skuteczny jest pamiętany, lecz rzeczywisty identyfikator użytkownika programu nie zawsze odpowiada identyfikatorowi użytkownika wykonującego dany program. Chronione identyfikatory użytkowników pozwalają procesowi na określanie jego skutecznego identyfikatora użytkownika za pomocą wartości jego rzeczywistego identyfikatora użytkownika, a następnie powrót do poprzedniej wartości jego skutecznego identyfikatora, bez potrzeby zmieniania rzeczywistego identyfikatora użytkownika za każdym razem.

Drugim ulepszeniem istniejącym w systemie Linux jest dodawanie charakterystyki procesu, co powoduje udzielanie jedynie podzbioru praw skutecznego identyfikatora użytkownika. Do udzielania praw dostępu do plików stosuje się cechy procesu o nazwach **fsuid** i **fsgid**; są one określane każdorazowo przy określaniu skutecznego identyfikatorów **uid** lub **gid**. Jednakże identyfikatory **fsuid** i **fsgid** mogą być określone niezależnie od identyfikatorów skutecznego, co umożliwia procesowi dostęp do plików w imieniu innego użytkownika bez przejmowania jego identyfikacji w żaden inny sposób. W szczególności procesy serwera mogą korzystać z tego mechanizmu do

udostępniania plików pewnemu użytkownikowi, nie narażając się na zakończenie lub zawieszenie przez tego użytkownika.

Linux ma jeszcze jeden mechanizm, który spopularyzował się w nowoczesnych systemach uniksowych jako środek elastycznego przekazywania praw od jednego programu do drugiego. Po utworzeniu między dwoma dowolnymi procesami w systemie lokalnego gniazda sieciowego każdy z procesów może wysłać drugiemu deskryptor jednego ze swoich otwartych plików: drugi proces otrzymuje kopię deskryptora odnoszącą się do tego samego pliku. Mechanizm ten umożliwia klientowi wybiórcze przekazanie dostępu do jednego pliku jakiemuś procesowi serwera, bez dawania temu procesowi jakichkolwiek innych przywilejów. Na przykład staje się już zbędne, aby serwer drukowania miał możliwość czytania wszystkich plików użytkownika przedkładającego nowe zadanie do druku. Klient chcący coś wydrukować może po prostu przekazać serwerowi deskryptory plików do drukowania, zakazując mu dostępu do wszelkich innych plików użytkownika.

22.12 ■ Podsumowanie

Linux jest nowoczesnym, będącym w wolnym obiegu systemem operacyjnym, którego podstawą są standardy systemu UNIX. Zaprojektowano go w celu wydajnej i niezawodnej pracy na typowych komputerach osobistych, choć działa on także na różnych innych platformach. Dostarczany przez system Linux interfejs programowy oraz interfejs użytkownika wykazuje zgodność ze standardowymi systemami uniksowymi – pod nadzorem tego systemu można wykonywać wiele aplikacji systemu UNIX, łącznie z rosnącą liczbą aplikacji komercyjnych.

Linux nie ewoluował w próżni. Pełny system Linux zawiera wiele składowych opracowanych niezależnie od niego. Jądro systemu operacyjnego Linux jest w całości oryginalne, lecz umożliwia wykonywanie sporej ilości oprogramowania uniksowego znajdującego się w wolnym obiegu. Powstaje w wyniku tego całkowicie zgodny z uniksowym standardem system operacyjny, wolny od zastrzeżonego kodu.

Jądro systemu Linux jest wykonane w tradycyjnej, monolitycznej postaci, co jest podyktowane względami wydajnościowymi. Jego projekt jest jednak modularny na tyle, że pozwala na dynamiczne ładowanie i rozładowywanie modułów sterujących w trakcie działania systemu.

Linux jest systemem wielodostępnym, chroniącym procesy przed sobą i wykonującym wiele procesów według planowania z podziałem czasu. Nowo utworzone procesy mogą dzielić wybrane części swojego środowiska wykonywania z ich procesami macierzystymi, co pozwala na programowanie wie-

lowątkowe. Komunikację międzyprocesową zapewniają zarówno mechanizmy Systemu V, tj. kolejki komunikatów, semafory i pamięć dzielona, jak i interfejs gniazd systemu BSD. Poprzez interfejs gniazd istnieje możliwość jednoczesnego dostępu do wielu protokołów sieciowych.

Dla użytkownika system plików wygląda jak hierarchiczne drzewo katalogów o semantyce uniksowej. Wewnątrz systemu Linux zastosowano warstwę abstrakcji pozwalającą na zarządzanie różnymi systemami plików. W użyciu są systemy plików urządzeń, sieciowe i wirtualne. Systemy plików urządzeń kontaktują się z pamięcią dyskową za pośrednictwem dwu pamięci podręcznych: dane przechowuje się w pamięci podręcznej stron, która jest zunifikowana z systemem pamięci wirtualnej; przechowuje się również metadane w podręcznej pamięci buforów, czyli oddzielnej pamięci podręcznej, indeksowanej za pomocą fizycznego bloku dyskowego.

System zarządzania pamięcią posługuje się dzieleniem stron i techniką kopiowania przy zapisie, aby minimalizować liczbę zbędnych kopii danych używanych wspólnie przez różne procesy. Strony są ładowane na żądanie z chwilą pierwszego do nich odwołania, a w przypadku konieczności odzyskania pamięci operacyjnej powrotnie kopowanie stron do pamięci pomocniczej odbywa się według algorytmu LFU.

■ Ćwiczenia

- 22.1** Linux działa na różnorodnym sprzęcie. Jakie kroki muszą podjąć osoby pracujące nad rozwojem systemu Linux, aby zapewnić jego przenoszenie na różne procesory i architektury zarządzania pamięcią oraz osiągnięcie minimalizacji ilości kodu jądra zależnego od architektury?
- 22.2** Dynamicznie ładowalne moduły jądra uelastyczniają dołączanie modułów sterujących do systemu. Czy jednak mają one też wady? W jakich warunkach opłacaloby się skompilowanie jądra do postaci jednego pliku binarnego, a kiedy byłoby lepiej trzymać je podzielone na moduły? Odpowiedź uzasadnij.
- 22.3** Wielowątkowość jest popularną techniką programowania. Opisz trzy różne sposoby implementowania wątków. Wyjaśnij, jak mają się te sposoby do mechanizmu `clone` systemu Linux. Kiedy każdy z tych alternatywnych mechanizmów może okazać się lepszy, a kiedy gorszy od stosowania klonów?
- 22.4** Jakie dodatkowe koszty są ponoszone przy tworzeniu i planowaniu procesu w porównaniu z kosztem zarządzania sklonowanym wątkiem?

- 22.5 Procedura planująca systemu Linux realizuje *łagodne* planowanie w czasie rzeczywistym. Jakich cech niezbędnych w pewnych zastosowaniach czasu rzeczywistego brakuje w takim planowaniu? W jaki sposób można je dodać do jądra?
- 22.6 Jądro systemu Linux nie pozwala na stronicowanie poza swoją pamięcią. Jaki wpływ wywiera to ograniczenie na projekt jądra? Jakie są dwie zalety i dwie wady takiej decyzji projektowej?
- 22.7 Wspólnie użytkowane biblioteki wykonują wiele operacji mających zasadnicze znaczenie w systemie operacyjnym Linux. Na czym polega zaleta trzymania tych funkcji poza jądrem? Czy pociąga to za sobą jakieś niedogodności? Odpowiedź uzasadnij.
- 22.8 Podaj trzy zalety dynamicznego łączenia bibliotek (do wspólnego użytku) w porównaniu z łączeniem statycznym? Wskaż dwa przypadki, w których można zalecać łączenie statyczne.
- 22.9 Porównaj zastosowanie gniazd sieciowych z zastosowaniem pamięci dzielonej jako mechanizmu przekazywania danych między procesami w jednym komputerze. Jakie zalety ma każda z metod? Kiedy każda z nich może być szczególnie zalecana?
- 22.10 W systemach uniksowych przy optymalizacji rozmieszczenia danych na dysku korzystano z ich obrotowej lokalizacji. jednak współczesne implementacje, w tym Linux, optymalizują po prostu sekwencyjny dostęp do danych. Dlaczego tak robią? Jaka cecha sprzętu decyduje o zakresie dostępu sekwencyjnego? Dlaczego optymalizacja obrotowa przestaje być przydatna?
- 22.11 Kod źródłowy systemu Linux jest dostępny szeroko i bez opłat za pośrednictwem sieci Internet lub dostawców płyt CD-ROM. Jakie są trzy konsekwencje tej dostępności dla bezpieczeństwa systemu Linux?

Uwagi bibliograficzne

System Linux jest produktem internetowym, wskutek czego większość opisującej go dokumentacji jest osiągalna w pewnych formach w sieci Internet. Pod następującymi, podstawowymi adresami sieciowymi znajduje się najwięcej użytecznych informacji:

- Zestawienie stron systemu Linux (Linux Cross-Reference Pages) pod adresem <http://lxr.linux.no/> zawiera bieżące teksty jądra Linux, które wraz z pełnym skorowidzem można przeglądać za pomocą sieci WWW.

- Pod adresem <http://www.linuxhq.com/> zgromadzono wielkie ilości informacji Linux-HQ, odnoszących się do jądra Linux 2.0 oraz jądra Linux 2.1. W komputerze tym znajdują się również odsyłacze do stron mierzących większość dystrybucji systemu Linux, a także archiwum ważniejszych list dyskusyjnych.
- Linux Documentation Project dostępny za pomocą adresu <http://sunsite.unc.edu/linux/> zawiera wykazy wielu książek poświęconych systemowi Linux, dostępnych w formacie źródłowym jako część projektu dokumentacji Linuxa. W projekcie tym znajdują się także przewodniki Linux How-To, czyli seria wskazówek i porad dotyczących aspektów systemu Linux.
- Poradnik Kernel Hackers' Guide jest internetowym przewodnikiem ogólnych, wewnętrznych rozwiązań jądra. To stale rozwijające się stanowisko jest zlokalizowane pod adresem

<http://www.redhat.com:8080/HyperNews/get/khg.html>

Systemowi Linux jest ponadto poświęconych wiele list dyskusyjnych. Najważniejsza spośród nich jest utrzymywana przez zarządcę, którego można znaleźć pod adresem majordomo@vger.rutgers.edu. Aby uzyskać informacje na temat dostępu do serwera list i zapisania się na dowolną z list wystarczy wysłać pod ten adres pocztę zawierającą jeden wiersz o treści: help.

Jedyną książką opisującą obecnie wewnętrzne szczegóły jądra Linux jest *Linux Kernel Internals* autorstwa Becka i in. [26] – odnosi się ona tylko do jądra Linux 1.2.13. Jako dalszą ogólną lekturę dotyczącą systemu UNIX można polecić na dobry początek książkę *UNIX Internals: The New Frontiers*, której autorem jest Vahalia [435].

Dodajmy na koniec, że za pomocą sieci Internet możnatrzymać sam system Linux. Kompletne dystrybucje systemu Linux są osiągalne w mierzących komputerach firm zainteresowanych tym projektem. Archiwia bieżących składowych systemu są również utrzymywane przez społeczność linukową w kilku miejscach sieci Internet. Najważniejsze z nich mają adresy:

- <ftp://tsx-11.mit.edu/pub/linux/>
- <ftp://sunsite.unc.edu/pub/Linux/>
- <ftp://linux.kernel.org/pub/linux/>



Rozdział 23

SYSTEM WINDOWS NT

Windows NT firmy Microsoft jest 32-bitowym systemem operacyjnym działającym wielozadaniowo i z wywłaszczeniem; jest przeznaczony dla nowoczesnych mikroprocesorów. System NT daje się przenosić na różnorodne architektury procesorów. Jedną lub więcej wersji systemu NT przeniesiono na procesory Intel 386 i nowsze, MIPS R4000, DEC Alpha oraz PowerPC. Podstawowe cele systemu to: przenośność, bezpieczeństwo, zgodność ze standardem IEEE 1003.1 interfejsu przenośnego systemu operacyjnego (ang. *Portable Operating System Interface – POSIX*), możliwość korzystania z wieloprocesorów, rozszerzalność, adaptacje międzynarodowe i zgodność z aplikacjami systemów MS-DOS i MS-Windows. W systemie NT zastosowano architekturę mikrojądra (podobnie jak w systemie Mach), dzięki czemu daną część systemu można ulepszać bez zbytniego naruszania jego innych części. System NT (do wersji 4) nie jest wielodostępnym systemem operacyjnym.

Istnieją dwie wersje systemu NT: stacja robocza Windows NT oraz serwer Windows NT. Obie korzystają z tego samego jądra i kodu systemu operacyjnego, lecz oprogramowanie serwera NT jest skonfigurowane do pracy z aplikacjami typu klient-server i może działać jako serwer aplikacji w sieciach lokalnych NetWare i Microsoft. Wersja 4.0 serwera NT zawiera oprogramowanie internetowego serwera WWW oraz interfejs użytkownika systemu Windows 95. W 1996 r. sprzedano więcej licencji serwera NT niż wszystkich licencji różnych wersji systemu UNIX.

23.1 ■ Historia

W połowie lat osiemdziesiątych firmy Microsoft i IBM współpracowały przy tworzeniu systemu operacyjnego OS/2, który napisano w języku asemblera dla jednoprocesorowych systemów Intel 80286. W 1988 r. firma Microsoft postanowiła zacząć wszystko od nowa i opracować przenośny system operacyjny, który zawierałby interfejs API (ang. *application programming interface*) programowania zarówno aplikacji systemu OS/2, jak i standardu POSIX. W październiku 1988 r. do zbudowania takiego systemu wynajęto Dave'a Cutlera, konstruktora systemu operacyjnego VMS dla komputerów VAX firmy DEC. Z początku zakładano, że naturalnym środowiskiem systemu NT będzie platforma programowa API systemu OS/2, lecz w trakcie prac rozwojowych system NT zmieniono tak, aby można było w nim korzystać z 32-bitowego środowiska Windows API lub Win32 API, wychodząc na przeciw popularności systemu Windows 3.0. Pierwszymi wersjami systemu NT były: Windows NT 3.1 oraz zaawansowany serwer Windows NT 3.1. (W tym czasie 16-bitowy system Windows miał wydanie 3.1). W wersji 4.0 systemu NT zaadaptowano interfejs użytkownika Windows 95, a także wdrożono internetowy serwer WWW oraz oprogramowanie przeglądarki. Ponadto niektóre procedury interfejsu użytkownika oraz kod graficzny przesunięto do jądra w celu polepszenia wydajności, czego ubocznym skutkiem było zmniejszenie niezawodności systemu.

23.2 ■ Podstawy projektu

Cele projektowe, które firma Microsoft postawiła przed systemem NT, obejmowały rozszerzalność, przenośność, niezawodność, zgodność, wydajność oraz dostosowanie do wymogów międzynarodowych.

Rozszerzalność (ang. *extensibility*) jest ważną cechą każdego systemu; pokłada się w niej nadzieję dorównania postępu w technice obliczeniowej. Ze względu na możliwość przyszłych zmian system NT zrealizowano z zastosowaniem architektury warstwowej. Egzekutor systemu NT, działający w trybie jądra, czyli chronionym, dostarcza podstawowych usług systemowych. Powyżej egzekutora działa w trybie użytkownika kilka podsystemów usługowych. Należą do nich *podsystemy środowiskowe* (ang. *environmental subsystems*) emulujące różne systemy operacyjne. Tak więc programy napisane dla systemów MS-DOS, Microsoft Windows i POSIX mogą pracować w systemie NT w odpowiednim otoczeniu. (Więcej informacji o podsystemach środowiskowych zawiera p. 23.4). Dzięki modularnej strukturze można dodawać do systemu następne podsystemy środowiskowe bez naruszania

egzekutora. Ponadto w systemie NT stosuje się ładowalne moduły sterujące systemu wejścia-wyjścia, a więc system może być uzupełniany podczas działania o nowe systemy plików, nowe rodzaje urządzeń wejścia-wyjścia oraz nowe rodzaje sieci. W systemie NT znajduje zastosowanie model klient-serwer na podobieństwo systemu operacyjnego Mach; jest również możliwe przetwarzanie rozproszone za pomocą zdalnych wywołań procedur (RPC), zdefiniowanych przez konsorcjum Open System Foundation.

System operacyjny jest *przenośny* (ang. *portable*), jeżeli daje się przetransportować z jednej architektury sprzętowej na inną ze stosunkowo niewielką ilością zmian. System NT zaprojektowano jako system przenośny. Podobnie jak system UNIX, większość systemu NT napisano w językach C i C++. Cały kod zależny od procesora wyizolowano w bibliotece dodatkowej dynamicznie (DLL), nazywanej *warstwą abstrakcji sprzętu* (ang. *hardware abstraction layer* – HAL). Biblioteka DLL jest plikiem, którego zawartość odwzorowuje się w przestrzeni adresowej procesu w taki sposób, że wszelkie umieszczone w niej funkcje zachowują się tak, jakby były częścią procesu. Górnne warstwy systemu NT zależą od warstwy HAL, a nie od znajdującego się pod nią sprzętu, co ułatwia przenoszenie systemu NT. Warstwa HAL działa bezpośrednio na sprzęcie, izolując resztę systemu NT od różnic sprzętowych między platformami, na których przychodzi mu działać.

Przez *niezawodność* (ang. *reliability*) rozumie się zdolność radzenia sobie z sytuacjami błędymi, w co wchodzi również zdolność systemu operacyjnego do ochrony samego siebie i jego użytkowników przed wadliwym lub złośliwym oprogramowaniem. Projekt systemu NT uodparnia go na uszkodzenia i ataki za pomocą sprzętowej ochrony pamięci wirtualnej oraz programowych mechanizmów ochrony zasobów systemu operacyjnego. System NT jest także zaopatrzony w system plików, zwany rdzennym systemem plików NT (NTFS), który automatycznie usuwa skutki wielu rodzajów błędów w systemie plików powstających po awarii systemu. System NT w wersji 3.51 ma rzadową, amerykańską klasę bezpieczeństwa C-2, co oznacza umiarkowany poziom ochrony przed wadliwym oprogramowaniem lub złośliwym atakiem.

System NT zapewnia *zgodność* (ang. *compatibility*) na poziomie kodu źródłowego dla aplikacji spełniających standard IEEE 1003.1 (POSIX); można je zatem kompilować do pracy w systemie NT bez zmian w ich kodzie źródłowym.

Ponadto system NT może wykonywać wiele programów binarnych skompilowanych dla procesorów Intel X86 działających pod kontrolą systemów MS-DOS, 16-bitowego systemu Windows, OS/2, Lan Manager i Windows w wersji 32-bitowej. Osiaga to za pomocą dwóch mechanizmów. Po pierwsze, wersje systemu NT dla procesorów innych niż Intel umożliwiają programową emulację zbioru rozkazów X86. Po drugie, odwołania do sys-

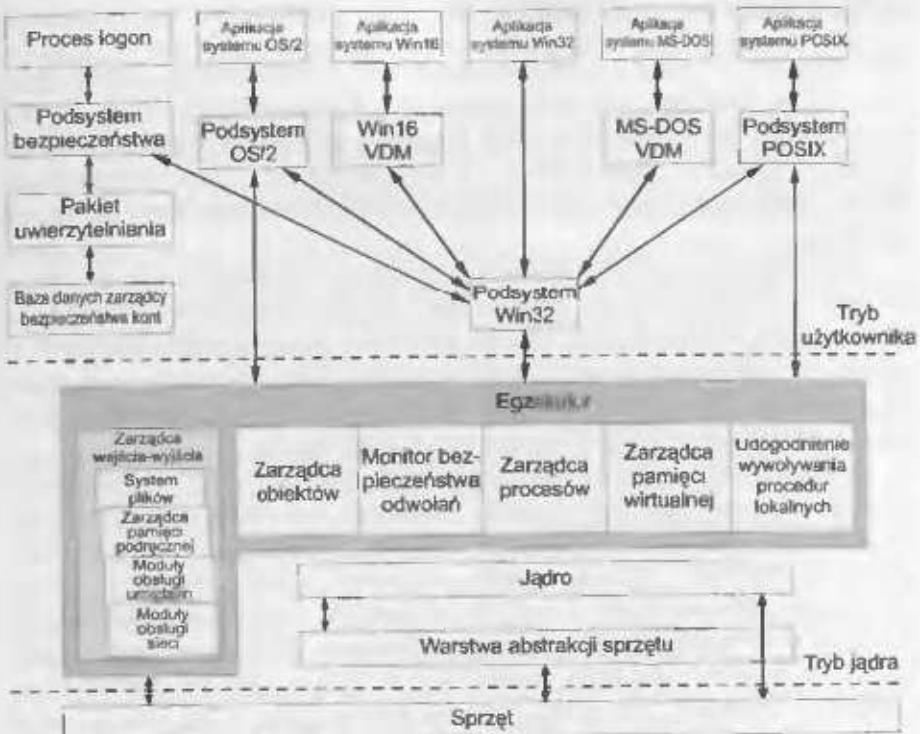
mów MS-DOS, 32-bitowego Windows itd. są obsługiwane przez podsystemy środowiskowe, o których wspomnialiśmy uprzednio. Podsystemy środowiskowe wspomagają działanie wielu systemów plików, m.in. takich jak: system plików FAT MS-DOS, system plików HPFS OS/2, system plików ISO9660 CD oraz system NTFS. Binarna zgodność systemu NT nie jest jednak doskonała. Na przykład w systemie MS-DOS programy użytkowe mogą bezpośrednio kontaktować się z portami sprzętowymi. Ze względu na niezawodność i bezpieczeństwo system NT kontaktów takich zabrania.

System NT zaprojektowano z celu osiągnięcia dobrej *wydajności* (ang. *performance*). Wchodzące w jego skład podsystemy mogą wydajnie komunikować się ze sobą za pomocą udoskonalenia w wywoływaniu lokalnych procedur zapewniającego szybkie przekazywanie komunikatów. Istnieje możliwość wywłaszczenia wątków w podsystemach NT przez wątki o wyższym priorytecie; nie dotyczy to tylko wątków jądra. Dzięki temu system może szybko reagować na zdarzenia zewnętrzne. Co więcej, system NT zaprojektowano do wieloprzetwarzania symetrycznego. W komputerze wieloprocesorowym kilka wątków może działać w tym samym czasie. W porównaniu z systemem UNIX aktualna skalowalność systemu NT jest ograniczona. W 1997 r. umożliwiał on działanie konfiguracji sprzętowych z co najwyżej osiemnoma jednostkami centralnymi, podczas gdy system Solaris mógł obsługiwać do 64 procesorów.

System NT ma również spełniać *wymogi międzynarodowe*. Zapewnia on wsparcie dla różnych cech lokalnych, urzeczywistniając je za pomocą interfejsu API z wdrożonymi właściwościami językowymi poszczególnych narodów (ang. *national language support* – NLS). Interfejs API NLS zawiera różnorodne procedury formatowania daty, czasu i kwot pieniężnych – stosownie do zwyczajów panujących w danym kraju. Porównywanie napisów wyspecjalizowano tak, aby działało dla różnych zbiorów znaków. Oryginalnym kodem systemu NT jest UNICODE, jednakże system NT umożliwia stosowanie znaków kodu ASCII, poprzedzając działania na nich zamianą na znaki kodu UNICODE (zamiana kodu 8-bitowego na 16-bitowy).

23.3 ■ Elementy systemu

Architekturę systemu NT tworzy warstwowy układ modułów. Na rysunku 23.1 jest przedstawiona architektura systemu NT w wersji 4. Głównymi warstwami są tu: warstwa abstrakcji sprzętu, jądro oraz egzekutor – działające w trybie chronionym, jak również duży zbiór podsystemów działających w trybie użytkownika. Podsystemy działające w trybie użytkownika należą do dwu klas. Podsystemy środowiskowe emulują różne systemy operacyjne



Rys. 23.1 Schemat blokowy systemu Windows NT

Podsystemy ochrony dostarczają funkcji zabezpieczających. Jedną z głównych zalet architektury tego typu jest prosta utrzymywanie współpracy między modułami. W pozostałej części rozdziału opisujemy te warstwy i podsystemy.

23.3.1 Warstwa abstrakcji sprzętu

Warstwa HAL (ang. *hardware abstraction layer*) jest oprogramowaniem, które ukrywa różnice sprzętowe przed górnymi warstwami systemu operacyjnego, aby ułatwiać uczynienie systemu NT przencsnym. Warstwa HAL eksportuje interfejs maszyny wirtualnej, z którego korzystają: jądro, egzekutor i moduły sterujące urządzeniami. Jedną z zalet takiego podejścia jest to, że wystarcza, aby każdy moduł sterujący występował tylko w jednej wersji: może on działać na wszystkich rodzajach sprzętu bez przenoszenia kodu modułu sterującego. Warstwa HAL umożliwia także przetwarzanie symetryczne. Z uwagi na wydajność moduły sterujące wejścia-wyjścia (oraz moduły obsługi urządzeń graficznych w systemie NT 4.0) mogą bezpośrednio kontaktować się ze sprzętem.

23.3.2 Jądro

Jądro systemu NT stanowi podstawę egzekutora i podsystemów. Strony jądra nigdy nie są usuwane z pamięci operacyjnej, a jego działanie nigdy nie doznaje skutków wywłaszczenia. Jądro ma cztery główne obowiązki: planować procesy, obsługiwać przerwania i sytuacje wyjątkowe, synchronizować na niskim poziomie procesor oraz podejmować działania naprawcze po awarii zasilania.

Jądro jest zorientowane obiektywnie. Przez *typ obiektu* rozumie się zdefiniowany w systemie NT typ danych, który ma zbiór atrybutów (wartości danych) i zbiór metod (tj. funkcji lub procedur). Reprezentant (egzemplarz) danego typu obiektowego nazywa się *obiektem*. Jądro wykonuje swoje zadania, posługując się zbiorem obiektów jądrowych, których atrybutami są dane jądra, a metody owych obiektów realizują działania jądra.

Jądro korzysta z dwóch zbiorów obiektów. Pierwszy zbiór tworzą *obiekty ekspedytora* (ang. *dispatcher objects*). Nadzorują one delegowanie procesów do procesora oraz synchronizację w systemie. Przykładami obiektów tego zbioru są zdarzenia, mutanty, zamki, semafory, wątki oraz czasomierze. *Obiekt typu zdarzenie* (ang. *event object*) służy do odnotowania wystąpienia zdarzenia oraz do synchronizowania go z pewnym działaniem. *Mutant* (ang. *mutant*) umożliwia wzajemne wykluczanie w trybie jądra lub użytkownika, z pojęciem własności. *Zamek* (ang. *mutex*)¹ jest osiągalny tylko w trybie jądra i zapewnia wzajemne wykluczanie, chroniąc przed zakleszczeniami. *Obiekt semaforowy* (ang. *semaphore object*) działa jak licznik lub furtka do nadzorowania liczby wątków mających dostęp do jakiegoś zasobu. *Obiekt wątku* (ang. *thread object*) jest wykonywany przez jądro i skojarzony z *obiektem procesu* (ang. *process object*). *Obiekt czasomierza* (ang. *timer object*) używa się do dozorowania czasu i sygnalizowania jego przekroczeń (ang. *time outs*), gdy operacje trwają za dugo i należy je przerwać.

Do drugiego zbioru obiektów jądra należą *obiekty sterujące*. Są tu asynchroniczne wywołania procedur, przerwania, sygnały dotyczące zasilania i jego stanu oraz procesy i obiekty profilujące. *Asynchroniczne wywołanie procedury* stosuje się w celu przerwania wykonywanego wątku i wywołania procedury. *Obiekty przerwań* (ang. *interrupt objects*) łączą procedury obsługi przerwań ze źródłami przerwań. *Obiekt informujący o zasilaniu* (ang. *power notify object*) służy do automatycznego wywołania ustalonej procedury po awarii zasilania, a *obiekt stanu zasilania* (ang. *power status object*) służy do sprawdzania, czy wystąpiła awaria zasilania. *Obiekt procesu* jest reprezentowany przez wirtualną przestrzeń adresową i informacje niezbędne do wyko-

¹ Od słów *mutual exclusion* – wzajemne wykluczanie. – Przyp. tłum.

nywania zbioru skojarzonych z procesem wątków. Wreszcie *obiekt profilujący* (ang. *profile object*) stosuje się do mierzenia ilości czasu zużywanego przez blok kodu.

Tak jak przyjęto w wielu nowoczesnych systemach operacyjnych, w systemie NT wyróżnia się w odniesieniu do wykonywalnego kodu pojęcia procesów i wątków. *Proces* dysponuje przestrzenią adresową i informacjami takimi, jak podstawowy priorytet oraz przypisanie do jednego lub większej liczby procesorów. Każdy proces ma jeden lub więcej wątków, które są jednostkami wykonywania zarządzanymi przez jądro. Każdy wątek ma własny stan, w tym – priorytet, przypisanie do procesora i informacje rozliczeniowe.

Wątek może się znajdować w jednym z sześciu stanów: gotowości, gotowania, wykonywania, oczekiwania, przejściowym i zakończenia. Gotowość oznacza oczekивание na wykonywanie. Wątek w stanie gotowości i z najwyższym priorytetem przechodzi do stanu pogotowania (ang. *standby*), co oznacza, że zostanie wykonany w pierwszej kolejności. W systemie wieloprocesorowym dla każdego procesora jest utrzymywany jeden wątek w stanie pogotowania. Wątek jest w stanie wykonywania, gdy procesor wykonuje jego instrukcje. Wątek taki będzie działał aż do wywłaszczenia przez wątek o większym priorytecie lub do swojego zakończenia, albo do upływu przydzielonego mu kwantu czasu, względnie do momentu, w którym wywoła funkcję systemową powodującą jego zablokowanie. Nowy wątek jest w stanie przejściowym (ang. *transition*), jeżeli oczekuje na zasoby niezbędne do działania. Wątek wchodzi w stan końcowy z chwilą, gdy kończy swoje działanie.

Ekspedytor przy określaniu kolejności wątków korzysta z 32 priorytetów. Priorytety są podzielone na dwie klasy: klasa czasu rzeczywistego zawiera wątki o priorytetach z przedziału 16-31, a klasa zmienia zawsze wątki o priorytetach 0-15. Z każdym priorytetem planowania przydzielu procesora ekspedytor wiąże kolejkę i przebiega zbiór kolejek od najwyższej do najniższej, aż znajdzie wątek gotowy do pracy. Jeżeli wątek ma specjalne przypisanie procesora, lecz dany procesor nie jest dostępny, to ekspedytor pomija go i kontynuuje poszukiwanie wątku gotowego do działania. Jeśli nie znajdzie się żadnego wątku w stanie gotowości, to ekspedytor podejmie wykonywanie specjalnego wątku zwanego *hezczynnym*.

Wątek, którego kwant czasu się wyczerpie, zostaje przerwany i jeśli jego priorytet należał do klasy zmiennej, to mu się go zmniejsza. Jednak priorytet nigdy nie spada poniżej priorytetu podstawowego. Oznaczanie priorytetu wątku zmierza do zmniejszenia czasu użytkowania jednostki centralnej przez wątki ograniczone obliczeniami. Kiedy wątek o priorytecie zmiennym uwalnia się od czekania spowodowanego jakąś operacją, wtedy ekspedytor zwiększa jego priorytet. Stopień zwiększenia priorytetu zależy od tego, na co wątek czekał. Przyrost priorytetu wątku, który czekał na dane z klawiatury, będzie większy,

natomiasz w odniesieniu do wątku, który czekał na operację dyskową, przyrost ten będzie bardziej umiarkowany. Strategia ta ma na celu tworzenie dobrych czasów odpowiedzi w wątkach interakcyjnych, korzystających z myszki i okien, i pozwala wątkom ograniczonym przez wejście-wyjście na utrzymywanie urządzeń wejścia-wyjścia w ruchu, umożliwiając wątkom ograniczonym przez procesor korzystanie z zaoszczędzonych cykli procesora w trybie drugoplanowym. Strategię tę stosuje się w kilku systemach operacyjnych z podziałem czasu, w tym systemie UNIX. Okno, w którym użytkownik prowadzi konwersację, również otrzymuje podwyższony priorytet w celu polepszenia czasu reakcji.

Do planowania przydziału procesora może dochodzić wówczas, gdy wątek przechodzi do stanu gotowości lub oczekiwania, gdy wątek kończy działanie lub wtedy, kiedy aplikacja zmienia priorytet wątku lub przypisanie procesora. Jeżeli wysokopriorytetowy wątek czasu rzeczywistego staje się gotowy do działania w czasie, gdy działa wątek niskopriorytetowy, to niskopriorytetowy wątek zostaje wywłaszczyony. Wywłaszczenie takie daje wątkowi czasu rzeczywistego preferencyjny dostęp do procesora, jeśli wątek dostępu takiego potrzebuje. Niemniej jednak system NT nie jest rygorystycznym systemem czasu rzeczywistego, gdyż nie zapewnia, że wątek czasu rzeczywistego rozpoczęcie działanie w konkretnym limicie czasu.

W jądrze odbywa się także obsługa pułapek powodowanych wyjątkami i przerwaniemi generowanymi przez sprzęt lub oprogramowanie. System NT określa kilka wyjątków niezależnych od jego architektury, m.in. takich, jak niedozwolona próba odwołania do pamięci, nadmiar stałopozycyjny, nadmiar lub niedomiar zmennopozycyjny, dzielenie całkowite przez zero, dzielenie zmennopozycyjne przez zero, niedozwolony rozkaz, złe przyleganie danych, rozkaz uprzywilejowany, błąd czytania strony, próba naruszenia ochrony strony, przekroczenie rozmiaru zbioru stronicowania, osiągnięcie punktu kontrolnego przez program uruchomiony oraz wykonanie jednego kroku tegoż programu.

Proste wyjątki mogą być obsługiwane przez procedurę obsługi pułapki: obsługa innych należy do obowiązków *ekspedytora wyjątków* (ang. *exception dispatcher*). Ekspedytor wyjątków tworzy rekord z cipsem przyczyny wyjątku i znajduje procedurę, która może się zająć obsługą wyjątku.

Jeśli zdarzenie wyjątkowe pojawi się w trybie jądra, to ekspedytor wyjątków wywołuje po prostu podprogram lokalizujący procedurę obsługi wyjątku. W przypadku nieodnalezienia takiej procedury powstaje poważny błąd systemowy i użytkownik pozostaje z niesławnym „błękitnym ecranem śmierci”, znamionującym awarię systemu.

W przypadku procesów działających w trybie użytkownika obsługa wyjątków jest bardziej skomplikowana, gdyż podsystem środowiskowy (w ro-

dzaju systemu POSIX) może ustanowić port programu diagnostycznego (ang. *debugger*) oraz port wyjątku dla każdego z utworzonych przez siebie procedur. Jeśli zarejestrowano port programu diagnostycznego, to procedura obsługi wyjątku przesyła wyjątek programowi diagnostycznemu. Jeśli portu programu diagnostycznego nie odnaleziono lub nie obsługuje on danego wyjątku, to ekspedytor próbuje znaleźć odpowiednią procedurę obsługi. W przypadku jej nieznalezienia program diagnostyczny jest wywoływany ponownie, aby mógł zasygnalizować błąd. Jeśli program diagnostyczny nie jest wykonywany, to następuje wysłanie komunikatu do procesowego portu wyjątku, co daje podsystemowi środowiskowemu szansę przetłumaczenia wyjątku. Na przykład środowisko systemu POSIX tłumaczy wyjątek systemu NT na sygnał systemu POSIX, zanim wyśle go do wątku, który zdarzenie wyjątkowe spowodował. Jeśli nic już nie działa, to jądro po prostu kończy proces z wąkiem, który spowodował wyjątek.

Koordynator przerwań w jądrze obsługuje przerwania, wywołując procedury obsługi przerwań (np. moduły obsługi urządzeń) lub wewnętrzne procedury jądra. Przerwanie jest reprezentowane przez obiekt przerwania, który zawiera wszystkie informacje potrzebne do obsługi przerwania. Posługiwanie się obiektem przerwań ułatwia kojarzenie procedur obsługi przerwań z przerwaniami, eliminując konieczność bezpośredniego dostępu do sprzętu powodującego przerwanie.

Różne architektury procesorów, jak na przykład Intel lub DEC Alpha, mają różne rodzaje i liczbę przerwań. Ze względu na przenośność koordynator przerwań odwzorowuje przerwania sprzętowe w standardowy zbiór. Przerwania mają priorytety i są obsługiwane w ich kolejności. W systemie NT są 32 poziomy przerwań (IRQL^{*}). Osiem poziomów jest zarezerwowanych do użytku jądra; na pozostałych 24 poziomach są rozlokowane przerwania sprzętowe warstwy HAL. Przerwania systemu NT są zdefiniowane na rys. 23.2.

Aby powiązać przerwanie dowolnego poziomu z procedurą obsługową, jądro korzysta z tablicy rozdzielczej przerwań (ang. *interrupt dispatch table*). W komputerze wieloprocesorowym system NT przechowuje oddzielną tablicę rozdzielczą przerwań dla każdego procesora, a poziom przerwań (IRQL) może być w celu ich maskowania ustalany niezależnie dla każdego procesora. Przerwania występujące na poziomie równym lub niższym niż poziom IRQL danego procesora będą blokowane aż do obniżenia poziomu IRQL przez wątek z poziomu jądra. System NT wykorzystuje to do stosowania przerwań programowych w celu wykonywania swoich funkcji. Jądro na przykład używa przerwań programowych, gdy należy rozpocząć ekspedycję wątku, do obsługi czasomierzy i umożliwiania działań asynchronicznych.

* IRQL – z ang. *interrupt request level* – poziom zgłoszenia przerwania. – Przyp. tłum.

Poziomy przerwań	Rodzaje przerwań
31	błąd procesora lub szyny
30	awaria zasilania
29	powiadomienie międzyprocesorowe (żądanie uaktywnienia innego procesora, np. wyekspediowanie do niego procesu lub aktualizacja TLB)
28	zegar (używany do pomiaru czasu)
12-27	typowe przerwania sprzętowe komputera PC
4-11	ekspediowanie i opóźnione (przez jądro) wywołanie procedury
3	program diagnostyczny
0-2	przerwania programowe (używane przez moduły sterujące urządzeń)

Rys. 23.2 Przerwania systemu NT

Nadzór nad przełączaniem wątków jądro sprawuje za pomocą przerwań ekspedytorskich. Podczas swojej pracy jądro podnosi poziom IRQL procesora powyżej poziomu ekspedytora. Kiedy jądro uzna, że należy podjąć wykonywanie wątku, wtedy generuje przerwanie ekspedytorskie, które jednak będzie blokowane do chwili zakończenia działania przez jądro i obniżenia przez nie poziomu przerwań IRQL. Wówczas może nastąpić obsługa przerwania ekspedytorskiego, dzięki czemu ekspedytor wybiera wątek do pracy.

Gdy jądro zadecyduje, że jakaś funkcja systemowa powinna być wykonana, lecz niekoniecznie natychmiast, wówczas ustawia w kolejce obiekt *opóźnionego wywołania procedury* (ang. *deferred procedure call* – DPC) zawierający adres tej funkcji i wytwarza przerwanie DPC. Gdy poziom blokowania przerwań procesora (IRQL) stanie się wystarczająco niski, wówczas obiekty DPC zostaną wykonane. Poziom IRQL obiektów DPC jest zazwyczaj wyższy niż poziom wątków użytkownika, więc wywołania DPC będą przerywać działanie wątków użytkownika. Aby uniknąć problemów, zakłada się, że wywołania DPC będą dość proste. Nie wolno im zmieniać pamięci wątku, tworzyć lub zamawiać obiektów albo na nie czekać, wywoływać innych usług systemu lub generować przerwań spowodowanych brakiem stron.

Mechanizm *asynchronicznego wywołania procedury* (ang. *asynchronous procedure call* – APC) jest podobny do mechanizmu DPC, lecz ogólniejszy. Umożliwia on wątkom aranżowanie przyszłych, niespodziewanych wywołań procedur. Wiele na przykład usług systemowych przyjmuje jako parametr procedur w trybie użytkownika. Zamiast powodować synchroniczne wywołanie systemowe, które zablokuje wątek aż do swojego zakończenia, wątek użytkownika może wykonać asynchroniczne wywołanie systemowe, dostarczając wywołania APC, po którym będzie kontynuował działanie. Po zakończeniu usługi

systemowej wątek użytkownika zostanie przerwany, aby mogło nastąpić samorzutnie wykonanie procedury APC. Wywołanie APC może być zaliczone do kolejki w wątku systemowym lub użytkowym, chociaż wywołanie APC w trybie użytkownika będzie wykonane tylko wtedy, gdy sam wątek zadeklarował, że jest podatny na alarmy. Wywołanie APC ma mocniejsze właściwości niż DPC, ponieważ może ono zamawiać obiekty i czekać na nie, powodować braki stron i korzystać z usług systemowych. Z uwagi na to, że wywołania APC przebiegają w przestrzeni adresowej docelowego wątku, egzekutor systemu NT posługuje się nimi często przy wykonywaniu operacji wejścia-wyjścia.

System NT może pracować na maszynach o symetrycznej architekturze wieloprocesorowej, toteż jądro musi pilnować, aby dwa jego wątki nie zmieniały wspólnych struktur danych w tym samym czasie. Do uzyskiwania wzajemnego wykluczania w wieloprocesorze jądro używa wirujących blokad organizowanych w pamięci operacyjnej. W procesorze wykonującym wątek pozostający w stanie wirującej blokady zamiera wszelkie inne działanie, wątek taki jest niewywłączalny, więc może zakończyć blokadę, gdy tylko stanie się to możliwe*.

Przerwanie powodowane awarią zasilania – drugie pod względem wielkości priorytetu – powiadamia system operacyjny o każdej utracie zasilania. Obiekt informujący o zasilaniu umożliwia modułowi sterującemu urządzenia zarejestrowanie procedury, której wywołanie nastąpi po przywróceniu zasilania i zapewnia powrót urządzeń do właściwego stanu. W systemach zaopatrzonych w awaryjne zasilanie baterijne znajduje zastosowanie obiekt stanu zasilania określający jakość zasilania. Przez sprawdzanie obiektu stanu zasilania moduł sterujący rozstrzyga, czy awaria zasilania wystąpiła przed czy po rozpoczęciu przez niego krytycznej operacji. Jeśli moduł sterujący ustali, że nie było awarii zasilania, to podnosi poziom IRQL swojego procesora do stopnia awarii zasilania, wykonuje daną operację i odtwarza poprzedni stan maskowania przerwań IRQL. Taki ciąg działań blokuje przerwanie sygnalizujące awarię zasilania do chwili zakończenia krytycznej operacji.

23.3.3 Egzekutor

Egzekutor systemu NT wykonyuje zbiór usług, z których mogą korzystać wszystkie podsystemy środowiskowe. Usługi te można podzielić na następujące grupy: zarządcę obiektów, zarządcę pamięci wirtualnej, zarządcę procesów, udogodnienie lokalnych wywołań procedur, zarządcę wejścia-wyjścia oraz monitor bezpieczeństwa odwołań.

* Wirująca blokada jest po prostu dynamiczną pętlą oczekiwania na zderzenie, organizowaną w wątku. – Przyp. tłum.

23.3.3.1 Zarządcy obiektów

Obiektowy system NT stosuje obiekty we wszystkich swoich usługach i fragmentach. Przykładami obiektów są obiekty katalogowe, obiekty dowiązań symbolicznych, obiekty semaforów, zdarzeń, procesów, wątków, portów i plików. Zadaniem zarządcy obiektów jest nadzorowanie użytkowania wszystkich obiektów. Jeśli wątek chce skorzystać z obiektu, to wywołuje metodę **open** zarządcy obiektów w celu otrzymania *uchwytu* (ang. *handle*) do obiektu. Uchwyty są standaryzowanymi interfejsami do obiektów wszystkich rodzajów. Podobnie jak uchwyt do pliku, uchwyt do obiektu jest niepowtarzalną w procesie liczbą całkowitą, umożliwiającą dostęp do zasobu systemowego i wykonywanie na nim działań.

Ponieważ zarządcy obiektów jest jedyną częścią systemu będącą w stanie wytworzyć uchwyt do obiektu, z naturalnych przyczyn jest więc odpowiednim miejscem do dbałości o bezpieczeństwo. Gdy proces na przykład próbuje otworzyć obiekt, zarządcy obiektów sprawdza, czy proces ma prawo dostępu do tego obiektu. Zarządcy obiektów jest również w stanie narzucać pewne limity, takie jak maksymalna ilość pamięci, którą wolno przydzielić procesowi.

Zarządcy obiektów może też pilnować, które procesy używają których obiektów. Każdy nagłówek obiektu zawiera licznik procesów dysponujących uchwytem do danego obiektu. Kiedy licznik ten się wyzeruje, następuje usunięcie obiektu z przestrzeni nazw, jeżeli jego nazwa była tymczasowa. Ponieważ sam system NT do kontaktowania się z obiektami często używa wskaźników (zamiast uchwytów), więc zarządcy obiektów utrzymuje również licznik odwołań zwiększanego wówczas, gdy dostęp do obiektu otrzymuje system NT, i zmniejszany wtedy, kiedy obiekt przestaje być potrzebny. Obiekt tymczasowy, którego licznik odwołań spada do zera, zostaje usunięty z pamięci. Obiekty trwałe reprezentują elementy fizyczne, takie jak napędy dysków, więc nie są usuwane mimo zerowych wartości liczników odwołań i utworzonych uchwytów.

Działania na obiektach odbywają się za pomocą standardowego zbioru metod, tj. operacji: **create**, **open**, **close**, **delete**, **query name**, **parse** i **security**. Trzy ostatnie wymagają wyjaśnienia:

- operacja **query name** jest wywoływaną wówczas, gdy wątek ma uchwyt do obiektu, lecz chce poznać jego nazwę;
- zarządcy obiektów posługują się operacją **parse** w celu odnalezienia obiektu na podstawie jego nazwy;
- operację **security** wywołuje się wtedy, kiedy proces otwiera obiekt lub zmienia tryb jego ochrony.

23.3.3.2 Obiekty nazewnicze

Egzekutor systemu NT umożliwia nadawanie dowolnego obiektu. Przestrzeń nazw jest globalna, zatem jeden proces może utworzyć obiekt z nazwą, a inny proces może utworzyć uchwyt do takiego obiektu i dzielić obiekt z pierwszym procesem. Proces otwierający nazwany obiekt może poprosić o poszukiwanie go z rozróżnianiem małych i wielkich liter, bądź bez rozróżniania.

Nazwa może być trwała lub tymczasowa. Nazwa trwała odnosi się do obiektu, który istnieje nawet wtedy, kiedy nie korzysta z niego żaden proces. Nazwa tymczasowa istnieje tylko przez okres, w którym jakiś proces utrzymuje uchwyt do obiektu.

Chociaż przestrzeń nazw nie jest bezpośrednio widoczna w sieci, zarządcą obiektów pomaga sobie w dotarciu do obiektu w innym systemie za pomocą operacji *parse*. Jeśli proces próbuje otworzyć obiekt rezydujący w odległym komputerze, to zarządcą obiektów wywołuje metodę *rozbioru*, która z kolei wywołuje sieciową procedurę przadresowującą w celu odnalezienia obiektu.

Nazwy obiektów są strukturalne niczym ścieżki dostępu w systemach MS-DOS i UNIX. Katalogi są reprezentowane za pomocą obiektów *katalogowych*, które zawierają nazwy wszystkich obiektów w danym katalogu. Przestrzeń nazw obiektów może się rozrastać przez dodawanie *domen obiektów* (ang. *object domains*), które są zamkniętymi w sobie zbiorami obiektów. Przykładami domen obiektów są dyski elastyczne i dyski twarde. Łatwo zauważyc, że po dodaniu dyskietki do systemu przestrzeń nazw rozszerza się – dyskietka ma własną przestrzeń nazw, którą wszechepia się do istniejącej przestrzeni nazw.

Uniksowe systemy plików mają *dowiązania symboliczne* (ang. *symbolic links*), wskutek czego do tego samego pliku może się odnosić wiele skrótów lub synonimów. System NT w podobny sposób realizuje *obiekt dowiązań symbolicznych*. Jednym ze sposobów zastosowania dowiązań symbolicznych w systemie NT jest odwzorowanie nazw napędów dysków na standardowe literowe oznaczenia systemu MS-DOS. Litery dysków są po prostu dowiązaniem symbolicznym, które można odwzorowywać ponownie w celu spełnienia preferencji użytkownika.

Proces otrzymuje uchwyt do obiektu przez jego utworzenie, otwarcie istniejącego obiektu, uzyskanie drugiego uchwytu od innego procesu lub przez dziedziczenie po procesie macierzystym. Przypomina to sposób, w jaki proces systemu UNIX może dostać deskryptor pliku. Wszystkie uchwyty pamięta się w *tablicy obiektów* procesu. Wpis w tablicy obiektów zawiera prawa dostępu do obiektu i wskazuje, czy uchwyt powinien być dziedziczony przez procesy pochodne. Gdy proces kończy działanie, system NT automatycznie likwiduje wszystkie utrzymywane przez niego uchwyty.

Podczas sprawdzania tożsamości użytkownika przez proces rejestracyjny (ang. *log in process*) procesowi użytkownika przypisuje się obiekt żetonu dostępu. Żeton dostępu zawiera informacje takie, jak identyfikatory bezpieczeństwa, identyfikatory grup, przywileje, określenie grupy podstawowej i zastępczą listę kontroli dostępu. Atrybuty te przesądzają o rodzaju usług i obiektów, którymi można będzie się posługiwać.

Każdy obiekt systemu NT jest chroniony przez *listę kontroli dostępu* (ang. *access-control list*). Lista ta zawiera identyfikatory bezpieczeństwa i zbiór praw dostępu udzielanych dla każdego obiektu. Jeśli proces usiłuje skontaktować się z obiektem, to system porównuje identyfikatory bezpieczeństwa procesowego żetonu dostępu z występującymi na liście kontroli dostępu obiektu i w ten sposób rozstrzyga, czy na dostęp powinno się zezwolić. Sprawdzenia tego dokonuje się tylko podczas otwierania obiektu, więc wewnętrzne usługi systemu NT, które zamiast ubiegania się o uchwyty do obiektów stosują wskaźniki, omijają sprawdzanie poprawności dostępu.

Listę kontroli dostępu do obiektu z zasady określa jego twórca. Jeżeli nie dostarczono jawnic żadnych informacji na ten temat, to lista może być odziedziczona po twórcy obiektu lub może nastąpić zastępce jej określenie według obiektu żetonu dostępu należącego do użytkownika.

Jedno z pól w żetonie dostępu nadzoruje sposób doglądania obiektu. Operacje doglądane są odnotowywane w systemowym dzienniku działań nadzorowanych, wraz z identyfikacją użytkownika. Operacje związane z polem doglądania mają prawo wglądu do tego dziennika w celu wykrywania prób włamania do systemu lub usiłowań dostępu do obiektów chronionych.

23.3.3 Zarządcia pamięci wirtualnej

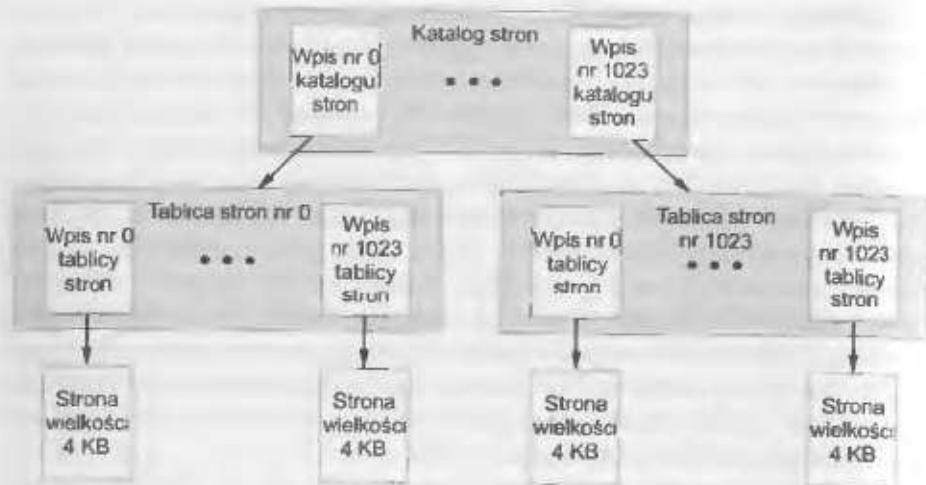
Część egzekutora NT zajmująca się pamięcią wirtualną nosi nazwę *zarządcy pamięci wirtualnej* (zarządcy VM). W projekcie zarządcy VM przyjęto, że odwzorowania między pamięcią fizyczną a wirtualną będą obsługiwane przez sprzęt, który zapewni też stronicowanie, przezroczystą zgodność zawartości pamięci podręcznych w systemie wieloprocesorowym umożliwi odwzorowywanie na tę samą ramkę wielu wpisów z tablicy stron. Zarządcy VM w systemie NT korzysta ze stronicowanego schematu administrowania pamięcią, używając stron o wielkości 4 KB. Strony danych przydzielone do procesu, lecz nie występujące w pamięci fizycznej są przechowywane w pliku stronicowania na dysku.

Zarządcy VM stosuje adresy 32-bitowe, więc każdy proces ma do dyspozycji przestrzeń adresową wielkości 4 GB. Górnego dwa gigabajty są identyczne dla wszystkich procesów i są używane przez system NT w trybie jądra. Dolne dwa gigabajty różnią się w każdym procesie i są dostępne zarówno dla wątków działających w trybie użytkownika, jak i w trybie jądra.

Zarządcą VM systemu NT przydziela pamięć dwustopniowo. W pierwszym kroku następuje *zarezerwowanie* części przestrzeni adresowej procesu. W drugim – dochodzi do *zatwierdzenia* przydziału przez wygospodarowanie miejsca w pliku stronicowania systemu NT. System NT może ograniczać wielkość przestrzeni pliku stronicowania zużywaną przez proces, narzucając rozmiar zatwierzonej pamięci. Proces może anulować zatwierdzony przydział pamięci, której już nie potrzebuje, zwalniając swój obszar stronicowania. Ponieważ pamięć jest reprezentowana przez obiekty, więc gdy pewien proces (macierzysty) tworzy inny proces (potomny), to rodzic może skorzystać z możliwości dostępu do pamięci swojego dziecka. W ten sposób podsystemy środowiskowe mogą zarządzać pamięcią swoich klientów. Ze względu na wydajność zarządcy VM pozwala uprzywilejowanemu procesowi na blokowanie wybranych stron pamięci fizycznej, co zapewnia, że nie będą one podlegały usuwaniu do pliku stronicowania.

Dwa procesy mogą dzielić pamięć, uzyskując uchwyty do tego samego obiektu pamięci, lecz metoda taka może być niewydajna, gdyż cały obszar pamięci takiego obiektu musi być zatwierdzony przed skorzystaniem z niego przez którykolwiek z procesów. Do reprezentowania bloku pamięci wspólnej system NT wprowadza możliwość pod nazwą *obiektu sekcji* (ang. *section object*). Po otrzymaniu uchwytu do obiektu sekcji procesowi wolno odwołać tylko potrzebny fragment pamięci. Fragment ten jest nazywany *widokiem* (ang. *view*). Mechanizm widoków umożliwia także procesowi dostęp do obiektu, który jest za duży, aby zmieścić się w przydzielonym procesowi liście pliku stronicowania. System może korzystać z widoków do przejedzienia przez przestrzeń adresową obiektu kawałek po kawałku.

Proces może na wiele sposobów nadzorować użytkowanie pamięci dziedziny reprezentowanej przez obiekt sekcji. Maksymalny rozmiar sekcji może być ograniczony. Sekcja może być składowana na dysku w pliku stronicowania lub w zwykłym pliku (nazywanym *plikiem odwzorowanym w pamięci*). Sekcja może mieć bazę, tzn. występować we wszystkich korzystających z niej procesach pod tym samym adresem wirtualnym. Strony pamięci sekcji mogą być chronione jako tylko do czytania, czytania i pisania, tylko do wykonywania, mogą być też strzeżone i kopowane przy zapisie. Dostęp do strony strzeżonej powoduje powstanie wyjątku. W trakcie obsługi tego wyjątku można na przykład sprawdzić, czy program wskutek błędu iteracji nie wykroczył poza tablicę. Mechanizm kopowania przy zapisie pozwala zarządcy VM oszczędzać pamięć. Gdy dwa procesy chcą niezależnych kopii obiektu, wówczas zarządcy VM umieszcza w pamięci fizycznej tylko jedną kopię do wspólnego użytku, nadając jednakże temu obszarowi cechę kopowania przy zapisie. Jeśli któryś z procesów sprobuje zmienić dane na stronie kopowanej przy zapisie, to na użytkownika procesu zarządcy VM najpierw utworzy prywatną kopię tej strony.



Rys. 23.3 Schemat pamięci wirtualnej

W tłumaczeniu adresu wirtualnego w systemie NT bierze udział kilka struktur danych. Każdy proces ma *katalog stron* mieszczący 1024 pozycje po 4 bajty. Katalog stron jest na ogół prywatny, może być jednakże dzielony przez procesy, jeśli wymaga tego sytuacja. Każda pozycja katalogu stron wskazuje na *tablicę stron* zawierającą również 1024 pozycje po 4 bajty (ang. *page-table entries* – PTE). Każdy wpis PTE wskazuje na 4-kilobajtową ramkę strony w pamięci fizycznej. Łączna wielkość wszystkich tablic stron procesu wynosi 4 MB, więc zarządcą VM będzie w razie potrzeby usuwał je na dysk. Schemat tej struktury jest przedstawiony na rys. 23.3.

Wszystkie wartości z przedziału 0-1023 można wyrazić za pomocą 10-bitowej liczby całkowitej. Zatem 10-bitowa liczba całkowita może posłużyć do wyboru wpisu w katalogu stron lub w tablicy stron. Wykorzystuje się to podczas tłumaczenia wskaznika adresu wirtualnego na adres bajta w pamięci fizycznej. Trzydziestodwubitowy adres pamięci wirtualnej rozbija się na trzy liczby całkowite, jak widać na rys. 23.4. Pierwszych 10 bitów adresu wirtualnego używa się jako indeksu do tablicy stron. Adres ten wskazuje jedną pozycję



Rys. 23.4 Tłumoczenie adresu wirtualnego na adres fizyczny

katalogu stron, która zawiera wskaźnik do tablicy stron. Następujących 10 bitów adresu wirtualnego służy do wybrania wpisu PTE z tablicy stron. Jego wartość wskazuje ramkę strony w pamięci fizycznej. Pozostałych 12 bitów adresu wirtualnego wskazuje konkretny bajt danej ramki. Wskaźnik do tego bajta w pamięci fizycznej tworzy się, łącząc 20 bitów wpisu z tablicy stron (PTE) z 12 młodszymi bitami adresu wirtualnego. Zatem 32-bitowy wpis PTE ma 12 bitów nadmiarowych – używa się ich do opisu strony. Pierwszych 5 bitów określa tryb ochrony strony, w rodzaju **PAGE_READONLY** lub **PAGE_READWRITE**. Następne 4 bity określają, który zbiór stronicowania jest stosowany do składowania tej strony pamięci. Ostatnie 3 bity określają stan strony w pamięci. Ogólniejsze informacje dotyczące schematu stronicowania zawiera p. 8.5.

Strona może się znajdować w jednym z sześciu stanów: ważnym, wyzerowanym, wolnym, pogotowia, zmienionym i złym. Strona *ważna* (ang. *valid*) jest używana przez aktywny proces. Strona *wolna* (ang. *free*) nie ma odniesienia w żadnej pozycji tablicy stron (PTE). Strona *wyzerowana* jest stroną wolną, którą zapelniono zerami i która jest gotowa do natychmiastowego użycia. Strona *w pogotowiu* (ang. *standby*) została usunięta z roboczego zbioru stron procesu. Stronę *zmienioną* zapisano, lecz jeszcze nie przekopiowano na dysk. Ostatnim rodzajem strony jest *strona zła* (ang. *bad page*) – jest ona bezużyteczna z powodu wykrycia błędu sprzętowego.

Budowa wpisu tablicy stron jest pokazana na rys. 23.5. Wpis taki zawiera 5 bitów ochrony strony, 20 bitów adresu ramki strony, 4 bity do wyboru pliku stronicowania i 3 bity opisujące stan strony. Jeśli strony nie ma w pamięci, to na 20 bitach ramki strony przechowuje się odległość strony w pliku stronicowania. Ponieważ kod wykonywalny oraz pliki odwzorowane w pamięci mają już swoje kopie na dysku, więc nie wymagają miejsca w pliku stronicowania. Jeśli jednej z tych stron nie ma w pamięci fizycznej, to wpis w tablicy stron ma budowę następującą: najstarszego bitu używa się do określenia ochrony strony, następnych 28 bitów służy jako indeks pliku w strukturze danych systemu i wskaźnik odległości strony w tym pliku, a ostatnie 3 bity określają stan strony.

31

0



Rys. 23.5 Standardowy wpis w tablicy stron.

Jeśli każdy proces ma indywidualnie zestawione tablice stron, to dzielenie stron między procesami jest rzeczą trudną, gdyż każdy proces ma własny wpis PTE dla danej ramki. Gdy wystąpi brak strony dzielonej w pamięci fizycznej, wówczas adres fizyczny powinien być zapamiętany we wpisie tablicy stron każdego procesu korzystającego z tej strony. Bitы ochrony oraz bitы stanu strony w tych wpisach powinny być spójnie uaktualnianie. Aby uniknąć tych problemów, system NT ucieka się do działań pośrednich. Dla każdej dzielonej strony proces ma wpis PTE wskazujący *prototyp wpisu tablicy stron*, a nie ramkę strony. Prototyp wpisu PTE zawiera adres ramki strony oraz bitы ochrony i stanu. W ten sposób zarządcą VM ma do uaktualnienia tylko jeden prototyp wpisu PTE zamiast po jednym wpisie w każdym procesie korzystającym ze strony.

Zarządcą pamięci wirtualnej odnotowuje informacje o wszystkich stronach pamięci fizycznej w *bazie ramek stron*. Ramka każdej strony ma w niej jeden wpis. Wpis ów pokazuje na pozycję w tablicy stron (PTE), a ta wskazuje na ramkę strony, więc zarządcą VM może panować nad stanem strony. Ramki stron są powiązane w listy (np. stron wyzerowanych oraz stron wolnych).

Jeśli pojawi się brak strony, to zarządcą VM wprowadza brakującą stronę w pierwszą ramkę na liście stron wolnych. Nie poprzestaje jednak na tym. Badania wykazują, że odwołania do pamięci w wątku mają cechę *lokalności* (ang. *locality*): jeśli jakas strona jest w użyciu, to prawdopodobnie strony przyległe będą użyte w najbliższej przyszłości. (Wyobraźmy sobie ciąg operacji na tablicy lub pobieranie kolejnych rozkazów tworzących kod wykonywalny wątku). Ze względu na lokalność zarządcą VM spostrzegający brak strony zakłada, że brak ten dotyczy również kilku sąsiednich stron. Takie postępowanie zmierza do zmniejszenia ogólnej liczby braków stron.Więcej informacji na temat lokalności zawiera p. 9.7.1.

Jeśli na liście wolnych stron brakuje ramek, to system NT stosuje w odniesieniu do każdego procesu algorytm zastępowania FIFO, aby odbierać strony procesom, które mają ich więcej, niż wynosi wielkość ich minimalnego zbioru roboczego. System NT nadzoruje braki stron każdego procesu, którego zbiór roboczy ma wielkość minimalną, odpowiednio korygując wielkość zbioru roboczego. I tak na początku proces w systemie NT otrzymuje zastępco 30 stron w zbiorze roboczym. System NT okresowo sprawdza tę wielkość, podkradając procesowi ważną stronę. Jeśli proces kontynuuje działanie, nie generując braku strony, to wielkość jego zbioru roboczego zmniejsza się o 1, a stronę dokłada się do listy stron wolnych.

23.3.3.4 Zarządcą procesów

Zarządcą procesów systemu NT dostarcza usług tworzenia, usuwania i użytkowania wątków i procesów. Nie ma on żadnych danych o związkach między

rodzicami a ich potomkami ani o hierarchii procesów – te szczegóły leżą w gestii odpowiedniego podsystemu środowiskowego będącego właścicielem procesu.

Oto przykład tworzenia procesu w środowisku podsystemu Win32. Kiedy aplikacja systemu Win32 wywołuje funkcję **CreateProcess**, do podsystemu Win32 zostaje wysłany komunikat wywołujący zarządcę procesów w celu utworzenia procesu. Zarządcą procesu wywołuje zarządcę obiektów w celu utworzenia obiektu procesu, a następnie zwraca uchwyt do obiektu systemowym Win32. System Win32 jeszcze raz wywołuje zarządcę procesu, aby ten utworzył wątek danego procesu. Na koniec system Win32 przekazuje (do aplikacji) uchwyty do nowego procesu i wątku.

23.3.3.5 Udogodnienie wywoływania procedur lokalnych

Udogodnienia wywoływania procedur lokalnych (LPC) używa się do przekazywania zamówień i wyników między procesami klienta i serwera na tej samej maszynie. W szczególności korzysta się z niego przy zamawianiu usług różnych podsystemów NT. Pod wieloma względami przypomina ono mechanizmy zdalnego wywołania procedury (RPC) stosowane w wielu systemach operacyjnych podczas przetwarzania rozproszonego w sieciach, lecz mechanizm LPC jest optymalizowany do użytku w jednym systemie NT.

LPC jest mechanizmem przekazywania komunikatów. Proces serwera upowszechnia globalnie widoczny obiekt portu łączającego. Klient, który potrzebuje usługi ze strony podsystemu, zaopatruje się w uchwyt do portu łączającego w tym podsystemie, po czym wysyla do portu prośbę o połączenie. Serwer tworzy kanał i zwraca klientowi uchwyt. Kanał składa się z pary prywatnych portów komunikacyjnych: jeden dla komunikatów od klienta do serwera, a drugi dla komunikatów od serwera do klienta. Kanały komunikacyjne dostarczają mechanizmu przywołania, więc klient i serwer mogą przyjmować zamówienia, podeczas gdy zazwyczaj oczekiwali by odpowiedzi.

Po utworzeniu kanału LPC należy określić jedną z trzech technik przekazywania komunikatów. Pierwsza jest odpowiednia dla małych komunikatów (do 256 B). W tym przypadku w charakterze pamięci pośredniej stosuje się portową kolejkę komunikatów, a komunikaty są kopiowane z jednego procesu do drugiego. Technika drugiego rodzaju nadaje się do dłuższych komunikatów. W tym przypadku w kanale jest tworzony obiekt sekcji pamięci dzielonej. Komunikaty przesypane przez portową kolejkę komunikatów zawierają wskaźnik i informację o rozmiarze odnoszące się do obiektu sekcji. Unika się w ten sposób konieczności kopowania dużych komunikatów. Nadawca umieszcza dane we wspólnej sekcji, gdzie odbiorca może je oglądać bezpośrednio.

Trzecia metoda przekazywania komunikatów LPC, nazywana *szymbkimi wywołaniami LPC*, jest stosowana przez części podsystemu Win32 odpowie-

dzialne za wyświetlanie graficzne. Jeśli klient poprosi o połączenie w trybie szybkich wywołań LPC, to serwer konstruuje trzy obiekty: wątek usługowy przeznaczony wyłącznie do obsługi zamówień, 64-kilobajtowy obiekt sekcji pamięci oraz obiekt pary zdarzeń. *Obiekt pary zdarzeń* jest obiektem synchronizacji używanym przez podsystem Win32 do powiadamiania o tym, że wątek klienta przekopiował komunikat do serwera Win32 lub na odwrot. Komunikaty LPC są przekazywane w obiekcie sekcji, co eliminuje ich kopowanie, gdyż sekcja ta jest obszarem pamięci wspólnej. Dzięki synchronizacji za pomocą obiektu pary zdarzeń eliminuje się koszt używania obiektu portu do przekazywania komunikatów ze wskaźnikami i długościami. Dedykowany wątek serwera eliminuje koszty określania, który wątek klienta wywołuje serwer, gdyż osobny serwer odpowiada każdemu wątkowi klienta. Jądro daje również dedykowanym wątkom usługowym preferencje dotyczące planowania przydziału procesora, co w jeszcze większym stopniu powiększa wydajność. Wadą metody szybkich wywołań LPC jest to, że zużywają one więcej zasobów niż każda z dwóch poprzednich technik, toteż podsystem Win32 korzysta z szybkich wywołań LPC tylko w przypadku zarządcy okien i interfejsów urządzeń graficznych.

23.3.3.6 Zarządcza wejścia-wyjścia

Zarządcza wejścia-wyjścia (zarządcza IO) odpowiada za systemy plików, administrowanie pamięcią podręczną oraz za moduły sterujące urządzeń i sieci. Pamięta on, które z instalowalnych systemów plików są załadowane, a także zarządza buforami zamówień wejścia-wyjścia. Współpracuje z zarządcą pamięci wirtualnej (zarządcą VM) w celu odwzorowywania w pamięci wyników plikowych operacji wejścia-wyjścia i nadzoruje pracę zarządcy pamięci podręcznej systemu NT, który zajmuje się podręcznymi przechowaniami na użytkę całego systemu wejścia-wyjścia. Zarządcza IO umożliwia stosowanie zarówno operacji synchronicznych, jak i niesynchronicznych, odlicza czas dla modułów sterujących i ma mechanizm pozwalający na wywoływanie jednego modułu sterującego przez drugi.

Zarządcza wejścia-wyjścia zamienia otrzymywane przez siebie zamówienia na standardową postać, nazywaną *pakietem zamówienia wejścia-wyjścia* (ang. *I/O request packet* – IRP), po czym kieruje pakiety IRP do przetwarzania do właściwych modułów sterujących. Po zakończeniu operacji zarządcza IO otrzymuje pakiet IRP od modułu sterującego, który wykonywał działania jako ostatni, i kończy zamówienie.

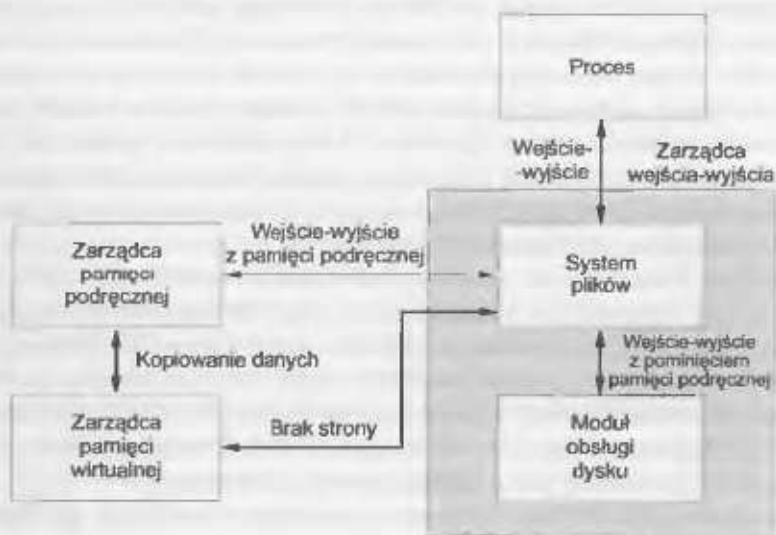
W wielu systemach operacyjnych przechowywanie podręczne należy do obowiązków systemu plików. W systemie NT zastosowano zamiast tego centralizowaną pamięć podręczną. Zarządcza pamięci podręcznej dostarcza usług przechowywania podręcznego wszystkim składowym pozostającym pod nadzo-

rem zarządcy wejścia-wyjścia i działa w bliskim związku z zarządcą pamięci wirtualnej. Wielkość pamięci podręcznej zmienia się dynamicznie, stosownie do tego, ile wolnej pamięci jest dostępnej w systemie. Przypomnijmy, że górne 2 GB przestrzeni adresowej procesu tworzą obszar systemowy, który jest taki sam we wszystkich procesach. Zarządcy VM przydziela nie więcej niż połowę tego obszaru jako systemową pamięć podręczną. Zarządcy pamięci podręcznej odwzorowują pliki w tej przestrzeni adresowej i korzystają ze zdolności zarządcy VM do obsługiwanego plikowego wejścia-wyjścia.

Pamięć podręczna jest podzielona na bloki wielkości 256 KB. Każdy blok może przechowywać widok pliku (tj. jego obszar odwzorowany w pamięci). Każdy blok pamięci podręcznej jest opisany przez *blok kontrolny adresu wirtualnego* (ang. *virtual-address control block* – VACB), w którym pamięta się adres wirtualny i odległość widoku w pliku, a także liczbę procesów korzystających z danego widoku. Bloki VACB znajdują się w osobnej tablicy administrowanej przez zarządcę pamięci podręcznej.

Gdy zarządcy IO otrzyma operację czytania z poziomu użytkownika, wówczas wysyła pakiet IRP do zarządcy pamięci podręcznej (chyba że w zamówieniu zaznaczono, że chodzi o czytanie z pominięciem pamięci podręcznej). Zarządcy pamięci podręcznej oblicza, który wpis tablicy indeksów bloków VACB pliku odpowiada bajtowej odległości zamówienia. Wpis ten albo wskazuje na widok w pamięci podręcznej, albo jest pusty. Jeżeli wpis jest pusty, to zarządcy pamięci podręcznej przydziela blok pamięci podręcznej (i odpowiedni wpis w tablicy VACB) i odwzorowuje w nim widok fragmentu pliku. Zarządcy pamięci podręcznej próbują następnie skopiować dane z odwzorowanego pliku do bufora w miejscu wywołania. Jeżeli kopiowanie się uda, to operacja się kończy. Jeżeli nie, to przyczyną jest brak strony, co powoduje, że zarządcy pamięci wirtualnej przesyła do zarządcy wejścia-wyjścia zamówienie na wykonanie czytania z pominięciem pamięci podręcznej. Zarządcy IO prosi odpowiedni moduł sterujący urządzeniem o przeczytanie danych i zwraca dane zarządcy VM, który wprowadza je do pamięci podręcznej. Dane znajdują się teraz w pamięci podręcznej, więc następuje ich przekopiowanie do bufora umieszczonego tam, skąd pochodziło zamówienie i operacja ulega zakończeniu. Przebieg tej operacji jest pokazany na rys. 23.6.

Operacja czytania z poziomu jądra wygląda podobnie, z tym że dane mogą być osiągane wprost w pamięci podręcznej i nie muszą być kopiowane do bufora w przestrzeni użytkownika. Aby użyć *metadanych* (ang. *metadata*) systemu plików, czyli struktur danych opisujących system plików, jądro korzysta z interfejsu odwzorowań organizowanego przez zarządcę pamięci podręcznej. Aby zmienić metadane, system plików korzysta z interfejsu spinającego zarządcy pamięci podręcznej. *Przypięcie* (ang. *pinning*) strony powoduje jej zablokowanie w ramce fizycznej pamięci, tak że zarządcy VM nie moze jej



Rys. 23.6 Plikowe wejście-wyjście

przenieść ani wyrzucić na dysk. Po zaktualizowaniu metadanych system plików prosi zarządcę pamięci podręcznej o odpięcie strony. Ponieważ strona została zmieniona, więc oznacza się ją jako zabrudzoną i zarządcą VM sporządzi jej kopię na dysku.

W celu polepszenia wydajności zarządcy pamięci podręcznej utrzymuje niewielką historię operacji czytania i pisania i próbuje przewidzieć przyszłe zamówienia. Jeśli uda mu się znaleźć w ostatnich trzech zamówieniach wzorzec w rodzaju sekwencyjnego dostępu w przód lub wstecz, to może załadować wstępnie dane do pamięci podręcznej, zanim nastąpi przedłożenie kolejnego zamówienia przez aplikację. Wówczas aplikacja może zastać swoje dane od razu w pamięci podręcznej i nie będzie musiała czekać na dyskową operację wejścia-wyjścia. Funkcje `OpenFile` i `CreateFile` interfejsu API Win32 mogą zawierać znacznik `FILE_FLAG_SEQUENTIAL_SCAN`, który jest wskazówką dla zarządcy pamięci podręcznej, aby próbował załadować wstępnie 192 KB kolejnych danych w stosunku do określonych w zamówieniach zgłaszanych przez wątek. System NT na ogół wykonuje operacje porcjami po 64 KB lub po 16 stron, zatem takie czytanie w przód jest trzykrotnie większe niż normalnie.

Zarządcy pamięci podręcznej odpowiadają również za informowanie zarządcy pamięci wirtualnej o potrzebie wyrzucania zawartości pamięci podręcznej na dysk. Zastępco zarządcy pamięci podręcznej stosuje algorytm składowania polegający na akumulowaniu zapisów i okresowym – co 4 do 5

sekund – budzeniu wątku przepisującego. Jeśli zachodzi potrzeba natychmiastowego przepisywania zawartości pamięci podręcznej, to proces może określić odpowiedni znacznik przy otwieraniu pliku lub może jawnie wywoływać funkcję opróżniania pamięci podręcznej w dogodnej dla siebie chwili.

Proces szybko piszący mógłby potencjalnie zapłonieć wszystkie wolne strony pamięci podręcznej, zanim wątek przepisujący pamięć podręczną miałby szansę na uaktywnienie i wyrzucenie stron na dysk. Aby zapobiec zalaniu systemu przez proces taką ilością danych, zarządcą pamięci podręcznej blokuje przejściowo procesy usiłujące zapisywać dane, jeśli ilość wolnej pamięci podręcznej staje się mała, budząc wątek przepisywania w celu przeniesienia stron na dysk. Jeśli proces szybko piszący jest w rzeczywistości sieciowym programem dokonującym przeadresowań na sieciowy system plików, to blokowanie go przez zbyt długi czas mogłoby spowodować wyczerpanie czasu przeznaczonego na przesyłanie w sieci i doprowadzić do retransmitowania danych. Retransmisje takie byłyby marnowaniem przepustowości sieci. W celu zapobieżenia stratom sieciowe programy przeadresowujące mogą informować zarządcę pamięci podręcznej, aby nie dopuszczał do gromadzenia w pamięci podręcznej wielkich ilości niedokończonych zamówień pisania.

Ponieważ sieciowy system plików musi przemieszczać dane między dyskiem a interfejsem sieci, zarządcą pamięci podręcznej dostarcza także interfejs DMA, aby przenosić dane do pamięci bezpośrednio. Bezpośrednie przenoszenie danych pozwala unikać ich kopiowania przez pośrednie bufory.

23.3.7 Zarządcza bezpieczeństwa odwołań

Obiektowa natura systemu NT umożliwia zastosowanie jednolitego mechanizmu bieżącego sprawdzania dostępu oraz doglądania wszystkich jednostek w systemie. Jeżeli tylko proces zgłasza się po uchwyt do obiektu, to monitor bezpieczeństwa odwołań sprawdza żeton bezpieczeństwa procesu i listę kontroli dostępu obiektu, aby przekonać się, czy proces ma niezbędne uprawnienia.

23.4 ■ Podsystemy środowiskowe

Podsystemy środowiskowe są procesami działającymi w trybie użytkownika, pomieszczenymi w warstwie powyżej usług rdzennego egzekutora NT; ich zadaniem jest umożliwianie systemowi NT wykonywania programów opracowanych dla innych systemów operacyjnych, takich jak: 16-bitowy system Windows, MS-DOS, POSIX oraz znakowe aplikacje 16-bitowego systemu OS/2. Każdy podsystem środowiskowy zawiera interfejs programowania aplikacji (API) lub przeznaczone dla aplikacji środowisko.

System NT korzysta z podsystemu Win32 jako z głównego środowiska operacyjnego; podsystem ten jest używany do rozpoczętania wszystkich procesów. Gdy ma nastąpić wykonanie aplikacji, wówczas podsystem Win32 wywołuje zarządcę pamięci wirtualnej (VM), aby ten załadował wykonywalny kod aplikacji. Zarządcą pamięci zwraca podsystemowi Win32 informacje o rodzaju kodu wykonywalnego. Jeżeli nie jest to rdzenny kod wykonywalny systemu Win32, to środowisko Win32 sprawdza, czy działa odpowiedni podsystem środowiskowy. Jeśli potrzebny podsystem nie jest wykonywany, to następuje jego uruchomienie jako procesu w trybie użytkownika. Następnie podsystem Win32 tworzy proces do wykonania aplikacji i przekazuje sterowanie właściwemu podsystemowi środowiskowemu.

Podsystem środowiskowy korzysta z udogodnienia lokalnego wywoływanego procedur systemu NT w celu uzyskiwania dla procesu usług jądra. Postępowanie takie zwiększa odporność systemu NT, gdyż parametry podawane w wywoaniu systemowym mogą być sprawdzane przed wywołaniem rzeczywistej procedury jądra. System NT zakazuje aplikacjom mieszanego procedur interfejsu API z różnych środowisk. Na przykład aplikacji Win32 nie wolno wywoływać procedur systemu POSIX.

Ponieważ każdy podsystem działa jako odrębny proces poziomu użytkownika, więc awaria jednego podsystemu nie wywołuje żadnych skutków w innych podsystemach. Wyjątek stanowi podsystem Win32, zapewniający wszystkie funkcje klawiatury, myszki i ekranu graficznego. W razie jego awarii system zostaje praktycznie unieruchomiony.

Podsystem Win32 dzieli aplikacje na graficzne i znakowe, przy czym za znakową uważa się aplikację, która zachowuje się tak, jakby wyjście interakcyjne dokonywało się na terminalu ASCII o wymiarach 80 na 24. Podsystem Win32 przekształca wyjście aplikacji znakowych na graficzną reprezentację okienkową. Przekształcenie takie jest łatwe: ilekroć jest wywoływana procedura wyjścia, tylekroć podsystem środowiskowy wywołuje procedurę systemu Win32, która wyświetla tekst. Ponieważ środowisko Win32 wykonuje tę czynność dla wszystkich okien znakowych, może przekazywać ekran tekstowy między oknami za pomocą schowka. Przekształcenia takie działają zarówno dla aplikacji systemu MS-DOS, jak i dla aplikacji złożonych z ciągów poleceń systemu POSIX.

23.4.1 Środowisko systemu MS-DOS

System MS-DOS nie jest tak złożony jak inne podsystemy środowiskowe systemu NT. Realizuje go aplikacja podsystemu Win32, nazywana maszyną wirtualną systemu DOS (ang. *virtual dos machine* – VDM). Ponieważ maszyna VDM jest po prostu procesem użytkowym, podlega ona stronicowaniu

i planowaniu przydziału procesora według reguł dotyczących każdego innego wątku systemu NT. Maszyna VDM ma jednostkę wykonywania rozkazów, której zadaniem jest wykonywanie lub emulowanie rozkazów procesora Intel 486. Zawiera ona również procedury emulowania funkcji ROM BIOS systemu MS-DOS oraz usług osiąganych za pomocą programowego przerwania „int 21”. Maszyna VDM ma także moduły sterujące wirtualnych urządzeń ekranu, klawiatury i portów komunikacyjnych. Maszyna VDM działa na podstawie kodu źródłowego systemu MS-DOS 5.0, pozostawiając programom użytkowym co najmniej 620 KB wolnej pamięci.

Powłoka polecen systemu NT jest programem, który tworzy okno wyglądające jak środowisko systemu MS-DOS. Może ona wykonywać zarówno programy 16-bitowe, jak i 32-bitowe. Jeśli ma być wykonana aplikacja systemu MS-DOS, to powłoka polecen rozpoczyna proces VDM, którego zadaniem jest wykonanie programu tej aplikacji.

Jeżeli system NT działa na procesorze x86, to graficzne aplikacje systemu MS-DOS działają w trybie pełnoekranowym, a aplikacje znakowe mogą działać w trybie pełnoekranowym lub w oknie. Jeśli system NT działa na procesorze o innej architekturze, to wszystkie aplikacje systemu MS-DOS są wykonywane w oknach. Niektóre z programów użytkowych systemu MS-DOS kontaktują się ze sprzętem dyskowym bezpośrednio – te aplikacje nie mogą działać poprawnie w systemie NT, ponieważ dostęp do dysku wymaga przywileju ze względu na ochronę systemu plików. Ogólnie biorąc, można powiedzieć, że aplikacje systemu MS-DOS, które kontaktują się ze sprzętem bezpośrednio, nie nadają się do wykonywania pod nadzorem systemu NT.

Ponieważ system MS-DOS nie jest środowiskiem wielozadaniowym, niektóre z jego aplikacji napisano w sposób, który monopolizuje jednostkę centralną, na przykład przez stosowanie pętli opóźniających lub przerw w działaniu. Mechanizm priorytetów stosowany przez ekspedytora w systemie NT wykrywa takie opóźnienia i automatycznie dławia zużycie czasu procesora (powodując, że zaborcze aplikacje działają nieprawidłowo).

23.4.2 Środowisko 16-bitowego systemu Windows

Maszyna VDM zawiera środowisko wykonawcze Win16 złożone z dodatkowego oprogramowania, nazywanego „Oknami w oknach” (ang. *Windows on Windows*). Dostarcza ono procedur jądra systemu Windows 3.1 oraz namiastek procedur (ang. *stubs*) zarządcy okien i funkcji interfejsu GDI. Namiastki procedur wywołują odpowiednie podprogramy systemu Win32, które zamieniają adresy 16-bitowe na 32-bitowe. Aplikacje zależne od wewnętrznej

^a Czyli zrębów procedur nazywanych też „stópkami”. – Przyp. tłum.

struktury 16-bitowego zarządcy okien lub interfejsu GDI mogą nie działać, gdyż „Okna w oknach” nie realizują naprawdę 16-bitowego interfejsu API.

„Okna w oknach” mogą być wykonywane wraz z innymi procesami w systemie NT, lecz pod wieloma względami przypominają one system Windows 3.1. W danej chwili może działać tylko jedna aplikacja Win16, wszystkie aplikacje są jednowątkowe i pozostają w tej samej przestrzeni adresowej, jak również dzielą tę samą kolejkę wejściową. Cechy te powodują, że aplikacja, która przestanie pobierać dane z wejścia, zablokuje wszystkie inne aplikacje Win16 – zupełnie jak w systemie Windows 3.x. Jedna aplikacja Win16 może też spowodować awarię innych aplikacji Win16, uszkadzając przestrzeń adresową.

23.4.3 Środowisko Win32

Głównym podsystemem systemu NT jest podsystem Win32. Wykonuje on aplikacje Win32 i zarządza wszystkimi funkcjami klawiatury, myszki i ekranu. Ponieważ jest on środowiskiem nadzorczym, zaprojektowano go jako podsystem możliwie wysoko odporny. Z odpornością tą jest związanych kilka jego właściwości. W przeciwieństwie do środowiska Win16 każdy proces podsystemu Win32 ma własną kolejkę wejściową. Zarządcy okien przydzielają wszystkie działania wejściowe systemu do odpowiednich procesowych kolejek wejściowych, toteż proces uszkodzony nie blokuje wejścia przeznaczonego dla innych procesów. Jądro systemu NT stosuje również wielozadaniowość z wywłaszczeniem, co pozwala użytkownikowi kończyć aplikacje, które uległy awariom lub nie są dalej potrzebne. Podsystem Win32 sprawdza też ważność wszystkich obiektów przed ich użyciem, aby zapobiegać załamaniom, do których mogłoby dochodzić z powodu prób korzystania w aplikacjach z nieważnych lub złych uchwytów. Podsystem Win32 przed użyciem obiektu wskazywanego za pomocą uchwytu sprawdza jego typ. Liczniki odwołań przechowywane przez zarządcę obiektów chronią przed usuwaniem obiektów będących w użyciu i zapobiegają przed używaniem obiektów już usuniętych.

23.4.4 Podsystem POSIX

Podsystem POSIX zaprojektowano po to, aby wykonywać aplikacje normy POSIX, spełniające standard POSIX.1, mający za podstawę model systemu UNIX. Aplikacje POSIX mogą być rozpoczynane przez podsystem Win32 lub przez inne aplikacje POSIX. Aplikacje POSIX korzystają z serwera PSXSS.EXE realizującego podsystem POSIX, dołączanej dynamicznie biblioteki PSXDLL.DLL oraz z zarządcy sesji operatorskiej POSIX.FXE.

Choc standard POSIX nie określa drukowania, aplikacje POSIX mogą przezrocznie korzystać z drukarek za pośrednictwem mechanizmu przesłania systemu NT. Aplikacje POSIX mają dostęp do dowolnego systemu plików w systemie NT. Środowisko POSIX narzuca uniksopodobne prawa dostępu do drzew katalogów. Kilka udogodnień podsystemu Win32 nie zrealizowano w podsystemie POSIX, m.in. plików odwzorowywanych w pamięci, działań sieciowych, graficznych i dynamicznej wymiany danych.

23.4.5 Podsystem OS/2

Choć system NT miał dostarczyć odpornego na awarie środowiska operacyjnego systemu OS/2, sukces systemu Microsoft Windows spowodował, że doszło do zmian we wczesnym stadium projektowania, prowadzących do tego, że środowisko Windows przyjęto traktować jako obowiązujące w systemie NT zastępczo. W konsekwencji system NT dostarcza w podsystemie środowiskowym OS/2 jedynie ograniczonych udogodnień. Aplikacje znakowe systemu OS/2 1.x można wykonywać tylko w systemie NT pracującym na komputerach zaopatrzonych w procesory Intel x86. Aplikacje pracujące w systemie OS/2 w trybie rzeczywistym mogą działać na wszelkich platformach, korzystając ze środowiska MS-DOS. Aplikacje łączone, zawierające kod przeznaczony dla systemu MS-DOS i dla systemu OS/2, działają w środowisku OS/2, jeśli użytkowanie tego środowiska nie jest zakazane.

23.4.6 Podsystemy rejestracji i bezpieczeństwa

Zanim użytkownik uzyska dostęp do obiektów systemu NT, musi uwierzytelnić swoją tożsamość za pomocą podsystemu rejestracji. W celu uwierzytelnienia użytkownik musi mieć konto w systemie i podać odpowiednie dla tego konta hasło.

Podsystem bezpieczeństwa wytwarza zetony dostępu służące do reprezentowania użytkowników w systemie. Wywołuje on pakiet uwierzytelniania (ang. *authentication package*) w celu sprawdzenia tożsamości za pomocą informacji pochodzących z podsystemu rejestracji lub z serwera sieciowego. Zazwyczaj pakiet uwierzytelniania przegląda po prostu informacje dotyczące kont w lokalnej bazie danych i sprawdza poprawność hasła. Podsystem bezpieczeństwa generuje wówczas odpowiadający identyfikatorowi użytkownika zeton dostępu zawierający stosowne przywileje, ograniczenia ilościowe i identyfikatory grup. Gdy tylko użytkownik spróbuje sięgnąć po obiekt w systemie, na przykład starając się o jego uchwyt, zeton dostępu zostaje przekazany monitorowi bezpieczeństwa odwołani, który przywileje te i limity poddaje kontroli.

23.5 ■ System plików

Ujmując rzecz historycznie, w systemach MS-DOS zastosowano system plików w postaci *tablicy przydziału plików* (ang. *file-allocation table* – FAT). Szesnastobitowy system plików FAT miał kilka wad, m.in. wewnętrzną fragmentację i ograniczenie wielkości do 2 GB, a także brak ochrony dostępu do plików. W trzydziestodwubitowym systemie plików FAT poradzono sobie z problemami rozmiaru i fragmentacji, lecz jego wydajność i inne cechy były wciąż słabe w porównaniu z nowoczesnymi systemami plików. System NTFS jest znacznie lepszy. Zaprojektowano go, biorąc pod uwagę wiele cech, takich jak odtwarzanie danych, bezpieczeństwo, tolerowanie uszkodzeń, wielkie pliki i systemy plików, wielość strumieni danych, nazwy standardu UNICODE oraz upakowywanie plików. Ze względu na zgodność w systemie NT istnieje możliwość korzystania z systemów plików FAT i OS/2 HPFS.

23.5.1 Budowa wewnętrzna

Podstawową jednostką systemu NTFS jest *tom* (ang. *volume*). Tom jest tworzony przez program administrowania dyskiem systemu NT; u jego podstaw leży logiczny podział dysku. Tom może zajmować część dysku lub cały dysk, może też rozciągać się na kilka dysków. W systemie NTFS wszystkie metadane, takie jak informacje dotyczące tomu, są pamiętane w zwykłym pliku.

System NTFS nie ma do czynienia z poszczególnymi sektorami dysku. Zamiast nich używa gron jako jednostek przydziału dyskowego. Grono* jest grupą przyległych sektorów dyskowych, których liczba jest potęgą liczby 2. Wielkość grona jest ustalana podczas formatowania systemu NTFS. Dla tomów nie przekraczających 512 MB zastępca wielkości grona równa się rozmiarowi sektora, tomy o rozmiarach do 1 GB mają zastępczo grona o wielkości 1 KB, grona tomów o wielkości do 2 GB wynoszą zastępczo 2 KB, a tomy jeszcze większe mają grona 4-kilobajtowe. Rozmiar tych gron jest znacznie mniejszy niż w 16-bitowym systemie plików FAT, co zmniejsza wielkość wewnętrznej fragmentacji. Jako przykład rozważmy dysk o pojemności 1,6 GB zawierający 16 000 plików. Jeśli użyto by systemu plików FAT-16, to 400 MB mogłyby zostać zmarnowanych na wewnętrzną fragmentację, gdyż wielkość grona wynosi 32 KB. Pod systemem NTFS przy pamiętaniu tej samej liczby plików zmarnowałoby się tylko 17 MB.

System NTFS używa w charakterze adresów dyskowych *logicznych numerów gron* (ang. *logical cluster numbers* – LCN). Przypisuje je przez ponu-

* W obiegu jest też zapożyczenie „klaster” od angielskiego *cluster*: grono, zlepck – Przyp. tłum.

merowanie gron od początku dysku do jego końca. Za pomocą tego schematu system może wyliczać fizyczną odległość na dysku (w bajtach), mnożąc numer LCN przez wielkość grona.

Plik w systemie NTFS nie jest zwyczajnym strumieniem bajtów, jak to ma miejsce w systemach MS-DOS lub UNIX. Plik jest tu obiektem strukturalnym złożonym z *atrybutów*. Kazdy atrybut pliku jest niezależnym strumieniem bajtów, który podlega tworzeniu, usuwaniu, czytaniu i zapisywaniu. Niektóre atrybuty są standardowe dla wszystkich plików, wliczając w to nazwę pliku (lub nazwy, jeśli plik ma synonimy), czas jego utworzenia oraz deskryptory bezpieczeństwa określający dozwolone rodzaje dostępu. Inne atrybuty są charakterystyczne dla plików pewnych rodzajów. Na przykład pliki typu Macintosh mają dwa atrybuty danych: odnośniki do zasobu i odnośniki do danych. Katalog ma atrybuty implementujące indeksy do zawartych w nim nazw plików. Większość tradycyjnych plików danych ma beznazwowy atrybut danych, mieszczący wszystkie dane pliku. Z opisu tego wynika jasno, że pewne atrybuty są małe, a inne – wielkie.

Każdy plik w systemie NTFS jest opisany przez jeden lub więcej rekordów tablicy przechowywanej w specjalnym pliku o nazwie: *główna tablica plików* (ang. *master file table* – MFT). Rozmiar rekordu jest określony podczas tworzenia systemu plików i waha się w granicach od 1 do 4 KB. Małe atrybuty przechowuje się w samym rekordzie MFT i nazywa rezydentnymi. Wielkie atrybuty, takie jak nienazwana masa danych – określane mianem nierezydentnych – są przechowywane w jednym lub większej liczbie ciągły rozszerzeń na dysku, do których wskaźniki przechowuje się w rekordzie MFT. W przypadku małych plików w rekordzie MFT może się zmieścić nawet atrybut danych. Jeżeli plik ma wiele atrybutów lub jeśli jest on mocno pofragmentowany i wymaga zapamiętania wielu wskaźników pokazujących wszystkie jego części, to jeden rekord w tablicy MFT może okazać się za mały. W tym przypadku plik jest opisany przez rekord o nazwie: *podstawowy rekord pliku* (ang. *base file record*), który zawiera wskaźniki do rekordów nadmiarowych, przechowujących pozostałe wskaźniki i atrybuty.

Każdy plik w tomie systemu NTFS ma niepowtarzalny identyfikator zwany *odsyłaczem do pliku* (ang. *file reference*). Odsyłacz do pliku jest wielkością 64-bitową, składającą się z 48-bitowego numeru pliku i 16-bitowego numeru kolejnego. Numer pliku jest numerem rekordu (tj. przegródki tablicy) w strukturze MFT opisującej plik. Numer kolejny jest zwiększany za każdym razem, gdy następuje powtórne użycie wpisu w tablicy MFT. Zwiększenie to umożliwia systemowi NTFS wykonywanie wewnętrznej kontroli spójności – na przykład wyłapywanie nieaktualnych odwołań do usuniętego pliku po użyciu wpisu MFT na nowy plik.

Podobnie jak w systemach MS-DOS i UNIX, przestrzeń nazwowa systemu NTFS jest zorganizowana w hierarchię katalogów. Każdy katalog stosuje strukturę danych zwaną B -drzewem, w której zapamiętuje indeks swoich nazw plików. B -drzewo znajduje zastosowanie dlatego, że pozwala ono unikać kosztu reorganizacji drzewa i ma tę cechę, że długość każdej ścieżki od korzenia drzewa do liścia jest taka sama. Górnny poziom B -drzewa zawiera indeks korzenia katalogu. W większych katalogach ten górnny poziom zawiera wskaźniki do rozszerzeń dyskowych przechowujących resztę drzewa. Każdy wpis w katalogu zawiera nazwę pliku i odsyłacz do niego oraz kopię znacznika czasu uaktualnienia i rozmiar pliku – pobrane z atrybutów pliku rezydujących w tablicy MFT. Kopie tych informacji są przechowywane w katalogu, co przyspiesza wyprowadzanie jego zawartości – nazwy wszystkich plików, ich rozmiary i czasy uaktualnień są obecne w samym katalogu, więc nie potrzeba ich zbierać na podstawie wpisów w tablicy MFT każdego z plików.

Wszystkie metadane tomu systemu NTFS są przechowywane w plikach. Pierwszym z takich plików jest tablica MFT. Drugi plik, używany do działań naprawczych w przypadku uszkodzenia tablicy MFT, zawiera kopię pierwszych szesnastu pozycji tablicy MFT. Kilka następnych plików ma także specjalne znaczenie. Ich nazwy to: plik dziennika, plik tomu, tablica definicji atrybutów, katalog główny, plik z mapą bitów, plik rozruchowy oraz plik złych gron. *Plik dziennika*, opisany w p. 23.5.2, przechowuje wszystkie uaktualnienia metadanych w systemie plików. *Plik tomu* zawiera nazwę tomu, dane o wersji systemu NTFS, który sformatował tom, oraz bit informujący, czy tom uległ uszkodzeniu i wymaga sprawdzenia spójności. *Tablica definicji atrybutów* pokazuje, jakie typy atrybutów są używane w tomie i jakie operacje można wykonywać na każdym z tych atrybutów. *Katalog główny* jest katalogiem z najwyższego poziomu hierarchii systemu plików. *Plik mapy bitów* wskazuje, które grona tomu są przydzielone do plików, a które pozostają wolne. *Plik rozruchowy* zawiera kod początkowy systemu NT i musi znajdować się pod specjalnym adresem na dysku, aby prosty program ładujący z pamięci ROM mógł go łatwo odnaleźć. Plik rozruchowy zawiera także fizyczny adres tablicy MFT. Na koniec *plik złych gron* przechowuje informacje o wszystkich wadliwych obszarach tomu; z informacji tych system NTFS korzysta przy usuwaniu skutków wystąpienia błędów.

23.5.2 Usuwanie skutków awarii

W wielu prostych systemach plików awaria zasilania występująca w złym czasie może uszkodzić struktury danych systemu plików tak poważnie, że zniszczeniu ulega cały tom. Wiele wersji systemu UNIX przechowuje na dysku nadmiarowe metadane i usuwa skutki awarii za pomocą programu fsek,

który sprawdza wszystkie struktury danych systemu plików, przywracając im siłą spójny stan. Odtwarzanie tych struktur często pociąga za sobą usunięcie uszkodzonych plików i zwolnienie gron danych zapisanych przez użytkownika, lecz nie zapamiętanych poprawnie w strukturach metadanych systemu plików. Sprawdzanie takie może być powolne i prowadzić do utraty znacznych ilości danych.

W systemie NTFS przyjęto inne podejście do zapewnienia odporności systemu plików. Wszystkie aktualnienia struktur danych systemu plików NTFS odbywają się w ramach *transakcji*. Zanim nastąpi zmiana struktury danych, transakcja zapisuje do dziennika działań wszelkie informacje niezbędne do powtórzenia lub anulowania podjętych czynności. Po zmianie struktury danych w dzienniku pojawia się wpis zatwierdzający, aby zaznaczyć, że transakcja zakończyła się pomyślnie. Po awarii system taki jest w stanie przywrócić struktury danych systemu plików do stanu spójnego w drodze przetwarzania zapisów dziennika: najpierw powtarzając czynności transakcji zatwierdzonych, a potem usuwając skutki działań transakcji, które nie dobiegły szczęśliwie do końca przed awarią. Okresowo (wykole co 5 s) do dziennika zapisuje się rekord zwany punktem kontrolnym (ang. *checkpoint*). System nie musi rejestrować rekordów sprzed punktu kontrolnego, aby poradzić sobie z awarią. Można je zatem usuwać, dzięki czemu plik dziennika nie rozrasta się w sposób nieograniczony. Po rozruchu systemu podczas pierwszego dostępu do tomu NTFS procedura usuwania skutków awarii jest wykonywana automatycznie.

Schemat ten nie gwarantuje, że zawartość wszystkich plików użytkownika będzie poprawna po awarii; zapewnia tylko, że struktury danych systemu plików (pliki metadanych) będą nieuszkodzone i będą odzwierciedlały pewien spójny stan przed awarią. Rozszerzenie schematu transakcyjnego na pliki użytkownika byłoby możliwe, lecz koszty byłyby niewspółmierne z wydajnością systemu plików.

Dziennik jest przechowywany w trzecim pliku metadanych na początku tomu. Tworzy się go ze stałym rozmiarem maksymalnym podczas formatowania systemu plików. Składa się on z dwu części: *obszaru rejestrowego* (ang. *logging area*), będącego cykliczną kolejką wpisów dziennika, i *obszaru ponownego startu* (ang. *restart area*), zawierającego informacje kontekstowe, takie jak pozycja w obszarze rejestrów, od której system NTFS powinien rozpoczęć czytanie podczas usuwania skutków awarii. W rzeczywistości w obszarze ponownego startu przechowuje się dwie kopie jego informacji, więc działania naprawcze są możliwe nawet wówczas, gdy w czasie awarii jedna z nich ulegnie uszkodzeniu.

Czynności aktualniania dziennika są realizowane przez *usługi rejestrowe* (ang. *log-file service*) systemu NT. Oprócz zapisywania rekordów dziennika i podejmowania działań naprawczych usługi rejestrowe dopilnowują, aby

w dzienniku było dość wolrego miejsca. Jeśli zaczyna brakować miejsca, to usługi rejestrów ustawiają niezakończone transakcje w kolejkę, a system NTFS wstrzymuje wszystkie nowe operacje wejścia-wyjścia. Po zakończeniu działań będących w toku system NTFS wzywa zarządcę pamięci podręcznej do opróżnienia jej ze wszystkich danych, po czym odświeża plik dziennika i wykonuje transakcje z kolejki.

23.5.3 Bezpieczeństwo

Bezpieczeństwo tomu NTFS wywodzi się z obiektowego modelu systemu NT. Każdy obiekt pliku ma w rekordzie tablicy MFT przechowywany atrybut deskryptora bezpieczeństwa. Atrybut ten zawiera żeton dostępu właściciela pliku oraz listę kontroli dostępu wykazującą przywileje dostępu udzielone użytkownikom, którym pozwolono na korzystanie z pliku.

23.5.4 Zarządzanie tomem i tolerowanie uszkodzeń

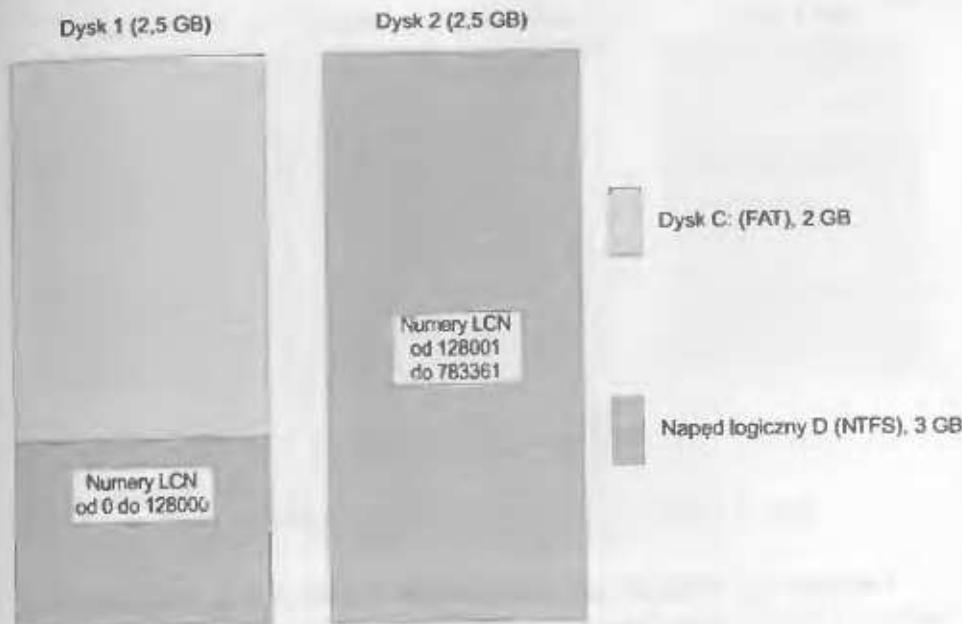
Program **FtDisk** jest tolerującym awarie modelem obsługi dysku systemu NT. Podczas instalowania dopuszcza on połączenie wielu napędów dyskowych w jeden tom logiczny, aby zwiększyć wydajność, pojemność i niezawodność.

Jeden sposób połączenia wielu dysków polega na ich logicznym zespoleniu w celu utworzenia wielkiego tomu logicznego, jak jest to pokazane na rys. 23.7. W systemie NT taki tom logiczny nazywa się *zbiorem tomów* (ang. *volume set*), który może zawierać do 32 stref fizycznych. Zbiór tomów zawierający tom systemu NTFS można rozszerzać bez naruszania danych, które już są zapamiętane w systemie plików. Rozszerzeniu podlega po prostu mapa bitowa metadanych tomu NTFS, tak aby objąć nowo dodaną przestrzeń. System NTFS nadal używa tego samego mechanizmu LCN^{*}, którego używał do jednego fizycznego dysku, a moduł sterujący **FtDisk** umożliwia odwzorowanie logicznej odległości w tomie na odległość na konkretnym dysku.

Inny sposób połączenia wielu stref fizycznych polega na rotacyjnym przeplataniu ich bloków w celu otrzymania struktury zwanej *zbiorem pasków* (ang. *stripe set*), co jest pokazane na rys. 23.8. Nazywa się to też *schematem RAID^{**} poziomu 0* lub *paskowaniem dysku* (ang. *disk striping*). Program **FtDisk** używa pasków o rozmiarze 64 KB. Pierwsze 64 KB tomu logicznego przechowuje się w pierwszej strefie fizycznej, drugie 64 KB tomu logicznego przechowuje się w drugiej strefie fizycznej itd., aż każda strefa dostanie

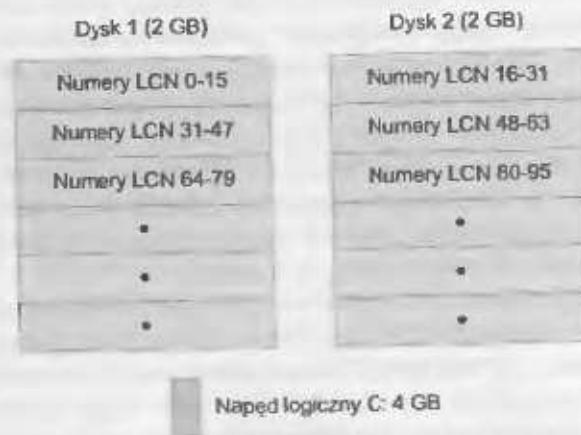
^{*} Logicznych numerów gron. – Przyp. tłum.

^{**} Nadmiernawa tablica niezależnych dysków (ang. *redundant array of independent disks*)

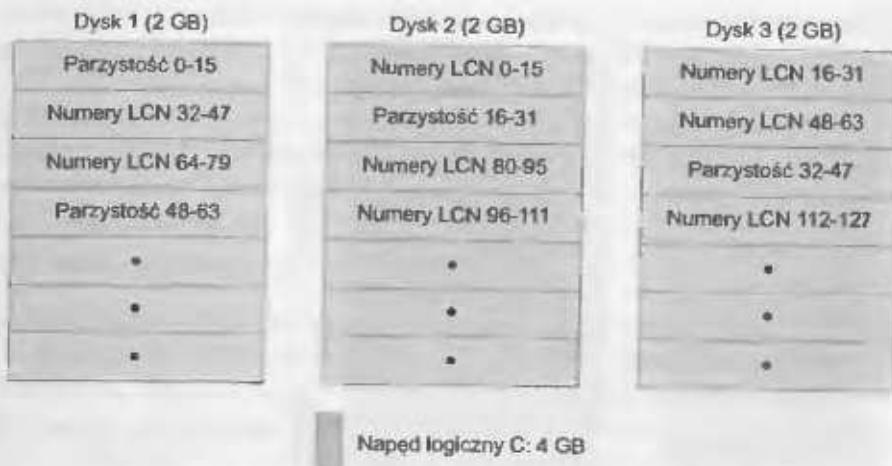


Rys. 23.7 Zbiór tomów na dwu napędach dysków

64 KB tej przestrzeni. Przydział kolejnych bloków wielkości 64 KB powtarza się potem, poczynając od pierwszego dysku. Zbiór pasków tworzy jeden wielki tom logiczny, lecz jego rozmieszczenie fizyczne może polepszyć przepustowość wejścia-wyjścia, gdyż dla wielkich operacji wejścia-wyjścia wszystkie dyski mogą przesyłać dane równolegle.



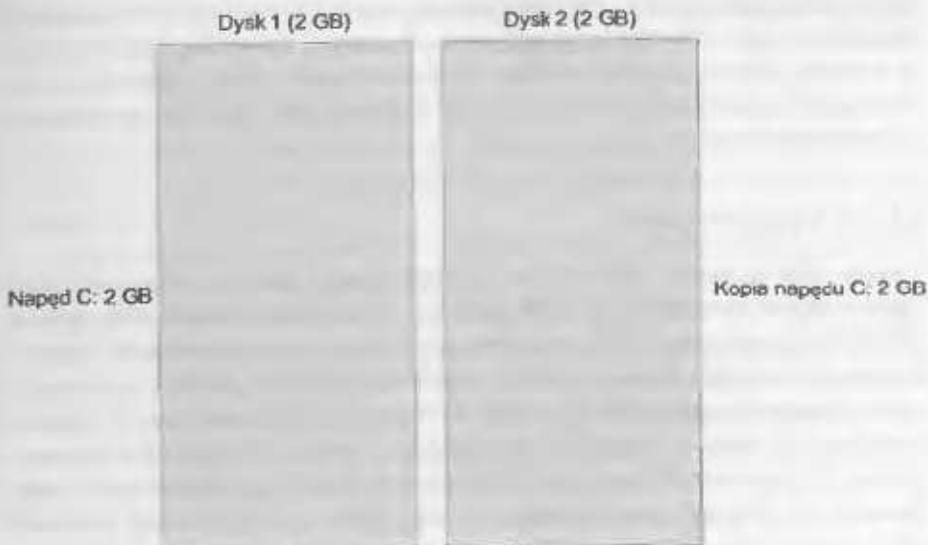
Rys. 23.8 Zbiór pasków na dwu napędach dysków



Rys. 23.9 Zbiór pasków z parzystością na trzech napędach dysków

Odmianą tego pomysłu jest zbiór pasków z parzystością, przedstawiony na rys. 23.9. Nosi on również nazwę RAID poziomu 5. Jeśli zbiór pasków ma osiem dysków, to dla każdego z siedmiu pasków danych, rozmieszczonych na siedmiu osobnych dyskach, na ósmym dysku znajdzie się pasek parzystości. Pasek parzystości zawiera zgodną z kierunkiem bajtów dysjunkcję (ang. *exclusive or*) pasków danych. Jeśli którykolwiek z ósmiu pasków zostanie uszkodzony, to system będzie w stanie zrekonstruować dane przez obliczenie wartości dysjunkcji pozostałych siedmiu. Owa zdolność do rekonstrukcji danych czyni tablicę dysków znacznie mniej podatną na utratę danych w wypadku awarii dysku. Zauważmy, że zaktualizowanie jednego paska danych wymaga również przeliczenia paska parzystości. Siedem współbieżnych zapisów do siedmiu różnych pasków danych wymagałoby więc uaktualnienia także siedmiu pasków parzystości. Gdyby wszystkie paski parzystości były zgromadzone na tym samym dysku, to dysk ten byłby siedem razy bardziej obejmowany niż dyski z danymi. Aby uniknąć tworzenia tego wąskiego gardła, rozrzucamy paski parzystości po wszystkich dyskach na zasadzie rotacyjnej. Do budowy zbioru pasków z parzystością potrzeba co najmniej trzech jednakowej wielkości stref położonych na trzech osobnych dyskach.

Jeszcze odporniejszy schemat zwie się *dyskami lustrzanymi* (ang. *disk mirroring*) lub *schematem RAID poziomu 1*. Widać go na rys. 23.10. *Zestaw luster* (ang. *mirror set*) składa się z dwu równej wielkości stref na dwóch dyskach, takich że zawarte w nich dane są identyczne. Kiedy aplikacja zapisuje dane do zestawu luster, program FtDisk zapisuje je w obu strefach. Jeśli jedna strefa ulegnie awarii, to program FtDisk ma jeszcze drugą kopię bezpiecznie



Rys. 23.10 Zestaw luster na dwu napędach dysków

przechowaną w lustrze. Zestaw luster może także polepszać wydajność, gdyż zamówienia czytania mogą być rozdzielane między oba lustra, powodując dwukrotnie mniejsze obciążenie każdego lustra. Aby uchronić się przed awarią sterownika dysku, oba lustrzane dyski możemy połączyć z osobnymi sterownikami. Taka organizacja jest nazywana *zestawem duplexowym* (ang. *duplex set*).

Aby poradzić sobie z uszkodzonymi sektorami dysku, moduł sterujący FtDisk korzysta z rozwiązania sprzętowego zwanego *zapasem sektorów* (ang. *sector sparing*), a system NTFS stosuje technikę programową zwaną *wtórnym odwzorowaniem grona* (ang. *cluster remapping*). Zapas sektorów jest właściwością sprzętu udostępnianą przez wiele modułów obsługi dysków. Podczas formatowania dysku z logicznych numerów bloków dyskowych tworzy się mapę dobrych sektorów na dysku. Część sektorów pozostawia się nie ujętych w mapie – w charakterze rezerwy. W razie awarii sektora moduł sterujący FtDisk instruuje napęd dysku, aby dokonał jego zamiany na sektor zapasowy. Wtórne odwzorowanie grona jest techniką programową realizowaną przez system plików. Jeśli blok dyskowy ulega uszkodzeniu, to system NTFS zastępuje go innym, nie przydzielonym blokiem, zmieniając wszystkie związane z nim wskaźniki w tablicy MFT. System NTFS odnotowuje także, aby uszkodzonego bloku nigdy nie przydzielono żadnemu plikowi.

Uszkodzeniu bloku dyskowego zazwyczaj towarzyszy utrata danych. Jednak w celu maskowania uszkodzeń bloków dyskowych zapas sektorów lub wtórne odwzorowanie grona można połączyć z tomami tolerującymi awarie,

takini jak zbiory pasków. W razie nieudanego czytania system rekonstruuje zagubione dane, czytając je z lustra lub obliczając dysjunkcyjną parzystość w zbiorze pasków z parzystością. Zrekonstruowane dane zapamiętuje się w nowym miejscu uzyskiwanym dzięki zapasowi sektorów lub ponownemu odwzorowaniu grona.

23.5.5 Upakowywanie

System NTFS może upakowywać (kompresować) dane w poszczególnych plikach lub we wszystkich plikach katalogu. W celu upakowania pliku system NTFS dzieli go na jednostki upakowywania (ang. *compression units*) złożone z szesnastu kolejnych gron. Podczas zapisywania każdej jednostki upakowywania następuje wykonanie algorytmu kompresji danych. Jeśli wynik zajmuje mniej niż 16 gron, to następuje zapamiętanie wersji upakowanej (skompresowanej). Podczas czytania system NTFS potrafi określić, czy dane zostały upakowane, bo długość zapamiętanej jednostki upakowania jest wtedy mniejsza niż 16 gron. W celu polepszenia wydajności przy czytaniu ciągu upakowanych jednostek system NTFS dokonuje z wyprzedzeniem ich pobrania i rozpakowania, zanim jeszcze aplikacja złoży na nie zamówienie.

Dla plików rozrzedzonych lub takich, które zawierają w większości zera, system NTFS stosuje inną technikę oszczędzania pamięci. Gronom zawierającym same zera nie przydziela się żadnego miejsca na dysku. Zamiast tego pozostawia się przerwy w ciągu numerów gron wirtualnych pamiętanych we wpisie danego pliku w tablicy MFT. Podczas czytania pliku, w przypadku znalezienia przerwy w numerach wirtualnych gron system NTFS po prostu wypełnia daną porcję zerami w buforze docelowym. Technika taka jest również stosowana przez system UNIX.

23.6 ■ Praca w sieci

System NT umożliwia pracę w sieci zarówno w trybie „każdy z każdym”, jak i w trybie klient-serwer. Ma również udogodnienia do zarządzania siecią. Sieciowe części systemu NT obejmują transport danych, komunikację międzyprocesową, wspólne użytkowanie plików przez sieć oraz możliwość wysyłania zadań drukowania do odległych drukarek.

System NT zawiera wiele protokołów stosowanych w sieciowej pracy komputerów – są one opisane w p. 23.6.1. W punkcie 23.6.2 przedstawiamy mechanizmy przetwarzania rozproszonego w systemie NT. *Zwrotnicą* (ang. *redirector*) nazywa się w systemie NT obiekt dostarczający jednolitego interfejsu dla plików, niezależnie od tego, czy są one lokalne czy odległe – oma-

wiamy go w p. 23.6.3. *Domeną* (ang. *domain*) nazywa się grupę maszyn usługowych systemu NT, które stosują wspólne zasady bezpieczeństwa i korzystają ze wspólnej bazy danych użytkownika, co opisujemy w p. 23.6.4. System NT ma jeszcze mechanizmy *rozbiórku nazw* (ang. *name resolution*) pozwalające jednemu komputerowi poszukiwać adresu innego komputera na podstawie znanej nazwy tego komputera. W punkcie 23.6.5 wyjaśniamy, jak mechanizmy te działają.

Przy opisie pracy sieciowej w systemie NT, będziemy odwoływać się do dwóch wewnętrznych interfejsów sieciowych, z których jeden nazywa się *specyfikacją interfejsu urządzenia sieci* (ang. *Network Device Interface Specification* – NDIS), a drugi nosi nazwę *interfejsu modułu obsługi transportu* (ang. *Transport Driver Interface* – TDI). Interfejs NDIS został opracowany w 1989 r. przez firmy: Microsoft i 3Com w celu oddzielenia adapterów sieciowych od protokołów transportowych, tak aby każdy z nich mógł podlegać niezależnym zmianom. Interfejs NDIS zajmuje w modelu OSI miejsce między warstwą sterowania łączami danych a warstwą sterowania dostępem do mediów i umożliwia stosowanie wielu protokołów działających ponad wieloma różnymi adapterami sieciowymi. W terminologii modelu OSI interfejs TDI lokuje się między warstwą transportu (warstwa 4) a warstwą sesji (warstwa 5). Interfejs ten umożliwia dowolnej składowej warstwy sesji stosowanie dowolnego dostępnego mechanizmu transportu. (Podobny sposób myślenia doprowadził do powstania mechanizmu strumieni w systemie UNIX). Interfejs TDI umożliwia zarówno transport połączeniowy, jak i bezpołączeniowy i ma funkcje do przesyłania danych dowolnego typu.

23.6.1 Protokoly

System NT implementuje protokoły transportowe w postaci modułów sterujących. Moduły te można wprowadzać do systemu i usuwać z niego dynamicznie, chociaż w praktyce system musi być na ogół po takiej zmianie uruchomiony od nowa. System NT jest zaopatrzony w kilka protokołów sieciowych.

Protokół bloku komunikatów serwera (ang. *server message-block* – SMB) wprowadzono po raz pierwszy w systemie MS-DOS 3.1. Używa się go do przesyłania zamówień wejścia-wyjścia przez sieć. Protokół SMB ma cztery typy komunikatów. Komunikaty sterowania sesją są polecaniami, które rozpoczęją i kończą działanie połączenia przeadresowanego na wspólny zasób serwera. Komunikaty plikowe są używane przez zwrotnicę do uzyskiwania dostępu do plików serwera. Komunikaty drukowania są stosowane do przesyłania danych do zdalnej kolejki drukowania i do odbierania informacji o jej stanie, a komunikat typu „wiadomość” (ang. *message*) służy do kontaktu z inną stacją roboczą.

Podstawowy, sieciowy system wejścia-wyjścia (ang. *Network Basic Input/Output System* – NetBIOS) jest interfejsem abstrakcji sprzętu na podobieństwo interfejsu abstrakcji sprzętu BIOS zaprojektowanego dla komputerów PC działających pod nadzorem systemu MS-DOS. NetBIOS opracowano we wczesnych latach osiemdziesiątych; stał się on standardowym interfejsem programowania sieci. NetBIOS jest stosowany do ustanawiania nazw logicznych w sieci, do ustanawiania połączeń logicznych (czyli sesji) między dwoma nazwami logicznymi w sieci oraz do zapewniania niezawodnego przesyłania danych w sesji za pomocą poleceń protokołu NetBIOS lub SMB.

Rozszerzony interfejs użytkownika NetBIOS (ang. *NetBIOS Extended User Interface* – NetBEUI) wprowadziła firma IBM w 1985 r. jako prosty i wydajny protokół sieciowy, przydatny w pracy z co najwyżej 254 maszynami. Jest on zastępczym protokołem do kontaktów dwustronnych (ang. *peer-to-peer*) w systemie Windows 95 oraz w systemie Windows przeznaczonym do pracy zespołowej (ang. *Windows for Workgroups*). System NT używa protokołu NetBEUI, jeśli zachodzi potrzeba dzielenia zasobów z tymi sieciami. Wśród ograniczeń protokołu NetBEUI znajduje się to, że w roli adresu stosuje on faktyczną nazwę komputera i nie umożliwia wyznaczania tras.

Komplet protokołów TCP/IP stosowany w sieci Internet stał się de facto standardową infrastrukturą sieciową i jest szeroko używany. System Windows NT stosuje protokół TCP/IP do połączeń z wieloma różnymi systemami operacyjnymi i platformami sprzętowymi. Pakiet NT TCP/IP zawiera prosty protokół zarządzania siecią (SNMP), protokół dynamicznego konfigurowania komputera sieciowego (DHCP), usługi nazewnictwa Windows Internet (WINS) oraz wsparcie dla protokołu NetBIOS.

Tunelowy protokół od punktu do punktu (ang. *point-to-point tunelling protocol* – PPTP) jest nowym protokołem występującym w systemie Windows NT 4.0 jako środek komunikacji między modułami serwera zdalnego dostępu, działającymi na maszynach z systemem NT połączonych za pomocą sieci Internet. Serwery zdalnego dostępu mogą szyfrować dane przesyłane za pośrednictwem połączenia i dostarczać wieloprotołowych, prywatnych sieci wirtualnych w ramach sieci Internet.

Protokoły systemu Novell NetWare (obsługa datagramów IPX w warstwie transportowej SPX) są szeroko stosowane w lokalnych sieciach komputerów PC. Protokół NT NWLink łączy NetBIOS z sieciami NetWare. W połączeniu ze zwrotnicą, taką jak Client Service for Netware firmy Microsoft lub NetWare Client for Windows NT firmy Novell, protokół ten umożliwia klientowi systemu NT łączenie się z serwerem systemu NetWare.

Protokół sterowania łączeniem danych (ang. *data-link control* – DLC) jest stosowany do kontaktu z komputerami głównymi IBM oraz z drukarkami HP

bezpośrednio podłączonymi do sieci. W innych przypadkach nie jest on używany przez systemy NT.

Protokół AppleTalk zaprojektowano jako tanie połączenie w firmie Apple po to, żeby komputery Macintosh mogły dzielić się plikami. Systemy NT mogą dzielić pliki i drukarki z komputerami Macintosh za pośrednictwem protokołu AppleTalk, jeżeli serwer NT w sieci wykonuje pakiet usług Windows NT dla komputerów Macintosh.

23.6.2 Mechanizmy przetwarzania rozproszonego

Choć NT nie jest rozproszonym systemem operacyjnym, umożliwia rozproszone zastosowania. Mechanizmy wspierające przetwarzanie rozproszone w systemie NT obejmują protokół NetBIOS, potoki nazwane i skrytki pocztowe, gniazda standardu Windows, zdalne wywołania procedur (RPC) oraz sieciową, dynamiczną wymianę danych (NetDDE).

W systemie NT aplikacje NetBIOS mogą się komunikować przez sieć za pomocą protokołów NETBEUI, NWLink lub TCP/IP.

Potoki nazwane (ang. *named pipes*) są połączeniowym mechanizmem przekazywania komunikatów. Pierwotnie opracowano je jako wysokiego poziomu interfejs do sieciowych połączeń standardu NetBIOS. Proces może posłużyć się potokami nazwanymi także do komunikacji z innym procesem na tej samej maszynie. Ponieważ potoki nazwane są dostępne z udziałem interfejsu systemu plików, więc można do nich odnosić mechanizmy bezpieczeństwa stosowane do obiektów plikowych.

Format nazwy potoku nosi miano *ujednoliconej konwencji nazewniczej* (ang. *uniform naming convention* – UNC). Nazwa UNC wygląda jak typowa nazwa odległego pliku. Format nazwy UNC ma postać: \\nazwa_serwera\\nazwa_wspólna\\x\\y\\z, gdzie nazwa_serwera identyfikuje serwer sieciowy, nazwa_wspólna określa dowolny zasób udostępniony użytkownikom sieci, taki jak katalog, plik, potok nazwany lub drukarka, a część \\x\\y\\z jest zwykłą nazwą ścieżki pliku.

Skrytki pocztowe (ang. *mailslots*) są bezpołączeniowym mechanizmem komunikacji. Skrytki pocztowe mogą zawodzić – komunikat wysłany do skrytki pocztowej może zostać utracony, zanim zdąży go odebrać przewidziany adresat. Skrytek pocztowych używa się w aplikacjach rozglaszających, na przykład do szukania jakichś elementów w sieci. Skrytki pocztowe znajdują także zastosowanie w usługach przeglądania systemu NT (ang. *NT Computer Browser*).

Program Winsock tworzy interfejs API gniazd standardu Windows. Winsock jest interfejsem warstwy sesji, w dużej mierze zgodnym z gniazdami systemu UNIX, z pewnymi rozszerzeniami dotyczącymi okien. Tworzy on

standaryzowany interfejs do wielu protokołów transportowych, które mogą mieć różne schematy adresowania, toteż dowolna aplikacja Winsock może działać z każdym stosem protokołów poddającym się wymaganiom standardu Winsock.

Zdalne wywołanie procedury (ang. *remote procedure call* – RPC) jest mechanizmem typu klient-serwer pozwalającym aplikacji na jednej maszynie wywoływać procedurę, której kod znajduje się na innej maszynie. Klient wywołuje procedurę lokalną – tzw. *namiastkę* (ang. *stub routine*), która pakuje swoje argumenty do komunikatu i wysyła je przez sieć do konkretnego procesu usługowego. Po tej czynności procedura-namiastka po stronie klienta ulega zablokowaniu. W międzyczasie serwer rozpakowuje komunikat, wywołuje procedurę, pakuje zwrocone przez nią wyniki do nowego komunikatu i wysyła je z powrotem do procedury-namiastki klienta. Procedura-namiastka klienta zostaje odblokowana, odbiera komunikat, rozpakowuje wyniki wywołania RPC i przekazuje je do miejsca wywołania. Pakowanie argumentów bywa nazywane *przetaczaniem* (ang. *marshaling*).

Mechanizm NT RPC odpowiada będącemu w szerokim użyciu standardowi komunikatów RPC środowiska obliczeń rozproszonych, więc programy korzystające z wywołań NT RPC są w dużym stopniu przenośne. Standard RPC jest szczegółowy. Przesłania on wiele różnic w architekturze komputerów, takich jak rozmiary liczb dwójkowych czy kolejność bajtów i bitów w słowach maszynowych, precyzując standardowy format danych dla komunikatów RPC.

W sieciach z protokołem TCP/IP system NT może wysyłać komunikaty RPC za pomocą oprogramowania NetBIOS lub Winsock, a w sieciach zarządzanych przez oprogramowanie Lan Manager może je przekazywać za pomocą portów nazwanych. Omówione wcześniej udogodnienie LPC działa podobnie – różnica polega na tym, że komunikaty są przekazywane między dwoma procesami wykonywanymi w tym samym komputerze.

Kodowanie przetaczania i przesyłania argumentów w standardowym formacie, odwrotnego przetaczania i wykonania zdalnej procedury, powrotnego przetaczania i przesyłania wyników oraz ponownego odwrotnego przetaczania i przekazywania wyników w miejsce wywołania jest czynnością nudną i podatną na błędy. Jednak spora część tego kodu może być wygenerowana automatycznie na podstawie prostego opisu argumentów i zwracanych wyników.

System NT dysponuje językiem opisu interfejsu firmy Microsoft^{*} umożliwiającym definiowanie nazw, argumentów oraz wyników procedur zdalnych.

* Microsoft Interface Definition Language – zapewne jest to odmiana języka IDL, pochodzącego używanego w środowisku DCE (Distributed Computing Environment) opracowanego przez konsorcjum OSF. – Przyp. tłum.

Kompilator tego języka wytwarza pliki nagłówkowe z deklaracjami namiastek procedur zdalnych oraz typów danych używanych w ich argumentach i przekazywanych w komunikatach powrotnych. Generuje on również kod źródłowy namiastek procedur używanych po stronie klienta oraz kod odwrotnego przetaczania i delegowania po stronie serwera. W czasie konsolidowania aplikacji następuje dołączenie do niej namiastek procedur. Jeśli aplikacja wywoła procedurę-namiastkę RPC, to wygenerowany kod wykona resztę zadania.

Mechanizm *dynamicznej wymiany danych* (ang. *Dynamic Data Exchange* – DDE) służy do komunikacji międzyprocesowej; opracowano go dla systemu Microsoft Windows. System NT dysponuje jego rozszerzeniem, nazywanym *Network DDE*, które może być stosowane w sieci. Komunikacja odbywa się tutaj za pomocą pary jednokierunkowych potoków.

23.6.3 Zwrotnice i serwery

Aplikacja systemu NT może używać na wejściu i wyjściu interfejsu NT API, za pomocą którego dostęp do plików zdalnych jest osiągany tak samo jak do plików lokalnych, jednak przy założeniu, iż w odległym komputerze działa serwer MS-NET, dostarczany z systemami NT lub Windows for Workgroups. *Zwrotnica* (ang. *redirector*) jest obiektem po stronie klienta, który wysyla w dalszą drogę zamówienia wejścia-wyjścia odnoszące się do odległych plików, aby – po osiągnięciu celu – mogły być zrealizowane przez *serwer*. Ze względu na wydajność i bezpieczeństwo zwrotnice i serwery pracują w trybie jądra.

Szczegóły dostępu do plików zdalnych wyglądają następująco:

- Aplikacja wywołuje zarządcę wejścia-wyjścia, prosząc go o otwarcie pliku i podając mu nazwę pliku w standardowym formacie UNC.
- Zarządcą wejścia-wyjścia buduje pakiet z zamówieniem na operację wejścia-wyjścia, jak to opisaliśmy w p. 23.3.3.6.
- Zarządcą wejścia-wyjścia rozpoznaje, że dostęp dotyczy pliku zdalnego, więc wywołuje moduł sterujący zwany wielotorowym dostawcą uniwersalnej konwencji nazewniczej (ang. *multiple universal-naming-convention provider* – MUP).
- Moduł sterujący MUP wysyla asynchronicznie pakiet z zamówieniem wejścia-wyjścia do wszystkich zarejestrowanych zwrotnic.
- Zwrotnica, która potrafi przeadresować zamówienie, odpowiada modułowi sterującemu MUP. Aby uniknąć zadawania w przyszłości wszystkim zwrotnicom tego samego pytania, moduł sterujący MUP zapamiętuje w pamięci podrzędnej dane o zwrotnicy zdolnej obsłużyć dany plik.

- Zwrotnica przesyła zamówienie sieciowe do systemu zdalnego.
- Moduły obsługi sieci systemu zdalnego odbierają zamówienie i przekazują modułowi sterującemu serwera.
- Moduł sterujący serwera wręcza zamówienie właściwemu, lokalnemu modułowi obsługi systemu plików.
- Następuje wywołanie odpowiedniego modułu obsługi urządzenia w celu uzyskania dostępu do danych.
- Wyniki są zwracane do modułu sterującego serwera, który odsyła dane z powrotem do zwrotnicy, z której nadeszło zamówienie. Zwrotnica przekazuje dane do potrzebującej ich aplikacji za pomocą zarządcy wejścia-wyjścia.

Podobne postępowanie ma miejsce w przypadku aplikacji używających zamiast usług UNC interfejsu sieciowego API podsystemu Win32, jednak wówczas zamiast modułu sterującego MUP jest stosowany moduł noszący nazwę rутera wielodostawczego (ang. *multiprovider router*).

Z uwagi na przenośność zwrotnice i serwery korzystają przy transporcie sieciowym z interfejsu TDI API. Same zamówienia są wyrażane w protokole wyższego poziomu, którym zastępco jest obierany protokół SMB, wymieniony w p. 23.6.1. Wykaz zwrotnic jest utrzymywany w systemowej bazie rejestrowej.

23.6.4 Domeny

Wiele środowisk sieciowych ma naturalne grupy użytkowników, w rodzaju studentów w szkolnym laboratorium komputerowym lub pracowników oddziału jakiejś instytucji. Często zachodzi potrzeba udostępniania wszystkim członkom tak pojмowanej grupy wspólnych zasobów zgromadzonych w należących do niej komputerach. W celu zarządzania globalnymi prawami dostępu w takich grupach wprowadzono w systemie NT pojęcie domeny. (Termin ten nie pozostaje w związku z domenowym systemem nazw DNS, odwzorowującym nazwy komputerów sieci Internet na adresy protokołu IP). Jest zatem *domena NT* grupą stacji roboczych i serwerów, które mają wspólne zasady bezpieczeństwa i wspólną bazę danych użytkowników. Jeden z serwerów domeny pełni rolę podstawowego nadzorcy domeny i odpowiada za bazy danych służące zapewnianiu bezpieczeństwa w domenie. Inne serwery domeny mogą działać jako nadzorcy zapasowi i mogą również dokonywać uwierzytelnień zgłoszeń rejestracji. Stacje robocze domeny obdarzają zaufaniem podstawowego nadzorcę domeny co do poprawności podawanych (za pośrednictwem żetonu dostępu)

informacji o prawach dostępu każdego z użytkowników. Wszyscy użytkownicy zachowują zdolność do ograniczania dostępu do własnych stacji roboczych, niezależnie od punktu widzenia dowolnego z nadzorców domeny.

Przedsiębiorstwo może mieć wiele oddziałów, a szkoła wiele klas, często zachodzi więc potrzeba zarządzania w ramach jednej instytucji wieloma domenami. W tym celu system NT dysponuje czterema modelami domen. W *modelu jednodomenowym* każda domena jest odizolowana od innych. W *modelu domeny głównej* jedna, wyznaczona domena główna jest obdarzana zaufaniem przez wszystkie pozostałe domeny. Utrzymuje ona globalną bazę danych kont użytkowników i świadczy usługi uwierzytelniania na rzecz innych domen. Model ten wprowadza centralną administrację wielu domen. Domena główna może obsłużyć do 40 000 użytkowników. *Model zwielokrotnionej domeny głównej* jest przeznaczony dla wielkich instytucji. Występuje w nim wiele domen głównych mających do siebie wzajemne zaufanie. Dzięki temu zaufaniu użytkownik mający konto w jednej z domen głównych może sięgać po zasoby innej domeny głównej, gdyż domena, w której użytkownik ma konto, może zaświadczać o jego przywilejach. W *modelu zwielokrotnionego zaufania* domena główna nie występuje. Wszystkie domeny obsługują własnych użytkowników, a także wierzą sobie nawzajem.

Gdyby więc użytkownik domeny A spróbował skorzystać z zasobu w domenie B, to interakcja między domeną B a użytkownikiem mogłaby wyglądać, jak następuje. W modelu jednodomenowym domena B powiedziałaby: „Niczego o tobicie nie wiem – idź sobie!”. W modelu domeny głównej domena B mogłaby rzec: „Pozwól mi spytać w domenie głównej, czy masz stosowne pozwolenie.” W modelu zwielokrotnionej domeny głównej domena B mogłaby wyrazić się tak: „Wpierw w mojej domenie głównej spytam, czy którakolwiek z głównych wie, czy prawo masz (czy inoże – nic)”. W modelu zwielokrotnionego zaufania domena B odezwała by się w te słowa: „Niech no się tylko dowiem, czy która z domen powie, że pozwolenie masz”.

23.6.5 Rozbiór nazw w sieciach TCP/IP

W sieci z protokołem IP przez *tłumaczenie* lub *rozbiór nazwy* (ang. *name resolution*) rozumie się proces zamiany nazwy komputera na adres IP, czyli na przykład zamianę nazwy www.bell-labs.com na postać **35.104.1.14**. System NT umożliwia tłumaczenie nazw kilkoma sposobami, takimi jak zastosowanie usług nazewniczych *Windows Internet Name Service* (WINS), rozbiór nazw za pomocą rozmów rozgłoszania, użycie domenowego systemu nazw (DNS), użycie pliku komputerów sieciowych (ang. *host file*) i pliku LMHOSTS. Większość z tych metod jest stosowana w wielu systemach operacyjnych, dlatego poniżej opiszemy tylko usługi WINS.

W systemie WINS dwa serwery (lub większa ich liczba) utrzymują dynamiczną bazę danych z powiązaniemi między nazwami a adresami IP oraz oprogramowanie klienta służące do kierowania pytań do serwerów. Używa się co najmniej dwu serwerów, aby usługi WINS mogły przetrwać awarię serwera i aby obciążenie tłumaczeniem nazw mogło być rozłożone na kilka maszyn.

System WINS stosuje protokół DHCP dynamicznego konfigurowania komputerów sieciowych. Protokół DHCP automatycznie aktualnia konfiguracje adresowe w bazie danych WINS, bez interwencji ze strony użytkownika lub administratora, a dokonuje tego w następujący sposób. Klient rozpoczętyjący wykonanie protokołu DHCP rozgłasza komunikat **discover** (poszukiwanie). Każdy serwer DHCP, który otrzyma ten komunikat, odpowiada za pomocą komunikatu **offer** (oferta), zawierającego adres IP i informacje konfiguracyjne dla klienta. Klient wybiera wówczas jedną z konfiguracji i wysyła do wybranego serwera komunikat **request** (zamówienie). Serwer DHCP przekazuje w odpowiedzi te same informacje o konfiguracji i adresie IP, których udzielił poprzednio wraz z dzierżawą tego adresu. Ta **dzierżawa** (ang. *lease*) uprawnia klienta do posługiwania się danym adresem IP przez określony czas. Gdy czas dzierżawy dobiegnie końca, klient będzie próbował odnowić dzierżawę danego adresu. Jeśli dzierżawa nie ulegnie przedłużeniu, to klient musi ubiegać się o nową.

23.7 ■ Interfejs programowy

Podstawowy dostęp do możliwości systemu NT daje interfejs Win32 API. W tym punkcie opiszemy pięć głównych aspektów interfejsu Win32 API: korzystanie z obiektów jądra, wspólne korzystanie z obiektów przez procesy, zarządzanie procesami, komunikację międzyprocesową i zarządzanie pamięcią.

23.7.1 Dostęp do obiektów jądra

Jądro systemu NT udostępnia programom użytkowym wiele usług. Programy użytkowe korzystają z tych usług za pomocą działań na obiektach jądra. Proses uzyskuje dostęp do obiektu o nazwie **XXX**, wywołując funkcję **CreateXXX** dostarczającą uchwytu (ang. *handle*) do **XXX**. Uchwyt ten jest jednoznaczny w danym procesie. W zależności od rodzaju otwieranego obiektu funkcja **create** może, w przypadku niepowodzenia, zwrócić 0 lub specjalną stałą o nazwie **INVALID_HANDLE_VALUE**. Proses może oddać uchwyt, wywołując funkcję **CloseHandle**, a system może usunąć obiekt, jeśli liczba korzystających z niego procesów spadnie do zera.

23.7.1.1 Wspólne korzystanie z obiektów

System NT umożliwia dzielenie obiektów przez procesy trzema sposobami. Sposób pierwszy polega na dziedziczeniu uchwytu do obiektu przez proces potomny. Wywołując funkcję **CreateXXX**, proces macierzysty podaje strukturę **SECURITIES_ATTRIBUTES** z wartością **TRUE** w polu **bInheritHandle**. To pole powoduje, że uchwyty do obiektu mogą być dziedziczone. Za pomocą funkcji **CreateProcess** można wtedy utworzyć proces potomny z podaniem w jej argumencie **bInheritHandle** wartości **TRUE**. Na rysunku 23.11 widać przykład kodu, który tworzy uchwyty do semafora, dziedziczone w procesie potomnym.

Zakładając, że procesowi potomnemu jest wiadome, które uchwyty podlegają dzieleniu, on i proces macierzysty mogą wymieniać informacje za pomocą wspólnych obiektów. W przykładzie na rys. 23.11 proces potomny otrzymałby wartość uchwytu na podstawie pierwszego argumentu połączenia i mógłby dzielić semafor z procesem, który go utworzył.

Drugi sposób dzielenia obiektów polega na nadaniu obiektowi przez proces nazwy w chwili tworzenia obiektu i zwróceniu się do obiektu za pomocą tej nazwy. Metoda ta ma dwie wady. Jedna wynika z tego, że w systemie NT nie ma sposobu sprawdzenia, czy obiekt o danej nazwie już istnieje. Druga wada wynika z globalności przestrzeni nazw obiektów, niezależnie od typu obiektu. Jest na przykład możliwe, aby dwie aplikacje utworzyły obiekt o nazwie „potok”, choć należałoby utworzyć dwa obiekty o różnych nazwach (i być może odrębne).

Zaletą ponazywanych obiektów jest łatwość wspólnego ich użytkowania przez niepowiązane ze sobą procesy. Jeden z procesów może wywołać któryś

```

...
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa, 1, 1, NULL);
char command_line[132];
ostrstream ostrng(command_line, sizeof(command_line));
ostrng << a_semaphore << ends;
CreateProcess("another_process.exe", command_line, NULL, NULL,
    TRUE, ...);
...

```

Rys. 23.11 Kod umożliwiający procesowi potomnemu dzielenie obiektu przez odziedziczenie uchwytu

```

// proces A
...
Handle a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEMI");
...
// proces B
...
Handle b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, "MySEMI"); ...

```

Rys. 23.12 Kod umożliwiający dzielenie obiektu za pomocą odnalezionej nazwy

z funkcji **CreateXXX** i podać nazwę w parametrze **IpszName**. Drugi proces może pobrać uchwyt do dzielenia tego obiektu, wywołując funkcję **OpenXXX** (lub **CreateXXX**) z taką samą nazwą, jak jest pokazane na rys. 23.12.

Trzeci sposób dzielenia obiektów polega na posłużeniu się funkcją **DuplicateHandle**. Potrzebna jest do tego jeszcze jakaś inna metoda komunikacji międzyprocesowej w celu przekazania podwojonego uchwytu. Proces, któremu udostępniono uchwyt do innego procesu i wartość uchwytu do jakiegoś obiektu w tamtym procesie, może uzyskać kontakt z tym obiektem, czyli dzielić obiekt z innym procesem. Przykład zastosowania tej metody jest przedstawiony na rys. 23.13.

```

...
// Proces A chce udzielić procesowi B dostępu do semafora.
// proces A
Handle a_semaphore = CreateSemaphore(NULL, 1, 1, NULL);
// wysłanie wartości semafora do procesu B za pomocą komunikatu
// lub wspólnej pamięci
...
// proces B
Handle process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
    process_id_of_A);
Handle b_semaphore; DuplicateHandle(process_a, a_semaphore,
    GetCurrentProcess(),
    &b_semaphore, 0, FALSE,
    DUPLICATE_SAME_ACCESS);
// dostęp do semafora za pomocą uchwytu b_semaphore
...

```

Rys. 23.13 Kod umożliwiający dzielenie obiektu za pomocą przekazanego uchwytu

23.7.2 Zarządzanie procesami

W systemie NT przez *proces* rozumie się wykonywany egzemplarz programu użytkowego, a mianem *wątki* określa się jednostkę kodu, dla której system operacyjny planuje przydział procesora. Proces może więc zawierać więcej niż jeden wątek. Proces rozpoczyna się z chwilą, gdy jakiś inny proces wywoła procedurę **CreateProcess**. Procedura ta zapewnia dynamiczne wprowadzanie do pamięci funkcji z dowolnych bibliotek potrzebnych procesowi, jak również tworzy *wątek podstawowy*. Tworzenie dodatkowych wątków jest możliwe za pomocą funkcji **CreateThread**. Każdy nowo tworzony wątek otrzymuje własny stos, którego rozmiar wynosi standardowo 1 MB, chyba że w argumencie funkcji **CreateThread** określono inaczej. Ponieważ niektóre z funkcji systemu wykonawczego języka C utrzymują stan w zmiennych stałych, jak na przykład zmienna *errno*, więc wielowątkowa aplikacja musi być chroniona przed niezsynchronizowanym dostępem. Odpowiedniej synchronizacji dostarcza funkcja osłonowa pod nazwą **beginthreadex**.

Dynamicznie dołączane biblioteki lub pliki wykonywalne, umieszczane w przestrzeni adresowej procesu, są identyfikowane przez uchwyty do ich egzemplarzy. Wartością *uchwytu egzemplarza* (ang. *instance handle*) jest w istocie adres wirtualny umiejscowienia pliku w pamięci. Aplikacja może pobrać uchwyt modułu z własnej przestrzeni adresowej, podając nazwę modułu w wywołaniu funkcji **GetModuleHandle**. Jeśli jako nazwa zostanie podane **NULL**, to nastąpi zwrócenie podstawowego adresu procesu. Dolnych 64 KB przestrzeni adresowej nie używa się, zatem wadliwy program próbujący skorzystać ze wskaźnika **NULL** spowoduje błąd dostępu.

Priorytety w środowisku Win32 wywodzą się z modelu planowania przyjętego w systemie NT, nie można jednak wybierać wszystkich wartości priorytetów. W podsystemie Win32 rozróżnia się cztery klasy priorytetów: **IDLE_PRIORITY_CLASS** (priorytet poziomu 4), **NORMAL_PRIORITY_CLASS** (poziom 8), **HIGH_PRIORITY_CLASS** (poziom 13) i **REALTIME_PRIORITY_CLASS** (poziom 24). Procesy należą zazwyczaj do klasy **NORMAL_PRIORITY_CLASS**, chyba że proces macierzysty był klasy **IDLE_PRIORITY_CLASS** lub w wywołaniu **CreateProcess** określono inną klasę. Klasę priorytetu procesu można zmienić za pomocą funkcji **SetPriorityClass** lub podając ją jako argument polecenia **START**. Na przykład polecenie **START/REALTIME cbserver.exe** spowodowałoby wykonanie programu **cbserver** z klasą priorytetu **REALTIME_PRIORITY_CLASS**. Zauważmy, że do klasy czasu rzeczywistego mogą proces przesuwać tylko użytkownicy o podwyższonym priorytecie planowania przydziału procesora. Administrator systemu oraz specjalni użytkownicy (ang. *Power Users*) mają ten przywilej na mocy ustaleń zastępczych.

System musi zapewniać szczególnie dobre warunki działania programom współdziałającym z użytkownikiem na zasadach dialogu. Z tego powodu system NT stosuje specjalne zasady planowania procesów w klasie priorytetów zwykłych (**NORMAL_PRIORITY_CLASS**). System NT odróżnia *procesy pierwszoplanowe* (ang. *foreground*) – aktualnie powiązane z ekranem – od *procesów drugoplanowych* (ang. *background*), których do działania na ekranie nie wytypowano. Procesowi przechodzącemu do trybu pierwszoplanowego system NT zwiększa przydzielany kwant czasu – zazwyczaj trzykrotnie. (Czynnik ten można zmienić, wybierając sposób działania w części systemowej panelu sterującego). Zwiększenie takie powoduje, że procesy pierwszoplanowe otrzymują do chwili wywłaszczenia trzykrotnie dłuższe odcinki czasu na wykonywanie swoich czynności.

Początkowy priorytet wątku jest określony przez jego klasę, jednak może on być zmieniony za pomocą funkcji **SetThreadPriority**. Przyjmuje ona argument określający priorytet względem podstawowego priorytetu klasy wątku:

- **THREAD_PRIORITY_LOWEST**: priorytet podstawowy - 2;
- **THREAD_PRIORITY_BELOW_NORMAL**: priorytet podstawowy - 1;
- **THREAD_PRIORITY_NORMAL**: priorytet podstawowy;
- **THREAD_PRIORITY_ABOVE_NORMAL**: priorytet podstawowy + 1;
- **THREAD_PRIORITY_HIGHEST**: priorytet podstawowy + 2.

Priorytety mogą być regulowane za pomocą jeszcze dwóch innych oznaćen. Przypomnijmy za punktem 23.3.2, że jądro ma dwie klasy priorytetów: klasę czasu rzeczywistego z priorytetami 16-31 oraz klasę zmiennych priorytetów o wartościach 0-15. Kwalifikator **THREAD_PRIORITY_IDLE** ustawia priorytet wątków działających w czasie rzeczywistym na poziomie 16, a wątki zmiennopriorytetowe zostają sprowadzone do poziomu 1. Kwalifikator **THREAD_PRIORITY_TIME_CRITICAL** ustawia priorytet wątków czasu rzeczywistego na poziomie 31, a wątki zmiennopriorytetowe otrzymują priorytet poziomu 15.

Jak powiedzieliśmy w p. 23.3.2, jądro reguluje priorytet wątku dynamicznie, w zależności od tego, czy wątek jest ograniczony przez wejście-wyjście czy przez jednostkę centralną. Interfejs Win32 API zawiera metodę wyłączenia tej regulacji za pomocą funkcji **SetProcessPriorityBoost** i **SetThreadPriorityBoost**.

Można utworzyć wątek w stanie zawieszenia. Wątek taki nie będzie działać do chwili wybrania go przez inny wątek za pomocą funkcji **ResumeThread**. Funkcja **SuspendThread** ma działanie odwrotne. Funkcje te ak-

tualizują licznik, więc jeśli wątek jest zawieszany dwukrotnie, to przed podjęciem działania musi być dwukrotnie ozywiany.

Do synchronizowania dostępów do obiektów dzielonych przez równolegle działające wątki służą zawarte w jądrze obiekty synchronizujące, takie jak semafory lub zamki. Oprócz tego wątki można synchronizować za pomocą funkcji `WaitForSingleObject` i `WaitForMultipleObject`. Jeszcze inną metodą synchronizacji w interfejsie Win32 API jest sekcja krytyczna. Sekcja krytyczna jest synchronizowaną porcją kodu, który może być wykonywany tylko przez jeden wątek w danej chwili. Wątek ustanawia sekcję krytyczną za pomocą wywołania `InitializeCriticalSection`. Przed wejściem do sekcji krytycznej aplikacja powinna wywołać funkcję `EnterCriticalSection`, a po wyjściu z sekcji krytycznej – funkcję `LeaveCriticalSection`. Te dwie funkcje gwarantują, że w przypadku gdy wiele wątków będzie się ubiegać o wejście do sekcji krytycznej, w danym czasie pozwoli się na jej wykonanie tylko jednego z nich, natomiast inne będą oczekiwane w treści funkcji `EnterCriticalSection`. Mechanizm sekcji krytycznej jest nieco szybszy od jądrowych obiektów synchronizujących.

Włóknem (ang. *fiber*) nazywa się kod działający w trybie użytkownika, dla którego przydział procesora jest planowany według algorytmu zdefiniowanego przez użytkownika. Proces może mieć wiele włókien, tak samo jak może mieć wiele wątków. Główna różnica między wątkami a włóknami polega na tym, że wątki mogą działać równolegle, natomiast nawet w środowisku wieloprocesorowym pozwala się na działanie tylko jednego włókna w danym czasie. Mechanizm ten dołączono do systemu NT, aby umożliwić poprawne przenoszenie tych aplikacji systemu UNIX, które napisano zgodnie z modelem wykonywania włókien.

System tworzy włókno za pomocą funkcji `ConvertThreadToFiber` lub `CreateFiber`. Zasadniczą różnicą między tymi funkcjami jest to, że funkcja `CreateFiber` nie rozpoczyna wykonywania utworzonego włókna. Aby to spowodować, aplikacja musi wywołać funkcję `SwitchToFiber`. Aplikacja może zakończyć włókno, wywołując funkcję `DeleteFiber`.

23.7.3 Komunikacja międzyprocesowa

Jednym ze sposobów, w jaki aplikacje podsystemu Win32 mogą realizować komunikację międzyprocesową, jest wspólne użytkowanie obiektów jądra. Inny sposób polega na przekazywaniu komunikatów – jest on szczególnie popularny w aplikacjach korzystających z interfejsu Windows GUI.

Wątek może wysłać komunikat do innego wątku lub do okna za pomocą któregoś z wywołań: `PostMessage`, `PostThreadMessage`, `SendMessage`, `SendThreadMessage` lub `SendMessageCallback`. Różnica między przeka-

zywaniem komunikatu w trybie pocztowym (ang. *posting a message*) a jego ekspediowaniem w trybie niepocztem (ang. *sending a message*) polega na tym, że procedury pocztowe są asynchroniczne. Zwracają one sterowanie natychmiast i wywołujący je wątek nie jest zorientowany, kiedy naprawdę komunikat zostaje doręczony. Procedury wysyłające typu *Send...* są synchronizowane – blokują nadawcę do czasu doręczenia i przetworzenia komunikatu.

Oprócz wysyłania samego komunikatu wątek może również wysyłać w nim dane. Z uwagi na to, że procesy mają osobne przestrzenie adresowe, dane muszą być kopiowane. System kopiuje je za pomocą wywołania funkcji *SendMessage*, która powoduje przesłanie komunikatu typu **WM_COPYDATA** wraz ze strukturą danych **COPYDATASTRUCT** zawierającą długość i adres przekazywanych danych. Po wysłaniu komunikatu system NT kopiuje dane do nowego bloku pamięci i przekazuje adres wirtualny nowego bloku procesowi odbiorczemu.

W odróżnieniu od środowiska 16-bitowych okien każdy wątek Win32 ma własną kolejkę wejściową, z której odbiera komunikaty. (Całe wejście odbywa się za pomocą komunikatów). Jest to struktura bardziej niezawodna niż wspólna kolejka wejściowa 16-bitowych okien, gdyż przy wielu oddzielnych kolejkach jedna zablokowana aplikacja nie tamuje wejścia innym aplikacjom. Jeśli aplikacja Win32 nie wywołuje funkcji *GetMessage* w celu obsługiwania zdarzeń anonsowanych w jej kolejce wejściowej, to kolejka ta się zapłni i po upływie około 5 sekund system oznaczy aplikację jako „nie odpowiadającą”.

23.7.4 Zarządzanie pamięcią

Interfejs Win32 API umożliwia programom użytkowym kilka sposobów korzystania z pamięci: pamięć wirtualną, pliki odzwierczone w pamięci, sterty oraz lokalną pamięć wątków.

W celu zarezerwowania lub przyznania przydziału pamięci aplikacja wywołuje funkcję *VirtualAlloc*, a w celu unieważnienia przydziału lub zwolenia pamięci korzysta z funkcji *VirtualFree*. Funkcje te umożliwiają aplikacji określenie adresu wirtualnego, od którego pamięć zostanie przydzielona. Działają one na wielokrotnościach stron pamięci, a adres początku przydzielanego obszaru musi być większy niż 0x10000. Przykłady użycia tych funkcji występują na rys. 23.14.

Proces może zablokować w pamięci operacyjnej część z przyznanych mu stron za pomocą wywołania *VirtualLock*. Maksymalna liczba stron, które pozwala się procesowi zablokować, wynosi 30, chyba że proces wywoła najpierw funkcję *SetProcessWorkingSetSize*, aby zwiększyć minimalny rozmiar zbioru roboczego.

```

...  

// przydzielenie 16 MB u szczytu naszej przestrzeni adresowej  

void *buf = VirtualAlloc(0, 0x1000000, MEM_RESERVE  

                        | MEM_TOP_DOWN, PAGE_READWRITE);  

// przyznanie górnego 8 MB przydzielonej przestrzeni  

VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT,  

             PAGE_READWRITE);  

// użytkowanie przydzielonej pamięci  

// unieważnienie przydziału  

VirtualFree(huf + 0x800000, 0x800000, MEM_DECOMMIT);  

// zwolnienie całej przydzielonej przestrzeni  

VirtualFree(huf, 0, MEM_RELEASE);
...

```

Rys. 23.14 Fragmenty kodu przydzielającego pamięć wirtualną

Inny sposób używania pamięci przez aplikację polega na odwzorowywaniu pliku w jej przestrzeni adresowej. Odwzorowywanie w pamięci jest niejednokrotnie wygodną metodą dzielenia pamięci przez dwa procesy – oba odwzorowują ten sam plik w swojej pamięci wirtualnej. Odwzorowywanie pamięci jest wieloetapowe, co można obejrzeć w przykładzie na rys. 23.15.

```

...  

// otwarcie pliku połączone z jego utworzeniem, jeśli nie istnieje  

HANDLE hfile = CreateFile("somefile", GENERIC_READ  

                           | GENERIC_WRITE, FILE_SHARE_READ  

                           | FILE_SHARE_WRITE, NULL,  

                           OPEN_ALWAYS,  

                           FILE_ATTRIBUTES_NORMAL, NULL);  

// utworzenie odwzorowania wielkości 8 MB dla pliku  

HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE,  

                                 SEC_COMMIT, 0, 0x800000,  

                                 "SHM_1");  

// pobranie widoku odwzorowanej przestrzeni  

void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS, 0, 0,  

                          0x800000);  

// użytkowanie pobranego widoku  

// anulowanie odwzorowania pliku  

UnmapViewOfFile(buf);  

CloseHandle(hmap);  

CloseHandle(hfile);
...

```

Rys. 23.15 Fragmenty kodu odwzorowującego plik w pamięci

Jeżeli proces chce odwzorować część przestrzeni adresowej tylko w celu wspólnego użytkowania jakiegoś jej obszaru z innym procesem, to nie potrzeba do tego żadnego pliku. Proces taki może wywołać funkcję **CreateFileMapping** z wartością 0xffffffff w roli uchwytu plikowego i określonym rozmiarem. Wynikowy obiekt odwzorowujący plik może być dzielony za pomocą dziedziczenia, odnajdywania jego nazwy lub przez podwojenie.

Sterta (ang. *heap*) w środowisku Win32 jest po prostu obszarem zarezerwowanej przestrzeni adresowej. Na początku proces Win32 otrzymuje zastępco sterę wielkości 1 MB. Ponieważ tej zastępczej sterły używa wiele funkcji Win32, dostęp do niej podlega synchronizacji w celu ochrony przydzielanych w przestrzeni sterły struktur danych przed uszkodzeniami, do których mogłyby dochodzić wskutek jej współbieżnego aktualizowania przez wiele wątków. Podsystem Win32 zawiera kilka funkcji do zarządzania sterą, dzięki czemu proces może przydzielić sobie prywatną sterę i sprawować nad nią zarząd. Są to funkcje: **HeapCreate**, **HeapAlloc**, **HeapRealloc**, **HeapSize**, **HeapFree** i **HeapDestroy**.

W interfejsie Win32 API istnieje też możliwość wywoływanego funkcji **HeapLock** i **HeapUnlock**, umożliwiających wątkowi dostęp do sterły na prawach wyłączności. W odróżnieniu od **VirtualLock** funkcje te powodują tylko synchronizację – nie blokują one stron w pamięci fizycznej.

Funkcje zależne od danych globalnych lub statycznych zazwyczaj nie działają poprawnie w środowisku wielowątkowym. Na przykład funkcja **strtok** systemu wykonawczego języka C korzysta ze zmiennej statycznej do pamiętania bieżącego położenia w analizowanym napisie. Aby dwa współbieżące wątki mogły wykonywać poprawnie funkcję **strtok**, powinny mieć oddzielne zmienne „bieżącego położenia”. Mechanizm pamięci lokalnej wątku służy do przydzielania pamięci globalnej poszczególnym wątkom. Mechanizm pamięci lokalnej wątku umożliwia tworzenie lokalnej pamięci wątku zarówno w sposób dynamiczny, jak i statyczny. Metodę dynamicznego przydziału ilustruje przykład na rys. 23.16.

```
// zarezerwowanie przegródki na zmienną
DWORD var_index = TlsAlloc();
// nadanie jej jakiejś wartości
TlsSetValue(var_index, 10);
// pobranie tej wartości
int var = TlsGetValue(var_index);
// zwolnienie indeksu
TlsFree(var_index);
```

Rys. 23.16 Kod zarządzający dynamiczną pamięcią lokalną wątku

Aby posłużyć się zmienną statyczną lokalną w wątku, tak aby każdy wątek miał jej własną, prywatną kopię, w programie użytkowym można zamieścić następującą deklarację:

```
_declspec(thread) DWORD eur_pos = 0;
```

23.8 ■ Podsumowanie

Firma Microsoft zaprojektowała system NT jako rozszerzalny, przenośny system operacyjny, który potrafi wykorzystać nowe techniki i nowoczesny sprzęt. System NT umożliwia stosowanie wielu środowisk operacyjnych oraz symetryczne wieloprzetwarzanie. Duża rozmaitość środowisk udostępnianych programom użytkowym przez system NT wynika z zastosowania w jądrze systemu obiektów świadczących podstawowe usługi oraz z zaplecza programowego umożliwiającego obliczenia w trybie klient-serwer. System NT może na przykład wykonywać programy skompilowane dla systemów MS-DOS, Win16, Windows 95, NT i POSIX. System realizuje pamięć wirtualną, ma wbudowaną pamięć podręczną i stosuje planowanie wywłaszczające. Model bezpieczeństwa istniejący w systemie NT jest mocniejszy niż modele zastosowane we wcześniejszych systemach operacyjnych pochodzących z Microsoftu. System zawiera także cechy umożliwiające jego adaptacje międzynarodowe. System NT działa na wielu różnych platformach, więc użytkownicy mogą dokonywać wyborów i ulepszeń swojego sprzętu według własnych możliwości finansowych i zapotrzebowania na wydajność, nie musząc przy tym zmieniać już użytkowanych aplikacji.

■ Ćwiczenia

- 23.1** Rozważ, dlaczego w systemie NT przenoszenie kodu graficznego z trybu użytkownika do trybu jądra mogłoby zmniejszyć niezawodność systemu. W jaki sposób powodowałoby to naruszenie oryginalnych celów projektowych systemu NT?
- 23.2** Zarządcą pamięci wirtualnej systemu NT przydziela pamięć w dwu etapach. Jakie korzyści wynikają z takiego rozwiązania?
- 23.3** Omów zalety i wady którejś ze struktur tablicy stron w systemie NT.
- 23.4** Ile co najwyżej braków stron może wystąpić podczas dostępu wykonywanego za pomocą adresu wirtualnego, a ile w przypadku dostępu z wykorzystaniem dzielonego adresu wirtualnego? Jakie rozwiązania

sprzętowe występują w większości procesorów, aby zmniejszyć te liczby?

- 23.5 Czemu służy prototypowy wpis tablicy stron w systemie NT?
- 23.6 Jakie kroki musi przedsięwziąć zarządcą pamięci podręcznej w celu przekopiowania danych do pamięci podręcznej i na zewnątrz niej?
- 23.7 Jakie problemy pojawiają się podczas wykonywania aplikacji 16-bitowego systemu Windows za pomocą maszyny VDM? Jakie rozwiązania przyjęto w związku z tym w systemie NT? Jakie są ich wady?
- 23.8 Jakie zmiany należałyby wprowadzić do systemu NT, aby umożliwić wykonywanie procesu korzystającego z 64-bitowej przestrzeni adresowej?
- 23.9 System NT ma skoncentrowanego zarządcę pamięci podręcznej. Jakie są zalety i wady takiego zarządcy?
- 23.10 System wejścia-wyjścia NT jest sterowany pakietami. Rozważ argumenty na rzecz zastosowania sterowania pakietami w systemie wejścia-wyjścia oraz argumenty przeciwne takiemu rozwiązaniu.
- 23.11 Wyobraźmy sobie bazę danych wielkości 1 TB utrzymywana w pamięci operacyjnej. Jakie mechanizmy systemu NT można by zastosować przy dostępie do takiej bazy?

Uwagi bibliograficzne

Przegląd systemu NT oraz pewną ilość technicznych szczegółów dotyczących jego wnętrza i składowych zawiera książka Heleny Custer [90]. W książce [91] Custer opisuje bardzo dokładnie system plików NTFS. Przegląd działania i elementów systemu NT zamieszczono w publikacji *Windows NT Resource Kit: Resource Guide* [291]. Jej uaktualnienie odnoszące się do wersji 4.0 systemu NT stanowi książka pt. *Microsoft Windows NT Workstation Resource Kit* [292]. Ivens i in. [197] opisują właściwości systemu Windows NT 4.0. Periodyk *Microsoft Developer Network Library* ukazuje się co kwartał. Dostarcza on obszernych informacji na temat systemu NT i innych produktów firmy Microsoft. Richter [352] omawia szczegółowo pisanie programów używających interfejsu Win32 API. W książce Silberschatza i in. [389] zawarto dobre omówienie B-drzew.

Rozdział 24

PERSPEKTYWA HISTORYCZNA

W rozdziale 1 przedstawiliśmy krótki przegląd historycznego rozwoju systemów operacyjnych. Był on zwięzły i pozbawiony wielu szczegółów, ponieważ omówiliśmy jeszcze wtedy fundamentalnych pojęć systemów operacyjnych (planowania procesora, zarządzania pamięcią, procesów itd.). Teraz jednak rozumiemy już podstawowe koncepcje. Jesteśmy więc w stanie prześledzić, jak zostały one spożytkowane w kilku starszych systemach operacyjnych, wyróżniających się tym, że miały silny wpływ na inne systemy. Niektóre z nich (jak XDS-940 lub system THE) były systemami jedynymi w swoim rodzaju, inne (jak OS/360) znalazły szerokie zastosowanie. Porządek prezentacji podkreśla podobieństwa i różnice tych systemów i nie jest ściśle chronologiczny ani nie odpowiada ich ważności. Jeśli chce się poważnie studiować systemy operacyjne, to powinno się zaznajomić ze wszystkimi tymi systemami.

Rozpoczynamy od ogólnego opisu i zwięzkiej charakterystyki wczesnych systemów. W każdym punkcie podajemy odsyłacze do dalszej lektury. Artykuły napisane przez twórców tych systemów są ważne zarówno ze względu na ich techniczną zawartość, jak i z powodu ich stylu i atmosfery.

24.1 ■ Wczesne systemy operacyjne

Pierwsze komputery były wielkimi (fizycznie) maszynami obsługiwany mi za pośrednictwem konsoli. Programista, który był również operatorem systemu komputerowego, pisał i uruchamiał program bezpośrednio przy konsoli operatorskiej. Najpierw należało program (rozkaz po rozkazie) wprowadzić ręcznie do pamięci, ustawiając przełączniki na płycie czołowej, względnie posłu-

gując się czytnikiem taśmy papierowej lub kart dziurkowanych. Następnie trzeba było wcisnąć odpowiednie przyciski, aby ustawić adres startowy i rozpoczęć wykonywanie programu. Podczas wykonywania programu programista-operator mógł nadzorować jego działanie, obserwując lampki na konsoli. W razie wykrycia błędu programista mógł wstrzymać program, sprawdzić zawartość pamięci oraz rejestrów i wprowadzić poprawki wprost z konsoli. Wyniki były drukowane albo dziurkowane na taśmie papierowej lub kartach w celu późniejszego ich wydrukowania.

W miarę upływu czasu powstawało coraz to więcej oprogramowania i sprzętu. Upowszechniły się czytniki kart, drukarki wierszowe i taśmy magnetyczne. Dla ułatwienia programowania opracowano asemblery, programy ładujące i programy łączące (konsolidatory). Powstały biblioteki typowych funkcji. Można było odtań dołączać do nowych programów funkcje biblioteczne bez konieczności pisania ich za każdym razem od nowa, co dało początek wtórnemu użytkowaniu oprogramowania.

Szczególnie ważne okazały się podprogramy wykonujące operacje wejścia-wyjścia. Każde nowe urządzenie wejścia-wyjścia miało specyficzne cechy i wymagało starannego oprogramowania. Toteż napisano specjalne podprogramy obsługi dla każdego urządzenia wejścia-wyjścia, nazywane *modułami sterującymi urządzeń* (ang. *device driver*). Moduł sterujący urządzenia wie, jak należy używać buforów, znaczników, rejestrów, bitów kontrolnych i bitów stanu danego urządzenia. Każdy odmienny typ urządzenia ma swój własny moduł sterujący. Prosta czynność, na przykład przeczytanie jednego znaku przez czytnik taśmy papierowej, może powodować wykonanie złożonego ciągu operacji zależnych od rodzaju urządzenia. Zamiast pisania tych podprogramów od nowa za każdym razem, niezbędne fragmenty kodu – moduły sterujące urządzeń – po prostu dołączano z biblioteki.

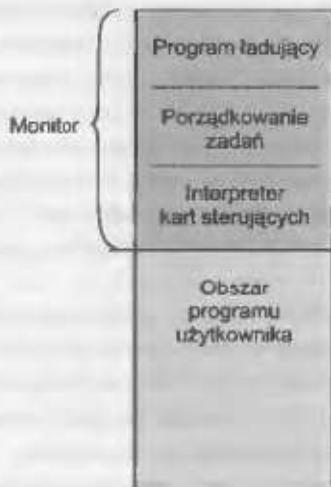
W następnej kolejności pojawiły się kompilatory języków – Fortranu, Cobolu i innych. Programowanie stało się znacznie łatwiejsze, natomiast działanie komputera bardziej skomplikowane. Aby na przykład program napisany w Fortranie przygotować do wykonania, programista musiał najpierw wprowadzić do komputera kompilator tego języka. Kompilator z reguły był przechowywany na taśmie magnetycznej, tak więc należało odpowiednią taśmę zamontować na przewijaku. Później następovalo przeczytanie programu z czytnika kart i zapisanie go na innej taśmie. Kompilator Fortranu wytwarzał na wyjściu przekład programu na język asemblerowy; następnie ten przekład podlegał przetworzeniu przez asembler. To wymagało zamontowania kolejnej taśmy – z asemblerem. Produkt pracy asemblera wymagał jeszcze połączenia z funkcjami bibliotecznymi. W końcu powstawał program wynikowy w postaci kodu w systemie dwójkowym, gotowy do wykonania. Można go było wprowadzić do pamięci i uruchamiać z konsoli operatorskiej, tak jak przedtem.

Zauważmy, że znaczną część czasu wykonania takiego zadania stanowił czas instalowania (ang. *set-up time*). Każde zadanie składało się z wielu oddzielnych kroków: załadowania taśmy z kompilatorem Fortranu, pracy kompilatora, zdemontowania taśmy z kompilatorem, zamontowania taśmy z assemblerem, pracy assemblera, zdjęcia taśmy z assemblerem, załadowania programu wynikowego i jego uruchomienia. Jeśli w którymkolwiek kroku wystąpił błąd, to można było rozpoczęta pracę od nowa. Każdy krok zadania mógł wymagać ładowania i rozładowania taśm magnetycznych, papierowych i czytników kart.

Czas instalowania zadania stanowił poważną trudność. W trakcie montowania taśm lub obsługiwanego przez programistę konsoli jednostka centralna pozostawała bezczynna. Pamiętajmy, że w owych wczesnych latach było bardzo mało komputerów i były one bardzo drogie (kosztowały miliony dolarów). Do tego dochodziły koszty eksploatacji, tj. zasilania, chłodzenia, wynagrodzenia programistów itd. Ponieważ czas komputerów był nader cenny, właściciele dążyli do tego, aby były one używane możliwie intensywnie. Zależało im na osiągnięciu wysokiego stopnia wykorzystania (ang. *utilization*) komputera, aby ich inwestycje przyniosły im możliwie jak największy dochód.

Na rozwiązywanie złożyły się dwa posunięcia. Po pierwsze, zatrudniono zawodowych operatorów. Programista przestał obsługiwać maszynę. Gdy tylko skończyło się jedno zadanie, wówczas operator mógł rozpoczętać następne. Operator, mając więcej wprawy niż programista, szybciej zakładał taśmy na przewijaki, dzięki czemu skrócono czas instalowania zadania. Użytkownik, oprócz kart i taśm, dostarczał krótki opis wykonania zadania. Oczywiście operator nie mógł poprawiać błędnego programu za pośrednictwem konsoli, ponieważ go nie rozumiał. Dlatego w przypadku wystąpienia błędu w programie wyprowadzano zawartość pamięci i rejestrów; na jej podstawie programista mógł szukać przyczyn błędów. Wyprowadzenie zawartości pamięci i rejestrów pozwalało operatorowi na natychmiastowe podejmowanie kolejnego zadania, ale programistę sprawdzającego poprawność programu stawiało w bardziej trudnej sytuacji.

Druga istotna oszczędność polegała również na skróceniu czasu instalowania zadania. Zadania o podobnych wymaganiach gromadzono razem i wykonywano na komputerze grupowo, jako tzw. *wsad* (ang. *batch*). Założymy na przykład, że operator otrzymał jedno zadanie w Fortranie, jedno w Cobolu i znowu jedno w Fortranie. Jeżeli uruchamiałby je w tej samej kolejności, to musiałby najpierw przygotować Fortran (załadować taśmy z kompilatorem itd.), następnie zainstalować Cobol, po czym znów przygotować komputer do pracy w Fortranie. Gdyby zebrał razem i wykonał oba zadania w Fortranie, dokonałby tylko jednego instalowania jego kompilatora, oszczędzając swój czas.



Rys. 24.1 Rozmieszczenie monitora rezydującego w pamięci

Problemy jednak pozostawały nadal. Na przykład o tym, że zadanie się zatrzymywało, operator był informowany za pomocą konsoli. Musiał wtedy rozstrzygnąć, czy program zakończył się poprawnie czy też z powodu błędu, w razie potrzeby wprowadzić zawartość pamięci i rejestrów z chwili zatrzymania programu, a następnie załadować nowe zadanie na odpowiednie urządzenie i wznowić pracę komputera. Podczas przechodzenia od jednego zadania do następnego procesor był bezczynny.

Aby zlikwidować ten rodzaj bezczynności, zastosowano metodę *automatycznego porządkowania zadań* (ang. *automatic job sequencing*), która posłużyła jako podstawa do zbudowania pierwszych, najprostszych systemów operacyjnych. Potrzebna była procedura automatycznego przekazywania sterowania od jednego zadania do następnego. W tym celu ułożono mały program zwany *monitorem rezydentnym* (ang. *resident monitor*) – rys. 24.1. Monitor ten przebywa stale (rezyduje) w pamięci operacyjnej komputera.

Po włączeniu komputera wywoływano monitor rezydentny, który przekazywał sterowanie do programu. Kiedy program kończył działanie, wtedy zwracał sterowanie do monitora, a ten wyznaczał kolejny program. W ten sposób monitor rezydentny zapewniał automatyczne przechodzenie od jednego programu do drugiego i od jednego zadania do następnego.

Powstaje pytanie, skąd monitor rezydentny wiedział, który program ma wykonać? Uprednio operator otrzymywał krótki opis, na podstawie którego wiedział, który program wykonać dla których danych. Aby taką informację można było przekazać bezpośrednio do monitora, wprowadzono *karty sterujące* (ang. *control cards*). Idea była całkiem prosta. Do zadania, oprócz pro-

gramu i danych, programista dołączał specjalne karty (karty sterujące), zawierające dyrektywy dla monitora i wskazujące, który program ma być wykonyany. Na przykład typowe zadanie użytkownika mogło wymagać pracy jednego z trzech programów: kompilatora Fortranu (FTN), asemblera (ASM) i samego programu użytkownika (RUN). Można więc było posłużyć się oddzielną kartą sterującą dla każdego z nich:

\$FTN – uruchomienie kompilatora Fortranu;

\$ASM – uruchomienie asemblera;

\$RUN – uruchomienie programu użytkownika.

Karty te mówiły monitorowi rezydentnemu, jakie programy należało uruchomić.

Do określenia granic każdego zadania użyto dwu dodatkowych kart:

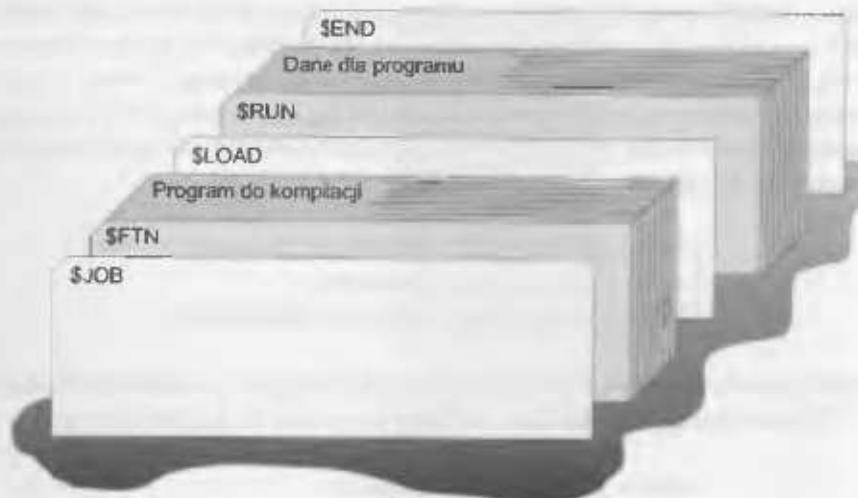
\$JOB – pierwsza karta zadania;

\$END – ostatnia karta zadania.

Te dwie karty mogły być przydatne do rozliczenia programisty ze zużytych zasobów maszyny. Można było na nich uwidocznić parametry z nazwą zadania, numerem rachunku, który należał obciążać opłatą za wykonanie zadania itd. Karty sterujące mogły też służyć do innych celów, na przykład do poproszenia operatora o założenie lub zdjęcie taśmy.

Kwestią do rozwiązania przy zastosowaniu kart sterujących było odróżnienie ich od kart programu. Zwykle wyróżniał je dodany na nich specjalny znak lub wzorzec. W kilku systemach do rozpoznawania kart sterujących posługiwano się znakiem dolara (\$) umieszczanym w pierwszej kolumnie. W innych używano odmiennego kodu. W opracowanym przez firmę IBM języku sterowania zadaniami JCL (ang. *Job Control Language*) stosowano ukośne kreski (/) w pozycjach dwu pierwszych kolumn. Na rysunku 24.2 widać przykład pakietu kart dla prostego systemu wsadowego.

W monitorze rezydentnym można zatem wyróżnić kilka części. Jedną z nich jest *interpreter kart sterujących* (ang. *control-card interpreter*), odpowiedzialny za czytanie i wykonywanie poleceń z kart podczas wykonywania zadania. W odpowiednich chwilach interpreter kart sterujących wywołuje program ładowający, aby załadować do pamięci programy systemowe bądź użytkowe. Program ładowający też jest częścią monitora rezydentnego. Zarówno interpreter kart sterujących, jak i program ładowający wymagają wykonywania operacji wejścia-wyjścia, wobec czego monitor ma jeszcze zestaw modułów sterujących do obsługi urządzeń wejścia-wyjścia. Nierzadko programy systemowe i użytkowe łączy się z tymi samymi modułami sterującymi urządzeń, co zapewnia ciągłość ich działania, oszczędza pamięć i czas programowania.



Rys. 24.2 Pakiet kart dla prostego systemu wsadowego

Tego rodzaju systemy wsadowe spisują się zupełnie nieźle. Monitor rezydentny zapewnia automatyczne następowanie po sobie zadań, zgodnie z kartami sterującymi. Kiedy karta sterująca sygnalizuje konieczność wykonania programu, wtedy monitor wprowadza program do pamięci i przekazuje mu sterowanie. Po zakończeniu pracy program zwraca sterowanie do monitora, który czyta następną kartę sterującą, ładuje odpowiedni program itd. Cykl ten powtarza się dopóty, dopóki nie zostaną wykonane polecenia ze wszystkich kart sterujących w zadaniu. Wtedy monitor automatycznie przechodzi do wykonania kolejnego zadania.

Wprowadzenie systemów z automatycznym porządkowaniem zadań miało na celu poprawienie ich działania. Sprawa jest zupełnie prosta – ludzie są nader powolni (oczywiście w porównaniu z komputerami). Dąży się zatem do zastąpienia działania człowieka przez oprogramowanie systemu operacyjnego. Automatyczne porządkowanie zadań eliminuje udział człowieka w ich instalowaniu i uruchamianiu.

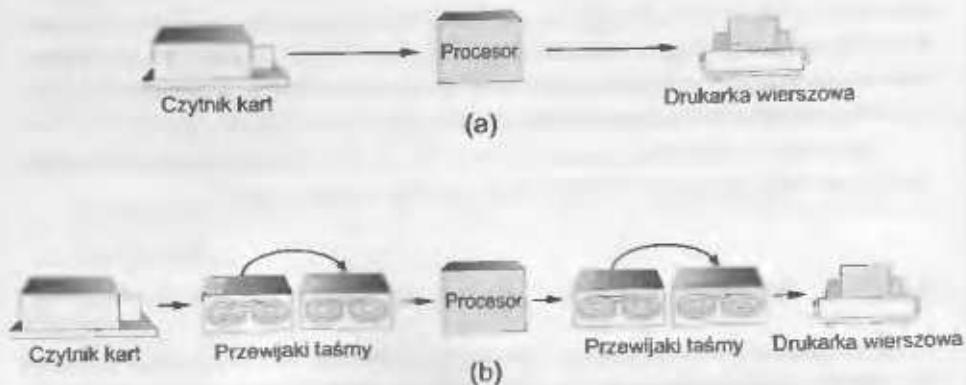
Jednak nawet przy takiej organizacji pracy, jak to podkreśliliśmy w p. 1.2, procesor często był bezczynny. Wynikało to z szybkości mechanicznych urządzeń wejścia-wyjścia, które z natury są powolniejsze od urządzeń elektronicznych. Nawet powolna jednostka centralna pracowała w tempie mikrosekundowym, wykonując tysiące operacji na sekundę. Natomiast szybki czytnik kart może czytać 1200 kart na minutę (17 kart na sekundę). Różnica między szybkością jednostki centralnej a urządzeń wejścia-wyjścia może zatem być większa niż trzy rzędy wielkości. Z czasem, rzecz jasna,

postęp w technologii przyspieszył działanie urządzeń zewnętrznych. Jednakże szybkość procesorów też wzrosła i to nawet bardziej, tak że nie tylko nie rozwiązało to problemu, ale nawet go pogłębiło.

Jednym z powszechnych rozwiązań było zastąpienie bardzo powolnych czytników kart (jako urządzeń wejściowych) i drukarek wierszowych (stosowanych na wyjściu) jednostkami taśmy magnetycznej. Większość systemów komputerowych późnych lat pięćdziesiątych i wczesnych sześćdziesiątych pracowała w trybie wsadowym, używając czytników kart do czytania, a do pisania – drukarek wierszowych lub dziurkarek kart. Jednak zamiast dawać karty bezpośrednio do czytania komputerowi, kopiowano je wpierw na taśmie magnetycznej za pomocą osobnego urządzenia i dopiero gdy taśma była zapelniona, przynoszono ją do komputera. Kiedy program potrzebował danych z karty, wtedy dostarczano mu równoważny rekord z tasmy. Podobnie, wyniki zapisywano na taśmie, a jej zawartość drukowano w późniejszym czasie. Czytnik kart i drukarka wierszowa były obsługiwane w *trybie pośrednim* (ang. *off-line*) – nie przez główny komputer (rys. 24.3).

Podstawową zaletą pracy pośredniej było to, iż komputer główny nie był już ograniczany przez szybkość czytników kart i drukarek wierszowych, ale jedynie przez szybkość działania o wiele szybszych jednostek taśm magnetycznych. Metoda wykonywania wszystkich transmisji za pośrednictwem taśmy magnetycznej mogła być stosowana do wszelkiego tego rodzaju wyposażenia (czytników i dziurkarek kart, ploterów, taśmy papierowej, drukarek).

Faktyczny zysk z pracy pośredniej wynika z możliwości pomnożenia liczby zestawów czytnik-tasma i tasma-drukarka przypadających na jeden procesor. Jeśli jakąś jednostkę centralną przetwarza dane wejściowe dwa razy szybciej, niż może je przeczytać czytnik, to dwa czytniki pracujące jednocześnie wyprodukują wystarczająco dużo tasmy, aby jednostka centralna była



Rys. 24.3 Działanie urządzeń wejścia-wyjścia: (a) bezpośredni; (b) pośredni

przez cały czas zajęta. Z drugiej strony, wydłuża się teraz zwłoka w dostarczeniu kolejnego zadania. Musi ono być najpierw przesłane ma taśmę. Dalsze opóźnienie wynika z umieszczania na taśmie innych zadań w celu jej „zapelnienia”. Taśmę trzeba następnie zwinąć, rozładować, przenieść do jednostki centralnej i zamontować na nie zajętym przewijaku. Systemowi wsadowemu takie postępowanie, co oczywiste, nie przeszkadzi. Na taśmie można zgromadzić wiele podobnych zadań, zanim przeniesie się ją do komputera.

Chociaż pośrednie przygotowywanie zadań na taśmach stosowano przez pewien czas, dość szybko zastąpiono je w większości systemów szeroko upowszechniającymi się systemami dyskowymi, znacznie polepszającymi działania pośrednie. Systemy taśmowe były kłopotliwe, ponieważ nie pozwalały, aby dane z czytnika kart były przesyłane na jeden koniec taśmy, jednocześnie gdy jednostka centralna pobierały inne dane z drugiego końca. Przed czytaniem taśmy musiała zostać w całości zapisana i przewinięta, gdyż taśmy są z natury urządzeniami o *dostępie sekwencyjnym*. W systemach dyskowych, będących urządzeniami o *dostępie swobodnym*, niedogodność ta została usunięta. Główica dysku może się przemieszczać z jednego miejsca dysku do innego, co pozwala błyskawicznie przejść z obszaru dyskowego, w którym zapamiętuje się nowo przeczytane karty, do miejsca, z którego procesor pobiera zawartość „następnej” karty.

W systemie dyskowym zawartość karty z czytnika kart jest przesyłana bezpośrednio na dysk. Rozmieszczenie obrazów kart jest zapisywane w tablicy przechowywanej przez system operacyjny. Podczas wykonywania zadania płynące z niego zamówienia na dane wejściowe z czytnika kart są realizowane przez czytanie z dysku. Podobnie, gdy zadanie zażąda wprowadzenia wiersza na drukarkę, wówczas dany wiersz będzie skopiowany do bufora systemowego i zapisany na dysku. Po zakończeniu zadania jego wyniki są oczywiście drukowane. Taka forma przetwarzania nosi nazwę *spoolingu* – omówiliśmy ją w p. 1.2. Spooling w istocie polega na tym, że używa się dysku jak bardzo wielkiego bufora do czytania danych z maksymalnym wyprzedzeniem z urządzeń wejściowych oraz do przechowywania plików wyjściowych do czasu, aż urządzenia wyjściowe będą w stanie je przyjąć.

Spooling w naturalny sposób prowadzi do wieloprogramowości będącej podstawą wszystkich nowoczesnych systemów operacyjnych.

24.2 ■ System Atlas

System operacyjny Atlas, opisany w pracach Kilburna i in. [213] oraz Howartha i in. [182], powstał w University of Manchester w Anglii na przełomie lat pięćdziesiątych i sześćdziesiątych. Wiele jego podstawowych cech, które były

w owym czasie nowatorskie, stało się standardowymi elementami współczesnych systemów operacyjnych. Moduły sterujące urządzeń stanowiły główną część oprogramowania systemu. Wprowadzono też funkcje systemowe w postaci zbioru specjalnych rozkazów zwanych *ekstrakodami* (ang. *extra codes*).

Atlas był systemem wsadowym, wykorzystującym spooling. Spooling pozwolił systemowi na planowanie zadań zgodnie z możliwością dostępu do urządzeń zewnętrznych, takich jak taśmy magnetyczne, czytniki i dziurarki taśmy papierowej, drukarki wierszowe, czytniki lub dziurarki kart.

Najbardziej znamienitą cechą systemu Atlas okazało się jednak jego zarządzanie pamięcią. Pamięć rdzeniowa była podówczas nowością, przy tym drogą. Wiele komputerów, jak na przykład IBM 650, używało bębna jako pamięci podstawowej. W systemie Atlas bęben pełnił rolę pamięci głównej, lecz istniała niewielka pamięć rdzeniowa służąca jako pamięć podręczna dla bębna. Do automatycznego przesyłania danych między pamięcią rdzeniową i bębnem zastosowano stronicowanie na żądanie.

System Atlas pracował na brytyjskim komputerze z 48-bitowymi słowami. Adresy miały 24 bity, lecz były kodowane dziesiętnie, co pozwalało na zaadresowanie tylko 1 miliona słów. W tamtych czasach była to bardzo rozległa przestrzeń adresowa. Fizyczna pamięć systemu Atlas składała się z bębna wielkości 98 K słów i 16 K słów pamięci rdzeniowej. Pamięć została podzielona na 512-słowowe strony, dając 32 ramki w pamięci fizycznej. Złożona z 32 rejestrów pamięć asocjacyjna implementowała odwzorowanie adresu wirtualnego na adres fizyczny.

Wystąpienie braku strony powodowało uaktywnienie algorytmu zastępowania stron. Jedna ramka pamięci zawsze pozostawała pusta, aby przesłanie z bębna mogło rozpoczynać się natychmiast. Algorytm zastępowania usiłował przewidzieć ciąg przyszłych odniesień do pamięci na podstawie ciągu minionych zdarzeń. Przy każdym dostępie do ramki następowało ustalenie jej bitu odniesienia. Bit odniesienia czytano do pamięci co każde 1024 wykonane rozkazy i zachowywano ostatnie 32 ich wartości. Służyły one do określania czasu ostatniego odwołania (t_1) i upływu czasu między dwoma ostatnimi odwołaniami (t_2). Strony do zastępowania wybierano w porządku określonym przez następujące kryteria:

1. Wybierano każdą stronę, dla której $t_1 > t_2 + 1$. Taką stronę uważano za bezczynną.
2. Jeśli $t_1 \leq t_2$ dla wszystkich stron, to zastępowano stronę z największą różnicą $t_1 - t_2$.

Algorytm zastępowania stron zakładał, że programy potrzebują dostępu do pamięci w pętlach. Jeśli czas między dwoma ostatnimi odwołaniami do pa-

mięci wynosił t_2 , to kolejnego odwołania oczekiwano po upływie następnych t_2 jednostek. Jeśli odwołanie nie następowało ($t_1 > t_2$), to zakładano, że strona nie była używana i zmieniano ją. Jeśli wszystkie strony były wciąż w użyciu, to zastępowano tę stronę, która najdłużej miała być niepotrzebna. Zakładano, że czas do następnego odniesienia określa różnicę $t_2 - t_1$.

24.3 ■ System XDS-940

System operacyjny XDS-940, opisany przez Lichtenbergera i Pirtle'a [255], zaprojektowano w University of California w Berkeley. Podobnie jak w systemie Atlas, w systemie XDS-940 stosowano stronicowanie do zarządzania pamięcią. W przeciwienstwie zaś do systemu Atlas, XDS-940 był systemem z podziałem czasu.

Stronicowanie służyło tylko do przemieszczania, a nie do sprawdzania stron na żądanie. Pamięć wirtualna procesu każdego użytkownika wynosiła tylko 16 K słów, a pamięć fizyczna miała 64 K słów. Każda ze stron miała rozmiar 2 K. Tablicę stron przechowywano w rejestrach. Ponieważ pamięć fizyczna była większa niż pamięć wirtualna, w tym samym czasie mogło przebywać w pamięci kilka procesów użytkowych. Liczbę użytkowników można było zwiększać, stosując dzielenie tych stron, które zawierały kod przeznaczony tylko do czytania i wznawialny. Procesy przechowywano na bieżnie i wymieniano z pamięcią zależnie od potrzeb.

System XDS-940 został skonstruowany w wyniku modyfikacji systemu XDS-930. Modyfikacje były spowodowane typowymi zmianami w podstawowym komputerze, wprowadzonymi po to, aby można było system operacyjny napisać w sposób właściwy. Dodano tryby użytkownika i monitora. Pewne rozkazy, takie jak operacje wejścia-wyjścia oraz rozkaz zatrzymania (ang. *halt*), określono jako uprzywilejowane. Próba wykonania rozkazu uprzywilejowanego w trybie użytkownika kończyła się interwencją systemu operacyjnego.

Do zbioru rozkazów trybu użytkownika dodano rozkaz wywołania systemu. Używano tego rozkazu do tworzenia nowych zasobów, takich jak pliki, umożliwiając systemowi operacyjnemu zarządzanie zasobami fizycznymi. Na przykład pliki były przydzielane w 256-słownowych blokach na bieżnie. Do zarządzania wolnymi blokami bieżna zastosowano mapę bitową. Każdy plik miał blok indeksowy ze wskaźnikami do rzeczywistych bloków danych. Bloki indeksowe były powiązane w łańcuchy.

System XDS-940 zawierał także funkcje umożliwiające procesom tworzenie, inicjowanie, zawieszanie i likwidowanie podprocesów. Programista-użytkownik mógł budować systemy procesów. Osobne procesy mogły dzie-

lic pamięć w celach komunikacji i synchronizacji. Mechanizm tworzenia procesów definiował strukturę drzewiastą, w której proces stawał się korzeniem, a jego podprocesy stanowiły węzły umieszczone w niższych partiach drzewa. Każdy z podprocesów mógł znowu tworzyć następne podprocesy.

24.4 ■ System THE

System operacyjny THE, opisany przez Dijkstrę [112] oraz McKeaga i Wilsona [275] (zob. rozdz. 3), opracowano w Technische Hogeschool w Eindhoven, w Holandii. Był to system wsadowy, pracujący na holenderskim komputerze EL X8, zawierającym 32 K słów 27-bitowych. System był znany głównie z powodu przejrzystego projektu, zwłaszcza struktury warstwowej, oraz z używanych w nim procesów współbieżnych korzystających z semaforów do synchronizacji.

W odróżnieniu od systemu XDS-940, zbiór procesów w systemie THE był statyczny. Sam system operacyjny został zaprojektowany jako zbiór współpracujących procesów. W dodatku utworzono pięć procesów użytkowych, które służyły jako aktywne agendy kompilowania, wykonywania i drukowania programów użytkowników. Po zakończeniu jednego zadania proces zwracał się do kolejki wejściowej w celu wybrania następnego zadania.

W systemie zastosowano priorytetowy algorytm planowania przydziału procesora. Priorytety przeliczano co 2 s, przy czym były one odwrotnie proporcjonalne do ostatnio zużytego czasu procesora (w ciągu minionych 8–10 s). Ten schemat dawał wyższy priorytet procesom zależnym od wejścia-wyjścia oraz procesom nowym.

Zarządzanie pamięcią było ograniczone z powodu braku środków sprzętowych. Niemniej jednak, ponieważ system był ograniczony, a programy użytkowników mogły być napisane tylko w Algolu, zastosowano schemat stronicowania programowego. Kompilator Algolu automatycznie generował wywołania procedur systemowych; gwarantowało to obecność żądanej informacji w pamięci i w razie potrzeby prowadziło do wymian. Pamięcią pomocniczą był bęben o pojemności 512 K. Rozmiar strony wynosił 512 słów, a zastępowanie stron regulował algorytm LRU.

Inną ważną sprawą, na której skoncentrowano się w systemie THE, było kontrolowanie zakleszczeń. W celu unikania zakleszczeń użyto algorytmu bankiera.

Z systemem THE jest blisko związany system Venus omówiony przez Liskova w artykule [259]. Projekt systemu Venus miał również strukturę warstwową i używał semaforów do synchronizacji procesów. Jednakże niskie poziomy projektu zaimplementowano za pomocą mikroprogramu, uzyskując

w efekcie system znacznie szybszy. Zarządzanie pamięcią zostało zmienione na stronicowaną segmentację pamięci. W systemie Venus wprowadzono podział czasu zamiast pracy wsadowej.

24.5 ■ System RC 4000

System RC 4000, podobnie jak system THE, jest znany głównie z powodu donioskich koncepcji projektowych. Został opracowany dla duńskiego komputera RC 4000 przez Regnecentralen, a w szczególności przez Brincha Hansena, który opisał ten system w pracach [54], [56] (zob. też rozdz. 8). Zadanie, które sobie postawiono, nie polegało na skonstruowaniu systemu wsadowego czy z podziałem czasu, czy też jakiegokolwiek innego, specyficznego systemu. Zamiast tego celem stało się opracowanie rdzenia – albo jądra – systemu operacyjnego, na podstawie którego można by zbudować pełny system operacyjny. Toteż struktura systemu przyjęła formę warstwową i wykonano tylko niższe poziomy – jądro.

Jądro stanowiło zbiór procesów współbieżnych. Dostęp procesów do procesora wyznaczał planista działający zgodnie z algorytmem rotacyjnym. Choć procesy mogły dzielić pamięć, podstawowym mechanizmem komunikacji i synchronizacji był *system komunikatów* (ang. *message system*) realizowany przez jądro. Procesy mogły komunikować się ze sobą, wymieniając stałej długości komunikaty zawierające po 8 słów. Wszystkie komunikaty przechowywano w buforach ze wspólnej puli buforów. Gdy bufor komunikatu przedstawiał być potrzebny, wówczas zwracano go do wspólnej puli.

Z każdym procesem była związana *kolejka komunikatów* (ang. *message queue*). Zawierała ona wszystkie komunikaty wysłane do danego procesu i jeszcze nie odebrane. Komunikaty usuwano z kolejki w porządku FIFO. System pozwalał posługiwać się czterema elementarnymi operacjami, wykonywanymi niepodzielnie:

- **send_message (in odbiorca, in komunikat, out bufor);**
- **wait_message (out nadawca, out komunikat, out bufor);**
- **send_answer (out wynik, in komunikat, in bufor);**
- **wait_answer (out wynik, out komunikat, in bufor).**

Dwie ostatnie operacje pozwalały procesom wymieniać kilka komunikatów jednocześnie.

Operacje te wymagały, aby proces obsługiwał swoją kolejkę komunikatów w porządku FIFO oraz aby się blokował, gdy inne procesy obsługiwały

swoje komunikaty. W celu usunięcia tego ograniczenia projektanci dodali dwie elementarne operacje komunikacji. Pozwoliły procesowi oczekwać na nadjście następnego komunikatu lub odpowiadać i obsługiwać swoją kolejkę w dowolnym porządku:

- **wait_event (in poprzedni-bufor, out następny-bufor, out wynik);**
- **get_event (out bufor).**

Urządzenia wejścia-wyjścia również potraktowano tak jak procesy. Mogały sterujące urządzenia zamieniały przerwania nadchodzące od urządzeń i stany ich rejestrów na komunikaty. W ten sposób proces mógł pisać na terminalu, wysyłając do niego komunikat. Moduł sterujący urządzenia odbierał komunikat i wyprowadzał znak na terminal. Znak wejściowy generował przerwanie w systemie, powodujące przejście do modułu sterującego urządzenia. Moduł sterujący urządzenia tworzył komunikat ze znaku wejściowego i posyłał go do oczekującego procesu.

24.6 ■ System CTSS

System CTSS (*Compatible Time-Sharing System*), opisany w pracy Corbato i in. [86], został zaprojektowany w instytucie MIT jako eksperymentalny system z podziałem czasu. Zrealizowano go na komputerze IBM 7090, w którym umożliwiał pracę konwersacyjną z udziałem co najwyżej 32 użytkowników. Użytkownicy mieli do dyspozycji zestaw poleceń interakcyjnych, dzięki którym mogli manipulować plikami oraz kompilować i wykonywać programy za pośrednictwem terminali.

Komputer IBM 7090 miał 32 K pamięci tworzonej z 36-bitowych słów. Monitor zużywał 5 K słów, pozostawiając 27 K dla użytkowników. Obrazy pamięci użytkowników podlegały wymianie między pamięcią główną a szybkim bębnem. W planowaniu procesora zastosowano algorytm wielopoziomowych kolejek ze sprzężeniem zwrotnym. Kwant czasu dla poziomu 1 wynosił 2^7 jednostek. Jeśli program niekończył swojej fazy procesora w jednym kwancie czasu, to przesuwano go w dół, na następny poziom kolejki, dając dwukrotnie więcej czasu. Najpierw był wykonywany program na najwyższym poziomie (z najkrótszym kwantem czasu). Początkowy poziom programu zależał od jego rozmiaru, tak by przydzielony mu kwant czasu był przynajmniej tak długi jak czas jego wymiany.

System CTSS cenił się wielkim sukcesem i pozostawał w użyciu do 1972 r. Choć był dość ograniczony, udało się za jego pomocą zademonstrować, że

podział czasu jest wygodnym i praktycznym modelem obliczeniowym. Powstanie systemu CTSS wpłynęło na wzmożony rozwój systemów z podziałem czasu, przyczyniło się też do podjęcia prac nad systemem MULTICS.

24.7 ■ System MULTICS

System MULTICS opracowano w instytucie MIT (zob. prace: Corbato i Vysotsky [85], Organick [317]) jako naturalne rozszerzenie systemu CTSS. System CTSS i inne wczesne systemy z podziałem czasu zyskały takie powodzenie, że natychmiast spowodowały zapotrzebowanie na szybkie przejście do większych i lepszych opracowań tego rodzaju. Z chwilą udostępnienia większych komputerów konstruktorzy systemu CTSS postanowili wprowadzić udogodnienie w postaci podziału czasu. Usługi komputerowe miały być dostarczane jak prąd elektryczny. Wielkie systemy komputerowe powinny być połączone liniami telefonicznymi z terminalami w biurach i domach w całym mieście. System operacyjny pracowałby nieprzerwanie w trybie podziału czasu z rozległym systemem plików wspólnie użytkowanych programów i danych.

System MULTICS został zaprojektowany przez zespół złożony z pracowników instytutu MIT, firmy General Electric (której oddział komputerowy został później sprzedany firmie Honeywell) i firmy Bell Laboratory (która zaniechła tego projektu w 1969 r.). Podstawowy komputer GE 635 przerobiono na nowy system komputerowy GE 645, przy czym modyfikacja polegała głównie na dodaniu sprzętu realizującego stronicowaną segmentację pamięci.

Adres wirtualny składał się z 18-bitowego numeru segmentu i 16-bitowej odległości słowa od początku segmentu. Segmente podlegały następnie podziałowi na strony wielkości 1 K. Do zastępowania stron zastosowano algorytm drugiej szansy.

Segmentowana, wirtualna przestrzeń adresowa została połączona z systemem plików – każdy segment był plikiem. Segmente adresowano za pomocą nazw plików. Sam system plików miał wielopoziomową strukturę drzewiastą, pozwalającą użytkownikom na tworzenie własnych struktur podkatalogów.

Podobnie jak w CTSS, w systemie MULTICS stosowano do planowania przydziału procesora wielopoziomową kolejkę ze sprzężeniem zwrotnym. Ochronę zrealizowano przez przyporządkowanie każdemu plikowi listy dostępów oraz zastosowanie zbioru pierścieni ochrony przy wykonywaniu procesów. System, który został niemal w całości napisany w języku PL/I, zajmował około 300 000 wierszy kodu. Rozszerzono go na system wieloprocesorowy, pozwalając na odłączanie procesorów w celach pielęgnacyjnych, podczas gdy system kontynuował działanie.

24.8 ■ System OS/360

Najdłuższą drogę rozwoju systemy operacyjne przeszły niewątpliwie na komputerach firmy IBM. Wczesne komputery tej firmy, takie jak IBM 7090 i IBM 7094, są pierwszymi przykładami rozwoju typowych procedur wejścia-wyjścia, monitora rezydentnego, rozkazów uprzywilejowanych, ochrony pamięci i prostego przetwarzania wsadowego. Systemy te rozwijano oddzielnie, często niezależnie w każdej instalacji. W rezultacie firma IBM stanęła w obliczu wielu różnych komputerów, wielu różnych języków i zróżnicowanego oprogramowania systemowego.

Aby zmienić tę sytuację, zaprojektowano system IBM/360, który tworzył rodzinę komputerów obejmującą cały zakres od małych maszyn biurowych do wielkich jednostek przeznaczonych do obliczeń naukowych. Systemy te posługiwały się tylko jednym zestawem oprogramowania, w szczególności – takim samym systemem operacyjnym, zwanym OS/360 i opisany w artykule Mealy'ego i in. [281]. Takie postawienie sprawy oznaczało dla firmy IBM zmniejszenie problemów eksploracji: pozwoliło użytkownikom na swobodne przenoszenie programów i aplikacji z jednego systemu IBM na inny.

Niestety, chciano, by system OS/360 robił wszystko dla wszystkich. W efekcie, z żadnego ze swoich zadań nie wywiązywał się nazbyt dobrze. System plików zawierał pole typu, definiujące typ każdego pliku. Pliki o stałej lub zmiennej długości rekordów, jak również blokowane lub o informacji nie grupowanej w bloki, miały odmienne typy. Zastosowano przydział ciągły, tak że użytkownik musiał z góry przewidywać rozmiar każdego pliku. W języku opisu zadań (JCL) dodano parametry dla każdej możliwej opcji, czyniąc go niezrozumiałym dla przeciętnego użytkownika.

Procedury zarządzania pamięcią były ograniczane przez architekturę sprzętu. Choć stosowano tryb adresowania z rejestrów bazowym, program mógł uzyskać dostęp do tego rejestrów i zmienić go tak, aby procesor generował adresy bezwzględne. Ta organizacja uniemożliwiła dynamiczne przemieszczenia; program ograniczano do rozmiaru pamięci fizycznej podczas ładowania. Powstały dwie oddzielne wersje tego systemu operacyjnego: OS/MFT stosująca stałe regiony pamięci i OS/MVT o regionach zmiennych.

System pisało w języku asemblerowym tysiące programistów – w wyniku powstały miliony wierszy kodu. Sam system operacyjny potrzebował wielkiej ilości pamięci na swój kod i tablice. Eksploracja systemu pochłaniała często połowę cykli procesora. Po latach ukazały się nowe wersje, napisane w celu dodania nowych właściwości i zlokalizowania błędów. Jednak usunięcie jednego błędu często powodowało powstanie innego błędu w odległej części systemu, tak więc liczba znanych błędów systemu utrzymywała się praktycznie na stałym poziomie.

Wraz ze zmianą architektury sprzętowej powstał system IBM 370, w którym system OS/360 zaopatrzono w pamięć wirtualną. Sprzęt umożliwiał segmentowaną-stronicowaną pamięć wirtualną. Nowe wersje OS korzystały z tego sprzętu na różne sposoby. System OS/VS1 tworzył jedną wielką przestrzeń adresów wirtualnych i wykonywał w tej pamięci system OS/MFT. Tak więc stronicowaniu podlegał sam system operacyjny na równi z programami użytkowymi. W wydaniu 1 systemu OS/VS2 w pamięci wirtualnej pracował system OS/MVT. Ostatecznie w wydaniu 2 systemu OS/VS2, obecnie nazywanym MVS, przydzielano każdemu użytkownikowi jego pamięć wirtualną.

System MVS nadal jest w zasadzie wsadowym systemem operacyjnym. Na komputerze IBM 7094 pracował system CTSS, ale w instytucie MIT uznano, że przestrzeń adresowa komputerów IBM 360, następców wersji 7094, była za mała dla systemu MULTICS, toteż zmieniono dostawców sprzętu. Firma IBM postanowiła wówczas zbudować własny system z podziałem czasu – TSS/360 opisany w pracy Letta i Konigsforda [249]. Podobnie jak MULTICS, system TSS/360 pomyślano jako dużej skali narzędzie podziału czasu. Podstawowa architektura sprzętowa komputera IBM 360 została ulepszona w modelu 67 przez umożliwienie stosowania pamięci wirtualnej. W reakcji na powstanie systemu TSS/360 w kilku instalacjach zakupiono model IBM 360/67.

Jednak prace nad systemem TSS/360 opóźniały się, toteż równolegle powstały inne systemy z podziałem czasu, które miały służyć doraźnie do chwili udostępnienia TSS/360. Do systemu OS/360 dodano możliwość podziału czasu (opcja TSO). W ośrodku IBM Cambridge Scientific Center opracowano system CMS przeznaczony dla indywidualnych użytkowników oraz system CP/67 realizujący dla nich maszyny wirtualne (zob. artykuły Meyera i Seawrighta [286] oraz Parmelego i in. [321]).

Cdy system TSS/360 ujrzał wreszcie światło dzienne, doznał porażki. Był za obszerny i za powolny. W rezultacie żadna z instalacji nie przeszła z tymczasowo użytkowanego systemu na system TSS/360. Obecnie podział czasu w systemach IBM istnieje głównie przy udziale systemu TSO pracującego pod systemem MVS albo systemu CMS wykonywanego pod kontrolą systemu CP/67 (przemianowanego na VM).

Co było złego w systemach TSS/360 i MULTICS? Część kłopotów wynikała stąd, że te zaawansowane systemy okazały się zbyt wielkie i skomplikowane, aby można było je zrozumieć. Inny problem spowodował założenie, że moc obliczeniowa będzie uzyskiwana z wielkiego, zdalnie położonego komputera na zasadzie podziału czasu. Współcześnie okazuje się, że większość obliczeń będzie wykonywana przez małe, indywidualne maszyny – komputery osobiste, a nie przez wielkie, zdalne systemy z podziałem czasu, próbujące być wszystkim dla wszystkich użytkowników.

24.9 ■ System Mach

System operacyjny Mach wywodzi się z systemu operacyjnego Accent, opracowanego w Carnegie-Mellon University (CMU) (zob. Raschid i Robertson [343]). Zasady komunikacji przyjęte w systemie Mach i jego filozofia pochodzą z systemu Accent, lecz wiele innych, istotnych części systemu (np. system pamięci wirtualnej, zarządzanie zadaniami i wątkami) opracowano od podstaw (zob. Raschid [342], Tevanian i Smith [424] oraz Accetta i in. [2]). Planistę systemu Mach opisują szczegółowo Tevanian i in. [422] oraz Black [41]. Wczesną wersję pamięci dzielonej w systemie Mach oraz systemu odzworowywania pamięci zaprezentowano w raporcie Tevaniana i in. [422].

Projektantom systemu operacyjnego Mach przyświecały trzy zasadnicze cele:

- Emulowanie systemu UNIX 4.3BSD, tak aby pliki wykonywalne w systemie UNIX można było poprawnie wykonywać pod nadzorem systemu Mach.
- Uzyskanie nowoczesnego systemu operacyjnego, rozporządzającego wieloma modelami pamięci oraz obliczeniami równoległymi i rozproszonymi.
- Otrzymanie jądra, które byłoby prostsze i łatwiejsze do modyfikacji niż jądro systemu 4.3BSD.

Podczas swojego rozwoju system Mach przebył ewolucyjną drogę rozpoczęającą się od systemów UNIX BSD. Kod systemu Mach opracowywano początkowo w obrębie jądra systemu 4.2BSD, w którym oryginalne elementy systemu BSD stopniowo zastępowano elementami systemu Mach, w miarę ich powstawania. Te elementy systemu BSD zostały uaktualnione po pojawieniu się systemu 4.3BSD. Do 1986 r. podsystemy pamięci wirtualnej i komunikacji pracowały na komputerach rodziny DEC VAX, łącznie z ich wersjami wieloprocesorowymi. Wkrótce potem pojawiły się wersje dla komputerów IBM RT/PC oraz dla stacji roboczych Sun 3. W 1987 r. ukończono wersje systemu dla wieloprocesorów Encore Multimax oraz Sequent Balance, zawierające oprogramowanie zadań i wątków; ukazały się również pierwsze oficjalne wydania systemu: Wydanie 0 i Wydanie 1.

Do Wydania 2 system Mach zapewniał zgodność z odpowiednimi systemami BSD, zawierając w jądrze większość kodu systemu BSD. Z powodu nowych cech i możliwości jądra tych wydań były większe niż odpowiadające im jądra systemów BSD. W systemie Mach 3 wyodrębniono z jądra kod BSD, tworząc w ten sposób znacznie mniejsze *mikrojądro*. W tym jądrze zrealizo-

wano tylko podstawowe właściwości systemu Mach – cały kod specyficzny dla systemu UNIX został wyniesiony na poziom serwerów pracujących w trybie użytkownika. Wyłączenie z jądra kodu specyficznego dla systemu UNIX pozwoliło na zastąpienie systemu BSD innym systemem operacyjnym lub na jednoczesne wykonywanie na szczytce jądra interfejsów wielu systemów operacyjnych. Oprócz implementacji systemu BSD na poziomie użytkowym zrealizowano systemy operacyjne DOS, Macintosh i OSF/1. Było to postępowanie podobne do koncepcji maszyny wirtualnej, lecz maszynę wirtualną zdefiniowano tu za pomocą oprogramowania (interfejs jądra systemu Mach) zamiast przez sprzęt. Od wersji 3.0 system Mach stał się szeroko dostępny na różnorodnym sprzęcie, obejmującym jednoprocesorowe maszyny Sun, Intel, IBM i DEC oraz systemy wieloprocesorowe DEC, Sequent i Encore.

Impuls do zainteresowania się czołowych producentów systemem Mach dalo ogłoszenie w 1989 r. przez konsorcjum Open System Foundation (OSF), że system Mach 2.5 będzie podstawą nowego systemu operacyjnego OSF/1. Pierwsze wydanie systemu OSF/1 pojawiło się w rok później i konkuruje obecnie z czwartym wydaniem systemu operacyjnego UNIX System V, które z kolei upodobali sobie członkowie stowarzyszenia *UNIX International* (UI). W skład konsorcjum OSF weszły przodujące technologicznie firmy, takie jak IBM, DEC i HP. Od tego czasu plany OSF uległy zmianie i tylko system UNIX firmy DEC ma za podstawę jądro Mach.

System Mach 2.5 jest także podstawą systemu operacyjnego stacji roboczej *NeXT* zrodzonej w umyśle Steve'a Jobsa ze słynnej firmy Apple Computer.

W odróżnieniu od systemu UNIX, który opracowano nie biorąc pod uwagę wieloprzetwarzania, system Mach jest pod każdym względem dostosowany do pracy na wieloprocesorach. Jego zdolności do wieloprzetwarzania są poza tym wyjątkowo elastyczne, poczynając od systemów z pamięcią dzieloną aż po systemy, których procesory nie dzielą żadnej pamięci. W systemie Mach używa się procesów lekkich, które przybierają postać wielu wątków wykonywanych w ramach jednego zadania (czyli przestrzeni adresowej), co umożliwia obliczenia wieloprocesorowe i równoległe. Szerokie zastosowanie komunikatów jako jednej metody łączności zapewnia kompletność i wydajność mechanizmów ochrony. Przez połączenie komunikatów i systemu pamięci wirtualnej uzyskano w systemie Mach wydajną obsługę komunikatów. Wreszcie, dzięki użyciu komunikatów jako środka łączności systemu pamięci wirtualnej z demonami zarządzającymi pamięcią pomocniczą osiągnięto w systemie Mach wielką elastyczność projektowania i realizacji owszych zadań zarządzania obiekttami pamięci. Niski, elementarny poziom wywołań systemowych umożliwia budowanie z nich bardziej złożonych funkcji, a jednocześnie zmniejsza rozmiary jądra i pozwala na emulowanie systemów operacyjnych na poziomie użytkownika na podobieństwo systemów maszyn wirtualnych IBM.

Poprzednie wydania *Podstawa systemów operacyjnych* zawierały cały rozdział o systemie Mach. Rozdział ten, w formie jaką przyjął w czwartym wydaniu*, jest dostępny w sieci Internet pod adresem URL:<http://www.bell-labs.com/topic/books/os-book/Mach.ps>.

24.10 ■ Inne systemy

Istnieją, rzecz jasna, inne systemy operacyjne, z których większość odznacza się interesującymi właściwościami. System MCP dla rodziny komputerów Burroughs [275] był pierwszym, który napisano w języku programowania systemowego (zob. McKeag i Wilson [275]). Realizował on także segmentację i obsługę wielu procesorów. System operacyjny SCOPE dla maszyny CDC 6600 (zob. jak wyżej) był również systemem wieloprocesorowym. Osiągnięto w nim zaskakująco dobrą koordynację i synchronizację wielu procesów. System Tenex, opisany przez Bobrowa i in. [45], był wczesnym systemem ze stronicowaniem na żądanie dla komputerów PDP-10 i wywarł duży wpływ na następne systemy z podziałem czasu, takie jak TOPS-20 dla maszyn DEC-20. System operacyjny VMS dla komputerów VAX oparte na systemie operacyjnym RSX, zrealizowanym dla komputerów PDP-11. CP/M był najpopularniejszym systemem operacyjnym dla komputerów 8-bitowych – niektóre z nich pracują do dzisiaj. Najszerzej upowszechnionym systemem operacyjnym dla komputerów 16-bitowych jest MS-DOS. Popularyzują się graficzne interfejsy użytkownika (ang. *Graphical User Interfaces* – GUIs), ułatwiające posługiwanie się komputerami. Przewodzą wśród nich systemy Macintosh OS oraz Microsoft Windows.

* Mowa o czwartym wydaniu amerykańskim z 1994 r. – Przyp. tłum.



BIBLIOGRAFIA

1. Abbot C.: Intervention Schedules for Real-Time Programming. *IEEE Transactions on Software Engineering*, 1984, SE-10, 3, s. 268-274.
2. Accetta M., Baron R., Bolosky W., Golub D. B., Rashid R., Tevanian A., Jr., Young M.: Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Summer 1986 USENIX Conference*, June 1986, s. 93-112.
3. Agrawal D. P., Janakiram V. K., Pathak G. C.: Evaluating the Performance of Multicomputer Configurations. *Communications of the ACM*, 1986, 29, 5, s. 23-37.
4. Agrawal D. P., El Abbadi A.: An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 1991, 9, 1, s. 1-20.
5. Ahituv N., Lapid Y., Neumann S.: Processing Encrypted Data. *Communications of the ACM*, 1987, 30, 9, s. 777-780.
6. Aki S. G.: Digital Signatures: A Tutorial Survey. *Computer*, 1983, 16, 2, s. 15-24.
7. Akyurek S., Salem K.: Adaptive Block Rearrangement. *Proceedings of the Ninth International Conference on Data Engineering*, April 1993, s. 182-189.
8. Alt H.: Removable Media in Solaris. *Proceedings of the Winter 1993 USENIX Conference*, January 1993, s. 281-287.
9. Ammon G. J., Calabria J. A., Thomas D. T.: A High-Speed, Large-Capacity, "Jukebox" Optical Disk System. *Computer*, 1985, 18, 7, s. 36-48.
10. Anderson I. E., Lazowska E. D., Levy H. M.: The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 1989, 38, 12, s. 1631-1644.
11. Anderson I. E., Bershad B. N., Lazowska E. D., Levy H. M.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, s. 95-109.
12. Anyanwu J. A., Marshall I. F.: A Crash Resistant UNIX File System. *Software - Practice and Experience*, 1986, 16, 2, s. 107-118.
13. Apple Computer Inc.: *Apple Technical Introduction to the Macintosh Family*. Reading, MA, Addison-Wesley 1987.
14. Apple Computer Inc.: *Inside Macintosh, Volume VI*. Reading, MA, Addison-Wesley 1991.

15. Artsy Y. (ed.): Special Issue on Process Migration. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*. Winter 1989.
16. Artsy Y.: Designing a Process Migration Facility: The Charlotte Experience. *Computer*, 1989, 22, 9, s. 47-56.
17. Asthana P., Finkelstein B.: Superdense Optical Storage. *IEEE Spectrum*, 1995, 32, 8, s. 25-31.
18. AT&T, Earhart S. V. (ed.): *UNIX Programmer's Manual*. New York, NY, Holt, Rinehart, and Winston 1986.
19. Babaoglu O., Joy W.: Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits. *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, December 1969, s. 78-86.
20. Bach M. J.: *The Design of the UNIX Operating System*. Englewood Cliffs, NJ, Prentice-Hall 1987.
21. Baer J. L.: *Computer System Architecture*. Rockville, MD, Computer Science Press 1980.
22. Balkovich E., Lerman S. R., Parmelee R. P.: Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 1985, 28, 11, s. 1214-1224.
23. Barak A., Kornatzky Y.: Design Principles of Operating Systems for Large Scale Multicomputers. *Experience with Distributed Systems, Lecture Notes in Computer Science*. Springer-Verlag 1987, 309, s. 104-123.
24. Bayer R., Graham R. M., Seegmuller G. (eds.): *Operating Systems - An Advanced Course*. Berlin, Springer-Verlag 1978.
25. Bays C.: A Comparison of Next-Fit, First-Fit, and Best-Fit. *Communications of the ACM*, 1977, 20, 3, s. 191-192.
26. Beck M., Bohne H., Dziadzka M., Kunitz U., Magnus R., Verworner D.: *Linux Kernel Internals*. Addison Wesley Longman 1996.
27. Belady L. A.: A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 1966, 5, 2, s. 78-101.
28. Belady L. A., Nelson R. A., Sheddler G. S.: An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine. *Communications of the ACM*, 1969, 12, 6, s. 349-353.
29. Ben-Ari M.: *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ, Prentice-Hall 1990.
30. Benjamin C. D.: The Role of Optical Storage Technology for NASA's Image Storage and Retrieval Systems. *Proceedings Storage and Retrieval Systems and Applications*, February 1990, s. 10-17.
31. Bernstein P. A., Goodman N.: Time-stamp-based Algorithms for Concurrency Control in Distributed Database Systems. *Proceedings of the International Conference on Very Large Data Bases*, 1980, s. 285-300.
32. Bernstein A. J., Siegel P.: A Computer Architecture for Level Structured Operating Systems. *IEEE Transactions on Computers*, 1975, C-24, 8, s. 785-793.
33. Bernstein A., Hadzilacos V., Goodman N.: *Concurrency Control and Recovery in Database Systems*. Reading, MA, Addison-Wesley 1987.
34. Bershad B. N., Pinkerton C. B.: Watchdogs: Extending the UNIX File System. *Proceedings of the Winter 1988 USENIX Conference*, February 1988.
35. Bershad B. N., Anderson T. E., Lazowska F. D., Levy H. M.: Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 1990, 8, 1, s. 37-55.

* W 1995 r. Wydawnictwa Naukowo-Techniczne w Warszawie wydaly te ksiazke po polsku pt. *Budowa systemu operacyjnego UNIX®* - Przyp. tłum.

** Istnieje polskie tłumaczenie pt. *Podstawy programowania współbieżnego i rozproszonego*. Warszawa, WNT 1996. - Przyp. red.

36. Bhuyan L. N., Yang Q., Agrawal D. P.: Performance of Multiprocessor Interconnection Networks. *Computer*, 1989, **22**, 2, s. 25-37.
37. Bic L., Shaw A. C.: *The Logical Design of Operating Systems*. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1988.
38. Birman K., Joseph T.: Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 1987, **5**, 1.
39. Birrell A. D.: An Introduction to Programming with Threads. *Technical Report 35*, Palo Alto, CA, DEC-SRC, January 1989.
40. Birrell A. D., Nelson B. J.: Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 1984, **2**, 1, s. 39-59.
41. Black D. L.: Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, May 1990, s. 35-43.
42. Black D. L., Golub D. B., Rashid R. F., Tevanian A. Jr., Young M.: The Mach Exception Handling Facility. *Technical Report*, Carnegie-Mellon University, April 1988.
43. Black D. L., Golub D. B., Julian D. P., Rashid R. F., Draves R. P., Dean R. W., Forin A., Barrera J., Tokuda H., Malan G., Bohm D.: Microkernel Operating System Architecture and Mach. *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, s. 11-30.
44. Blair G. S., Malone J. R., Mariani J. A.: A Critique of UNIX. *Software - Practice and Experience*, 1985, **15**, 6, s. 1125-1139.
45. Bobrow D. G., Burchfiel J. D., Murphy D. L., Tomlinson R. S.: TENEX, a Paged Time Sharing System for the PDP-10. *Communications of the ACM*, 1972, **15**, 3.
46. Bourstyn R. R., Frank H.: Large-Scale Network Topological Optimization. *IEEE Transactions on Software Communications*, 1977, **COM-25**, 1, s. 29-47.
47. Bourne S. R.: The UNIX Shell. *Bell System Technical Journal*, 1978, **57**, 6, s. 1971-1990.
48. Bourne S. R.: *The UNIX System*. Reading, MA, Addison-Wesley 1983.
49. Boykin J., Kirschen D., Langerman A., LoVerso S.: *Programming Under Mach*. Reading, MA, Addison-Wesley 1993.
50. Boykin J., Langerman A. B.: Mach/4.3BSD: A Conservative Approach to Parallelization. *Computing Systems*, Winter 1990, **3**, 1, s. 69-99.
51. Brain M.: *Win32 System Services*. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1996.
52. Brent R.: Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation. *ACM Transactions on Programming Languages and Systems*, July 1989.
53. Brereton O. P.: Management of Replicated Files in a UNIX Environment. *Software - Practice and Experience*, 1986, **16**, s. 771-780.
54. Brinch Hansen P.: The Nucleus of the Multiprogramming System. *Communications of the ACM*, 1970, **13**, 4, s. 230-241 i 250.
55. Brinch Hansen P.: Structured Multiprogramming. *Communications of the ACM*, 1972, **15**, 7, s. 574-578.
56. Brinch Hansen P.: *Operating Systems Principles*. Englewood Cliffs, NJ, Prentice-Hall 1973.
57. Brownbridge D. R., Marshall L. F., Randell B.: The Newcastle Connection or UNIXes of the World Unite! *Software - Practice and Experience*, 1982, **12**, 12, s. 1147-1162.
58. Brumfield J. A.: A Guide to Operating Systems Literature. *Operating Systems Review*, 1986, **20**, 2, s. 38-42.
59. Brunt R. F., Tufts D. E.: A User-Oriented Approach to Control Languages. *Software - Practice and Experience*, 1976, **6**, 1, s. 93-108.

* Istnieje polskie tłumaczenie pt. *Podstawy systemów operacyjnych*. Warszawa, WNT 1979. - Przyp. red.

60. Burns J. E.: Mutual Exclusion with Linear Waiting Using Binary Shared Variables. *SIGACT News*, 1978, **10**, 2, s. 42-47.
61. Cannon M. R.: Data Storage on Tape. Opublikowane w [282], rozdz. 4, s. 170-241.
62. Carvalho O. S., Roucairol G.: On Mutual Exclusion in Computer Networks. *Communications of the ACM*, 1983, **26**, 2, s. 146-147.
63. Carr W. R., Hennessy J. L.: WSClock - A Simple and Effective Algorithm for Virtual Memory Management. *Proceedings of the Eighth Symposium on Operating System Principles*, December 1981, s. 87-95.
64. Caswell D., Black D.: Implementing a Mach Debugger for Multithreaded Applications. *Technical Report*, Carnegie-Mellon University, PA, November 1989.
65. Caswell D., Black D.: Implementing a Mach Debugger for Multithreaded Applications. *Proceedings of the Winter 1990 USENIX Conference*, January 1990, s. 25-40.
66. Cerf V. G., Cain E.: The DoD Internet Architecture Model. *Computer Networks*, 1983, **7**, 5, s. 307-318.
67. Chang E.: N-Philosophers: An Exercise in Distributed Control. *Computer Networks*, 1980, **4**, 2, s. 71-76.
68. Chang A., Mergen M. F.: 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 1988, **6**, 1, s. 28-50.
69. Chen P. M., Lee E. K., Gibson G. A., Katz R. H., Patterson D. A.: RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 1994, **26**, 2, s. 145-185.
70. Cheriton D. R., Zwaenepoel W. Z.: The Distributed V Kernel and Its Performance for Diskless Workstations. *Proceedings of the Ninth Symposium on Operating Systems Principles*, October 1983, s. 129-140.
71. Cheriton D. R., Malcolm M. A., Melen L. S., Sager G. R.: Thoth, a Portable Real-Time Operating System. *Communications of the ACM*, 1979, **22**, 2, 105-115.
72. Chi C. S.: Advances in Computer Mass Storage Technology. *Computer*, 1982, **15**, 5, s. 60-74.
73. Chow T. C. K., Abraham J. A.: Load Balancing in Distributed Systems. *IEEE Transactions on Software Engineering*, 1982, **SE-8**, 4, s. 401-412.
74. Chu W. W., Opderbeck H.: Program Behavior and the Page-Fault-Frequency Replacement Algorithm. *Computer*, 1976, **9**, 11, s. 29-38.
75. Coffman E. G., Denning P. J.: *Operating Systems Theory*. Englewood Cliffs, NJ, Prentice-Hall 1973.
76. Coffman E. G., Kleinrock L.: Feedback Queuing Models for Time-Shared Systems. *Journal of the ACM*, 1968, **15**, 4, s. 549-576.
77. Coffman E. G., Elphick M. J., Shoshani A.: System Deadlocks. *Computing Surveys*, 1971, **3**, 2, s. 67-78.
78. Cohen E. S., Jefferson D.: Protection in the Hydra Operating System. *Proceedings of the Fifth Symposium on Operating Systems Principles*, November 1975, s. 141-160.
79. Comer D.: *Operating System Design: the Xinu Approach*. Englewood Cliffs, NJ, Prentice-Hall 1984.
80. Comer D.: *Operating System Design - Volume II: Internetworking with Xinu*. Englewood Cliffs, NJ, Prentice-Hall 1987.
81. Comer D.: *Internetworking with TCP/IP, Volume I*. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1991*

* Trzecie wydanie tej książki ukazało się w 1997 r. po polsku pt. *Sieci komputerowe TCP/IP. Tom I: Zasady, protokoły i architektura* nakładem Wydawnictw Naukowo-Teknicznych w Warszawie – Przyp. tłum.

82. Comer D., Stevens D. L.: *Internetworking with TCP/IP. Volume II*. Englewood Cliffs, NJ, Prentice-Hall 1991*.
83. Comer D., Stevens D. L.: *Internetworking with TCP/IP. Volume III*. Englewood Cliffs, NJ, Prentice-Hall 1993**.
84. Cooper E. C., Draves R. P.: C Threads. *Technical Report*, Carnegie-Mellon University, PA, July 1987.
85. Corbató F. J., Vyssotsky V. A.: Introduction and Overview of the MULTICS System. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1965, s. 185-196.
86. Corbató F. J., Mervin-Daggett M., Daley R. C.: An Experimental Time-Sharing System. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1962, s. 335-344.
87. Courtois P. J., Heymans F., Parnas D. L.: Concurrent Control with 'Readers' and 'Writers'. *Communications of the ACM*, 1971, **14**, **10**, s. 667-668.
88. Creasy R. J.: The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 1981, **25**, **5**, s. 483-490.
89. Computer Systems Research Group – University of California at Berkeley: *BSD UNIX Reference Manual*. Sześć tomów. USENIX Association, 1986.
90. Custer H.: *Inside Windows NT*. Redmond, WA, Microsoft Press 1993.
91. Custer H.: *Inside the Windows NT File System*. Redmond, WA, Microsoft Press 1994.
92. Davcev D., Burkhard W. A.: Consistency and Recovery Control for Replicated Files. *Proceedings of the Tenth Symposium on Operating Systems Principles*, 1985, **19**, **5**, s. 87-96.
93. Davies D. W.: Applying the RSA Digital Signature to Electronic Mail. *Computer*, 1983, **16**, **2**, s. 55-62.
94. Day J. D., Zimmerman H.: The OSI Reference Model. *Proceedings of the IEEE*, 1983, **71**, s. 1334-1340.
95. deBruijn N. G.: Additional Comments on a Problem in Concurrent Programming and Control. *Communications of the ACM*, 1967, **10**, **3**, s. 137-138.
96. Deitel H. M.: *An Introduction to Operating Systems*. Wyd. 2, Reading, MA, Addison-Wesley 1990.
97. Deitel H. M., Kogan M. S.: *The Design of OS/2*. Reading, MA, Addison-Wesley 1992.
98. Denning P. J.: The Working Set Model for Program Behavior. *Communications of the ACM*, 1968, **11**, **5**, s. 323-333.
99. Denning P. J.: Third Generation Computer System. *Computing Surveys*, 1971, **3**, **34**, s. 175-216.
100. Denning P. J.: Working Sets Past and Present. *IEEE Transactions on Software Engineering*, 1980, **SE-6**, **1**, s. 64-84.
101. Denning D. E.: *Cryptography and Data Security*. Reading, MA, Addison-Wesley 1982***.
102. Denning D. E.: Protecting Public Keys and Signature Keys. *IEEE Computer*, 1983, **16**, **2**, s. 27-35.

* Drugie wydanie tej książki ukazało się w 1997 r. po polsku pt. *Sieci komputerowe TCP/IP. Tom 2: Projektowanie i realizacja protokołów* nakładem Wydawnictw Naukowo-Technicznych. – Przyp. tłum.

** Książka ta ukazała się w 1997 r. po polsku pt. *Sieci komputerowe TCP/IP. Tom 3. Programowanie w trybie klient-serwer. Wersja BSD* nakładem Wydawnictw Naukowo-Technicznych. – Przyp. tłum.

*** Polskie tłumaczenie tej książki ukazało się w 1993 r. pt. *Kryptografia i ochrona danych* nakładem Wydawnictw Naukowo-Technicznych. – Przyp. tłum.

103. Denning D. E.: Digital Signatures with RSA and Other Public-Key Cryptosystems. *Communications of the ACM*, 1984, 27, 4, s. 388-392.
104. Dennis J. B.: Segmentation and the Design of Multiprogrammed Computer Systems. *Journal of the ACM*, 1965, 12, 4, s. 589-602.
105. Dennis J. B., Van Horn E. C.: Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 1966, 9, 3, s. 143-155.
106. Department of Defence Trusted Computer System Evaluation Criteria. Department of Defence, DoD 5200 28-STD, December 1985.
107. Diffie W., Hellman M. E.: New Directions in Cryptography. *IEEE Transactions on Information Theory*, 1976, 22, 6, s. 644-654.
108. Diffie W., Hellman M. E.: Privacy and Authentication. *Proceedings of the IEEE*, 1979, 67, 3, s. 397-427.
109. Digital Equipment Corporation: *VAX Architecture Handbook*. Maynard, MA, Digital Equipment Corporation 1981.
110. Dijkstra E. W.: Cooperating Sequential Processes. *Technical Report EWD-123*, Technological University, Eindhoven, the Netherlands, 1965; przedrukowane w [148], s. 43-112.
111. Dijkstra E. W.: Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 1965, 8, 9, s. 569.
112. Dijkstra E. W.: The Structure of the IEEE Multiprogramming System. *Communications of the ACM*, 1968, 11, 5, s. 341-346.
113. Dijkstra E. W.: Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1971, 1, 2, s. 115-138; przedrukowane w [174], s. 72-93.
114. Docppner T. W.: Threads: A System for the Support of Concurrent Programming. *Technical Report CS-87-11*, Department of Computer Science, Brown University, June 1987.
115. Donnelley J. E.: Components of a Network-Operating System. *Computer Networks*, 1979, 3, 6, s. 389-399.
116. Douglass F., Ousterhout J.: Process Migration in the Sprite Operating System. *Proceedings of the Seventh IEEE International Conference on Distributed Computing Systems*, 1987, s. 18-25.
117. Douglass F., Ousterhout J.: Log-Structured File Systems. *Proceedings of the 34th COMPCON Conference*, February 1989, s. 124-129.
118. Douglass F., Ousterhout J.: Process Migration in Srite: A Status Report. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.
119. Douglass F., Kaashoek M. F., Tanenbaum A. S.: A Comparison of Two Distributed Systems: Amoeba and Sprite. *Computing Systems*, Fall 1991, 4.
120. Draves R. P., Jones M. B., Thompson M. R.: MIG - The MACH Interface Generator. *Technical Report*, Carnegie-Mellon University, PA, November 1989.
121. Draves R. P., Bershad B. N., Rashid R. F., Dean R. W.: Using Continuations to Implement Thread Management and Communication in Operating Systems. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, s. 122-136.
122. Duncan R.: A Survey of Parallel Computer Architectures. *IEEE Computer*, 1990, 23, 2, s. 5-16.
123. Eager D., Lazowska E., Zahorjan J.: Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 1986, SE-12, 5, s. 662-675.
124. Eisenberg M. A., McGuire M. R.: Further Comments on Dijkstra's Concurrent Programming Control Problem. *Communications of the ACM*, 1972, 15, 11, s. 999.
125. Ekanadham K., Bernstein A. J.: Conditional Capabilities. *IEEE Transactions on Software Engineering*, 1979, SE-5, 5, s. 458-464.

126. Eskicioglu M.: Design Issues of Process Migration Facilities in Distributed Systems. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Summer 1990.
127. Eswaran K. P., Gray J. N., Lorie J. N., Traiger I. L.: The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*. 1976, **19**, 11, s. 624-633.
128. Eykholt J. R., Kleiman S. R., Barton S., Faulkner S., Shivalingiah M., Smith M., Stein D., Voll J., Weeks M., Williams D.: Beyond Multiprocessing: Multithreading of the SunOS Kernel. *Proceedings of the Summer 1992 USENIX Conference*, June 1992, s. 11-18.
129. Farrow R.: Security Issues and Strategies for Users. *UNIX World*, April 1986, s. 65-71.
130. Farrow R.: Security for Superusers, or How to Break the UNIX System. *UNIX World*, May 1986, s. 65-70.
131. Feitelson D., Rudolph L.: Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control. *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.
132. Ferguson D., Yemini Y., Nikolaou C.: Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. *Proceedings of the Eighth IEEE International Conference on Distributed Computing Systems*, 1988, s. 491-499.
133. Feng T.: A Survey of Interconnection Networks. *Computer*. 1981, **14**, 12, s. 12-27.
134. Filipski A., Hanko J.: Making UNIX Secure. *Byte*, April 1986, s. 113-128.
135. Finkel R. A.: *Operating Systems Vade Mecum*. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1988.
136. Finlayson R. S., Cheriton D. R.: Log Files: An Extended File Service Exploiting Write-Once Storage. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, s. 139-148.
137. Folk M. J., Zoellick B.: *File Structures*. Reading, MA, Addison-Wesley 1987.
138. Forsdick H. C., Schantz R. E., Thomas R. H.: Operating Systems for Computer Networks. *Computer*. 1978, **11**, 1, s. 48-57.
139. Fortier P. J.: *Handbook of LAN Technology*. New York, NY, McGraw-Hill 1989.
140. Frank G. R.: Job Control in the MUS Operating System. *Computer Journal*. 1976, **19**, 2, s. 139-143.
141. Freedman D. H.: Searching for Denser Disks. *InfoSystems*, September 1983, s. 56.
142. Freese R. P.: Optical Disks Become Erasable. *IEEE Spectrum*. 1988, **25**, 2, s. 41-45.
143. Fujitani L.: Laser Optical Disk: The Coming Revolution in On-Line Storage. *Communications of the ACM*, 1984, **27**, 6, s. 546-554.
144. Gait J.: The Optical File Cabinet: A Random-Access File System for Write-On Optical Disks. *Computer*. 1988, **21**, 6.
145. Garcia-Molina H.: Elections in Distributed Computing Systems. *IEEE Transactions on Computers*, 1982, C-31, 1.
146. Garfinkel S., Spafford G.: *Practical UNIX Security*. Sebastopol, CA, O'Reilly & Associates, Inc. 1991.
147. Geist R., Daniel S.: A Continuum of Disk Scheduling Algorithms. *ACM Transactions on Computer Systems*, 1987, **5**, 1, s. 77-92.
148. Genuys F. (ed.): *Programming Languages*. London, England, Academic Press 1968.
149. Gerla M., Kleinrock L.: Topological Design of Distributed Computer Networks. *IEEE Transactions on Communications*, 1977, COM-25, 1, s. 48-60.
150. Gifford D. K.: Cryptographic Scaling for Information Secrecy and Authentication. *Communications of the ACM*. 1982, **25**, 4, s. 274-286.

151. Golden D., Pechura M.: The Structure of Microcomputer File Systems. *Communications of the ACM*, 1986, **29**, 3, s. 222-230.
152. Goldman P.: Mac VM Revealed. *Byte*, September 1989.
153. Grampp F. T., Morris R. H.: UNIX Operating System Security. *AT&T Bell Laboratories Technical Journal*, 1984, **63**, s. 1649-1672.
154. Gray J. N.: Notes on Data Base Operating Systems. Opublikowane w [24], s. 393-481.
155. Gray J. N.: The Transaction Concept: Virtues and Limitations. *Proceedings of the International Conference on Very Large Data Bases*, 1981, s. 144-154.
156. Gray J. N., McJones P. R., Blasgen M.: The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 1981, **13**, 2, s. 223-242.
157. Grosshans D.: *File Systems Design and Implementation*. Englewood Cliffs, NJ, Prentice-Hall 1986.
158. Gupta R. K., Franklin M. A.: Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison. *IEEE Transactions on Computers*, 1978, **C-27**, 8, s. 706-712.
159. Habermann A. N.: Prevention of System Deadlocks. *Communications of the ACM*, 1969, **12**, 7, s. 373-377.
160. Hagmann R.: Comments on Workstation Operating Systems and Virtual Memory. *Proceedings of the Second Workshop on Workstation Operating Systems*, September 1989.
161. Haldar S., Subramanian D.: Fairness in Processor Scheduling in Time Sharing Systems. *Operating Systems Review*, January 1991.
162. Hall D. E., Scherrer D. K., Sventek J. S.: A Virtual Operating System. *Communications of the ACM*, 1980, **23**, 9, s. 495-502.
163. Halsall F.: Data Communications, *Computer Networks, and Open Systems*. Reading, MA, Addison-Wesley 1992.
164. Harker J. M., Brede D. W., Pattison R. E., Santana G. R., Taft L. G.: A Quarter Century of Disk File Innovation. *IBM Journal of Research and Development*, 1981, **25**, 5, s. 677-689.
165. Harrison M. A., Ruzzo W. L., Ullman J. D.: Protection in Operating Systems. *Communications of the ACM*, 1976, **19**, 8, s. 461-471.
166. Havender J. W.: Avoiding Deadlock in Multitasking Systems. *IBM Systems Journal*, 1968, **7**, 2, s. 74-84.
167. Hecht M. S., Johri A., Aditham R., Wei T. J.: Experience Adding C2 Security Features to UNIX. *Proceedings of the Summer 1988 USENIX Conference*, June 1988, s. 133-146.
168. Hendricks E. C., Hartmann T. C.: Evolution of a Virtual Machine Subsystem. *IBM Systems Journal*, 1979, **18**, 1, s. 111-142.
169. Hennessy J. L., Patterson D. A.: *Computer Architecture: A Quantitative Approach*. Palo Alto, CA, Morgan Kaufmann Publishers 1990.
170. Henry G.: The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, October 1984.
171. Hoagland A. S.: Information Storage Technology - A Look at the Future. *Computer*, 1985, **18**, 7, s. 60-68.
172. Hoare C. A. R.: Towards a Theory of Parallel Programming. Opublikowane w [174], s. 61-71.
173. Hoare C. A. R.: Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 1974, **17**, 10, s. 549-557. Erratum zamieszczono w *Communications of the ACM*, 1975, **18**, 2, s. 95.
174. Hoare C. A. R., Perrott R. H. (eds.): *Operating Systems Techniques*. London, Academic Press 1972.

175. Holley L. H., Parmelee R. P., Salisbury C. A., Saul D. N.: VM/370 Asymmetric Multiprocessing. *IBM Systems Journal*, 1979, **18**, 1, s. 47-70.
176. Holt R. C.: Comments on Prevention of System Deadlocks. *Communications of the ACM*, 1971, **14**, 1, s. 36-38.
177. Holt R. C.: Some Deadlock Properties of Computer Systems. *Computing Surveys*, 1972, **4**, 3, s. 179-196.
178. Holt R. C.: *Concurrent Euclid, the UNIX System, and Tunis*. Reading, MA: Addison-Wesley 1983.
179. Hong J., Tan X., Towsley D.: A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System. *IEEE Transactions on Computers*, 1989, **38**, 12, s. 1736-1744.
180. Howard J. H.: Mixed Solutions for the Deadlock Problem. *Communications of the ACM*, 1973, **16**, 7, s. 427-430.
181. Howard J. H., Kazar M. L., Menees S. G., Nichols D. A., Satyanarayanan M., Sidebotham R. N.: Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 1988, **6**, 1, s. 55-81.
182. Howarth D. J., Payne R. B., Sumner F. H.: The Manchester University Atlas Operating System, Part II: User's Description. *Computer Journal*, 1961, **4**, 3, s. 226-229.
183. Hsiao D. K., Kerr D. S., Madnick S. E.: *Computer Security*. New York, NY: Academic Press 1979.
184. Hyman D.: *The Columbus Chicken Statute, and More Bonehead Legislation*. Lexington, MA: S. Greene Press 1985.
185. Iacobucci F.: *OS/2 Programmer's Guide*. Berkeley, CA: Osborne McGraw-Hill 1988.
186. IBM Corporation: *Technical Reference*. IBM 1983.
187. Institute of Electrical and Electronic Engineers: *IEEE Computer*, 1994, **27**, 3.
188. Iliffe J. K., Jodeit J. G.: A Dynamic Storage Allocation System. *Computer Journal*, 1962, **5**, 3, s. 200-209.
189. Intel Corporation: *iAPX 86/88, 186/188 User's Manual Programmer's Reference*. Santa Clara, CA, Intel Corp. 1985.
190. Intel Corporation: *iAPX 286 Programmer's Reference Manual*. Santa Clara, CA, Intel Corp. 1985.
191. Intel Corporation: *iAPX 386 Programmer's Reference Manual*. Santa Clara, CA, Intel Corp. 1986.
192. Intel Corporation: *i486 Microprocessor*. Santa Clara, CA, Intel Corp. 1989.
193. Intel Corporation: *i486 Microprocessor Programmer's Reference Manual*. Santa Clara, CA, Intel Corp. 1990.
194. Intel Corporation: *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*. Mt. Prospect, IL, Intel Corp. 1993.
195. Isloor S. S., Marsland T. A.: The Deadlock Problem: An Overview. *Computer*, 1980, **13**, 9, s. 58-78.
196. ISO Open Systems Interconnection - Basic Reference Model, ISO/TC 97/SC 16 N 719 International Organization for Standardization, August 1981.
197. Ivens K., Hallberg B.: *Inside Windows NT Workstation 4*. Indianapolis, IN: New Riders Publishing 1996.
198. Jensen E. D., Locke C. D., Tokuda H.: A Time-Driven Scheduling Model for Real-Time Operating Systems. *Proceedings of the IEEE Real-Time Systems Symposium*, December 1985, s. 112-122.
199. Jones A. K.: Protection Mechanisms and the Enforcement of Security Policies. Opublikowane w [24], s. 228-250.

200. Jones A. K., Liskov B. H.: A Language Extension for Expressing Constraints on Data Access. *Communications of the ACM*, 1978, **21**, 5, s. 358-367.
201. Jones A. K., Schwarz P.: Experience Using Multiprocessor Systems – A Status Report. *Computing Surveys*, 1980, **12**, 2, s. 121-165.
202. Jul E., Levy H., Hutchinson N., Black A.: Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 1988, **6**, 1, s. 109-133.
203. Katz R. H., Gibson G. A., Patterson D. A.: Disk System Architectures for High Performance Computing. *Proceedings of the IEEE*, 1989, **77**, 12.
204. Kay J., Lauder P.: A Fair Share Scheduler. *Communications of the ACM*, 1988, **31**, 1, s. 44-55.
205. Kenan L. J., Goldenberg R. E., Bate S. F.: *VAX/VMS Internals and Data Structures*. Bedford, MA, Digital Press 1988.
206. Kenville R. F.: Optical Disk Data Storage. *Computer*, 1982, **15**, 7, s. 21-26.
207. Kernighan B. W., Pike R.: *The UNIX Programming Environment*. Englewood Cliffs, NJ, Prentice-Hall 1984.
208. Kernighan B. W., Ritchie D. M.: *The C Programming Language*. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1988.
209. Kessels J. L. W.: An Alternative to Event Queues for Synchronization in Monitors. *Communications of the ACM*, 1977, **20**, 7, s. 500-503.
210. Khanna S., Schree M., Zolnowsky J.: Runtime Scheduling in SunOS 5.0. *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, CA, January 1992, s. 375-390.
211. Kieburz R. B., Silberschatz A.: Capability Managers. *IEEE Transactions on Software Engineering*, 1978, **SE-4**, 6, s. 467-477.
212. Kieburz R. B., Silberschatz A.: Access Rights Expressions. *ACM Transactions on Programming Languages and Systems*, 1983, **5**, 1, s. 78-96.
213. Kilburn T., Howarth D. J., Payne R. B., Sumner F. H.: The Manchester University Atlas Operating System, Part I: Internal Organization. *Computer Journal*, 1961, **4**, 3, s. 222-225.
214. King R. P.: Disk Arm Movement in Anticipation of Future Requests. *ACM Transactions on Computer Systems*, 1990, **8**, 3, s. 214-229.
215. Kleinrock L.: *Queueing Systems, Volume II: Computer Applications*. New York, NY, Wiley-Interscience 1975.
216. Knapp E.: Deadlock Detection in Distributed Databases. *Computing Surveys*, 1987, **19**, 4, s. 303-328.
217. Knuth D. E.: Additional Comments on a Problem in Concurrent Programming Control. *Communications of the ACM*, 1966, **9**, 5, s. 321-322.
218. Knuth D. E.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Wyd. 2, Reading, MA, Addison-Wesley 1973**.
219. Koch P. D. L.: Disk File Allocation Based on the Buddy System. *ACM Transactions on Computer Systems*, 1987, **5**, 4, s. 352-370.
220. Kogan M. S., Rawson F. L.: The Design of Operating System/2. *IBM Systems Journal*, 1988, **27**, 2, s. 90-104.

* Książkę tę przetłumaczono na język polski i wydano pod tytułem: *Język ANSI C*. Wydawnictwa Naukowo-Techniczne, Warszawa 1994. Również pierwsze (już mniej aktualne) amerykańskie wydanie tej książki ukazało się w przekładzie polskim (*Język C*, WNI, Warszawa 1988). – Przyp. tłum.

** Polskie tłumaczenie ukazało się w 2001 r. nakładem Wydawnictw Naukowo-Technicznych. – Przyp. red.

221. Korn D.: KSH, A Shell Programming Language. *Proceedings of the Summer 1983 USENIX Conference*, July 1983, s. 191-202.
222. Kosaraju S.: Limitations of Dijkstra's Semaphores Primitives and Petri Nets. *Operating Systems Review*, 1973, 7, 4, s. 122-126.
223. Krakowiak S.: *Principles of Operating Systems*. Cambridge, MA, MIT Press 1988.
224. Kramer S. M.: Retaining SUID Programs in a Secure UNIX. *Proceedings of the 1988 Summer USENIX Conference*, June 1988, s. 107-118.
225. Lamport L.: A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 1974, 17, 8, s. 453-455.
226. Lamport L.: Synchronization of Independent Processes. *Acta Informatica*, 1976, 7, 1, s. 15-34.
227. Lamport L.: Concurrent Reading and Writing. *Communications of the ACM*, 1977, 20, 11, s. 806-811.
228. Lamport L.: Time, Clocks and Ordering of Events in a Distributed System. *Communications of the ACM*, 1978, 21, 7, s. 558-565.
229. Lamport L.: The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 1978, 2, 2, s. 95-114.
230. Lamport L.: Password Authentication with Insecure Communications. *Communications of the ACM*, 1981, 24, 11, s. 770-772.
231. Lamport L.: The Mutual Exclusion Problem. *Journal of the ACM*, 1986, 33, 2, s. 313-348.
232. Lamport L.: The Mutual Exclusion Problem Has Been Solved. *Communications of the ACM*, 1991, 34, 1, s. 110.
233. Lamport L., Shostak R., Pease M.: The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 1982, 4, 3, s. 382-401.
234. Lampson B.W.: A Scheduling Philosophy for Multiprocessing Systems. *Communications of the ACM*, 1968, 11, 5, s. 347-360.
235. Lampson B.W.: Dynamic Protection Structures. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1969, s. 27-38.
236. Lampson B.W.: Protection. *Proceedings of the Fifth Annual Princeton Conference on Information Science Systems*, 1971, s. 437-443; przedrukowane w: *Operating Systems Review*, 1974, 8, 1, s. 18-24.
237. Lampson B. W.: A Note on the Confinement Problem. *Communications of the ACM*, 1973, 16, 10, s. 613-615.
238. Lampson B. W., Sturgis H.: Crash Recovery in a Distributed Data Storage System. *Technical Report*, Palo Alto, CA, Computer Science Laboratory, Xerox, Palo Alto Research Center, 1976.
239. Landwehr C. E.: Formal Models of Computer Security. *Computing Surveys*, 1981, 13, 3, s. 247-278.
240. Larson P., Kajla A.: File Organization: Implementation of a Method Guaranteeing Retrieval in One Access. *Communications of the ACM*, 1984, 27, 7, s. 670-677.
241. Lazowska E. D., Zahorjan J., Graham G. S., Sevcik K. C.: *Quantitative System Performance*. Englewood Cliffs, NJ, Prentice-Hall 1984.
242. Leach P. J., Stump B. L., Hamilton J. A., Levine P. H.: UID's as Internal Names in a Distributed File System. *Proceedings of the First Symposium on Principles of Distributed Computing*, August 1982, s. 34-41.
243. Leffler S. J., Fabry R. S., Joy W. N.: A 4.2BSD Interprocess Communication Primer. *UNIX Programmer's Manual*, 2C, University of California at Berkeley, CA, 1978.
244. Leffler S. J., Joy W. N., Fabry R. S.: 4.2BSD Networking Implementation Notes. *UNIX Programmer's Manual*, 2C, University of California at Berkeley, CA, 1983.

245. Leffler S. J., McKusick M. K., Karels M. J., Quarterman J. S.: *The Design and Implementation of the 4.3BSD UNIX Operating System*. Reading, MA, Addison-Wesley 1989*.
246. Lehmann F.: Computer Break-Ins. *Communications of the ACM*, 1987, 30, 7, s. 584-585.
247. Le Lann G.: Distributed Systems – Towards a Formal Approach. *Proceedings of the IFIP Congress 77*, 1977, s. 155-160.
248. Lempel A.: Cryptology in Transition. *Computing Surveys*, 1979, 11, 4, s. 286-303.
249. Lett A. L., Konigsburg W. L.: TSS/360: A Time-Shared Operating System. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1968, s. 15-28.
250. Letwin G.: *Inside OS 2*. Redmond, WA, Microsoft Press 1988.
251. Leutenegger S., Veillon M.: The Performance of Multiprogrammed Multiprocessor Scheduling Policies. *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, May 1990.
252. Levin R., Cohen E. S., Corwin W. M., Pollack F. J., Wulf W. A.: Policy/Mechanism Separation in Hydra. *Proceedings of the Fifth ACM Symposium on Operating System Principles*, 1975, s. 132-140.
253. Levy H. M., Lipman P. H.: Virtual Memory Management in the VAX/VMS Operating System. *Computer*, 1982, 15, 3, s. 35-41.
254. Levy E., Silberschatz A.: Distributed File Systems: Concepts and Examples. *Computing Surveys*, 1990, 22, 4, s. 321-374.
255. Lichtenberger W. W., Pirtle M. W.: A Facility for Experimentation in Man-Machine Interaction. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1965, s. 589-598.
256. Lions J.: A Commentary on the UNIX Operating System. *Technical Report*, Department of Computer Science, The University of South Wales, November 1977.
257. Lipner S.: A Comment on the Confinement Problem. *Operating Systems Review*, 1975, 9, 5, s. 192-196.
258. Lipton R.: On Synchronization Primitive Systems. PhD Thesis, Carnegie-Mellon University, 1974.
259. Liskov B. H.: The Design of the Venus Operating System. *Communications of the ACM*, 1972, 15, 3, s. 144-149.
260. Litzkow M. J., Livny M., Mutka M. W.: Condor – A Hunter of Idle Workstations. *Proceedings of the Eighth IEEE International Conference on Distributed Computing Systems*, 1988, s. 104-111.
261. Liu C. L., Layland J. W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 1973, 20, 1, s. 46-61.
262. Lobel J.: *Fooling the System Breakers. Computer Security and Access Control*. New York, NY, McGraw-Hill 1986.
263. Loepere K.: Mach 3 Kernel Principles. *Technical Report*, Open Software Foundation, MA, January 1992.
264. Loucks L. K., Saucr C. H.: Advanced Interactive Executive (AIX) Operating System Overview. *IBM Systems Journal*, 1987, 26, 4, s. 326-345.
265. Lynch W. C.: An Operating System Design for the Computer Utility Environment. *Opus Mikowano w [174]*, s. 341-350.
266. MacKinnon R. A.: The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines. *IBM Systems Journal*, 1979, 18, 1, s. 18-46.

* Istnieje nowsza książka poświęcona tej tematyce, odnosząca się do systemu 4.4BSD – McKusick M. K., Bostic K., Karels M. J., Quarterman J. S.: *The Design and Implementation of the 4.4BSD Operating System*. Reading, MA, Addison-Wesley 1996. – Przyp. tłum.

267. Mackawa M.: A Square Root Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, 1985, 3, 2, s. 145-159.
268. Mackawa M., Oldehoeft A. E., Oldehoeft R. R.: *Operating Systems: Advanced Concepts*. Menlo Park, CA, Benjamin/Cummings 1987.
269. Maher C., Goldick J. S., Kerby C., Zumach B.: The Integration of Distributed File Systems and Mass Storage Systems. *Proceedings of the Thirteenth IEEE Symposium on Mass Storage Systems*, June 1994, s. 27-31.
270. Maples C.: Analyzing Software Performance in a Multiprocessor Environment. *IEEE Software*, 1985, 2, 4, s. 50-64.
271. March B. D., Scott M. L., LeBlanc T. J., Markatos E. P.: First-Class User-Level Threads. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, s. 110-121.
272. Massalin H., Pu C.: Threads and Input/Output in the Synthesis Kernel. *Proceedings of the 12th Symposium on Operating Systems Principles*, December 1989, s. 191-200.
273. Mattson R. L., Gecsi J., Slutz D. R., Traiger I. L.: Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 1970, 9, 2, s. 78-117.
274. McGraw J. R., Andrews G. R.: Access Control in Parallel Programs. *IEEE Transactions on Software Engineering*, 1979, SE-5, 1, s. 1-9.
275. McKeag R. M., Wilson R.: *Studies in Operating Systems*. London, Academic Press 1976.
276. McKeon B.: An Algorithm for Disk Caching with Limited Memory. *Byte*, 1985, 10, 9, s. 129-138.
277. McKusick M. K., Karels K. J.: Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel. *Proceedings of the Summer 1988 USENIX Conference*, June 1988, s. 295-304.
278. McKusick M. K., Joy W. N., Leffler S. J., Fabry R. S.: A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 1984, 2, 3, s. 181-197.
279. McNamee D., Armstrong K.: Extending the Mach External Pager Interface to Accommodate User-Level Page-Replacement Policies. *Proceedings of the USENIX MACH Workshop*, Burlington, VT, October 1990.
280. McVoy I. W., Kleiman S. R.: Extent-like Performance from a UNIX File System. *Proceedings of the Winter 1991 USENIX Conference*, January 1991, s. 33-44.
281. Mealy G. H., Witt B. L., Clark W. A.: The Functional Structure of OS/360. *IBM Systems Journal*, 1966, 5, 1.
282. Mee C. D., Daniel E. D. (eds.): *Magnetic Recording*. New York, NY, McGraw-Hill 1988.
283. Menasce D., Muntz R. R.: Locking and Deadlock Detection in Distributed Data Bases. *IEEE Transactions on Software Engineering*, 1979, SE-5, 3, s. 195-202.
284. Metzner J. R.: Structuring Operating Systems Literature for the Graduate Course. *Operating Systems Review*, 1982, 16, 4, s. 10-25.
285. Meyer J., Downing T.: *Java Virtual Machine*. Sebastopol, CA, O'Reilly and Associates, Inc. 1997.
286. Meyer R. A., Seawright L. H.: A Virtual Machine Time-Sharing System. *IBM Systems Journal*, 1970, 9, 3, s. 199-218.
287. Microsoft Corporation: *Microsoft Developer Network Development Library*. Redmond, WA, Microsoft Press 1997.
288. Microsoft Corporation: *Microsoft MS-DOS User's Reference and Microsoft MS-DOS Programmer's Reference*. Redmond, WA, Microsoft Press 1986.
289. Microsoft Corporation: *Microsoft Operating System/2 Programmer's Reference*. Trinity, Redmond, WA, Microsoft Press 1989.
290. Microsoft Corporation: *Microsoft MS-DOS User's Guide and Reference*. Redmond, WA, Microsoft Press 1991.

291. Microsoft Windows NT Resource Kit: Volume 1. Windows NT Resource Guide. Redmond, WA. Microsoft Press 1993.
292. Microsoft Windows NT Workstation Resource Kit. Redmond, WA, Microsoft Press 1996.
293. Milenkovic M.: Operating Systems: Concepts and Design. New York, NY, McGraw-Hill 1987.
294. Miller E. L., Katz R. H.: An Analysis of File Migration in a UNIX Supercomputing Environment. *Proceedings of the Winter 1993 USENIX Conference*. January 1993. s. 421-434.
295. Mohan C., Lindsay B.: Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on the Principles of Distributed Computing*, 1983.
296. Morris J. H.: Protection in Programming Languages. *Communications of the ACM*. 1973. 16. 1. s. 15-21
297. Morris J. H., Satyanarayanan M., Conner M. H., Howard J. H., Rosenthal D. S. H., Smith F. D., Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*. 1986. 29. 3. s. 184-201.
298. Morris R., Thompson K.: Password Security A Case History. *Communications of the ACM*. 1979. 22. 11. s. 594-597.
299. Morshedian D.: How to Fight Password Pirates. *Computer*. 1986. 19. 1
300. Motorola Inc.: MC68000 Family Reference. Wyd. 2. Englewood Cliff, NJ, Prentice-Hall 1989.
301. Motorola Inc.: MC68030 Enhanced 32-Bit Microprocessor User's Manual. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1989.
302. Motorola Inc.: PowerPC 601 RISC Microprocessor User's Manual. Phoenix, AZ, Motorola 1993
303. Mullender S. J. (ed.): Distributed Systems. Wyd. 2. New York, NY, ACM Press 1993
304. Mullender S. J., Tanenbaum A. S.: A Distributed File Service Based on Optimistic Concurrency Control. *Proceedings of the Tenth Symposium on Operating Systems Principles*, December 1985. s. 51-62.
305. Mullender S. J., Van Rossum G., Tanenbaum A. S., Van Renesse R., Van Staveren H.: Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*. 1990. 23. 5. s. 44-53.
306. Mukai M. W., Livny M.: Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network. *Proceedings of the Seventh IEEE International Conference on Distributed Computing Systems*, 1987.
307. Needham R. M., Walker R. D. H.: The Cambridge CAP Computer and its Protection System. *Proceedings of the Sixth Symposium on Operating Systems Principles*, November 1977. s. 1-10.
308. Nelson M., Welch B., Ousterhout J. K.: Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*. 1988. 6. 1. s. 134-154.
309. Newton G.: Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography. *Operating Systems Review*. 1979. 13. 2. s. 33-44.
310. Nichols D. A.: Using Idle Workstations in a Shared Computing Environment. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, October 1987, s. 5-12.
311. Norton P.: Inside the IBM PC. Wydanie poprawione i rozszerzone. New York, NY, Brady Books 1986.
312. Norton P., Wilton R.: *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*. Redmond, WA, Microsoft Press 1988.

313. O'Leary B. T., Kitts D. L.: Optical Device for a Mass Storage System. *Computer*, 1985, 18, 7, s. 24-35.
314. Obermarck R.: Distributed Deadlock Detection Algorithm. *ACM Transactions on Database Systems*, 1982, 7, 2, s. 187-208.
315. Oldehoeft R. R., Allan S. J.: Adaptive Exact-Fit Storage Management. *Communications of the ACM*, 1985, 28, 5, s. 506-511.
316. Olsen R. P., Kenley G.: Virtual Optical Disks Solve the On-Line Storage Crunch. *Computer Design*, 1989, 28, 1, s. 93-96.
317. Organick E. I.: *The Multics System. An Examination of its Structure*. Cambridge, MA: MIT Press 1972.
318. Open Software Foundation: *Mach Technology. A Series of Ten Lectures*. Cambridge, MA: OSF Foundation 1989.
319. Ousterhout J. K., Da Costa H., Harrison D., Kunze J. A., Kupfer M., Thompson J. G.: A Trace-Driven Analysis of the UNIX 4.2 BSD File System. *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, s. 15-24.
320. Ousterhout J. K., Cherenson A. R., Douglass F., Nelson M. N., Welch B. B.: The Sprite Network-Operating System. *IEEE Computer*, 1988, 21, 2, s. 23-36.
321. Parmelee R. P., Peterson T. I., Tillman C. C., Hatfield D.: Virtual Storage and Virtual Machine Concepts. *IBM Systems Journal*, 1972, 11, 2, s. 99-130.
322. Parnas D. L.: On a Solution to the Cigarette Smokers' Problem Without Conditional Statements. *Communications of the ACM*, 1975, 18, 3, s. 181-183.
323. Parnas D. L., Habermann A. N.: Comment on Deadlock Prevention Method. *Communications of the ACM*, 1972, 15, 9, s. 840-841.
324. Patil S.: Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes. *Technical Report*, MIT, 1971.
325. Patterson D. A., Gibson G., Katz R. H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1988.
326. Peacock J. K.: File System Multithreading in System V Release 4 MP. *Proceedings of the Summer 1992 USENIX Conference*, June 1992, s. 19-29.
327. Pease M., Shostak R., Lamport L.: Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 1980, 27, 2, s. 228-234.
328. Pechura M. A., Schoeffler J. D.: Estimating File Access Time of Floppy Disks. *Communications of the ACM*, 1983, 26, 10, s. 754-763.
329. Peterson G. L.: Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 1981, 12, 3.
330. Pfleeger C.: *Security in Computing*. Englewood Cliffs, NJ: Prentice-Hall 1989.
331. Pinkert J., Wear L.: *Operating Systems: Concepts, Policies, and Mechanisms*. Englewood Cliffs, NJ: Prentice-Hall 1989.
332. Popk G. J.: Protection Structures. *Computer*, 1974, 7, 6, s. 22-23.
333. Popk G. J., Walker B. (eds.): *The LOCUS Distributed System Architecture*. Cambridge, MA: MIT Press 1985.
334. Powell M. L., Kleiman S. R., Barton S., Shaw D., Stein D., Weeks M.: SunOS Multi-threaded Architecture. *Proceedings of the Winter 1991 USENIX Conference*, January 1991, s. 65-80.
335. Price B. G., Fabry R. S.: VMIN - An Optimal Variable Space Page-Replacement Algorithm. *Communications of the ACM*, 1976, 19, 5, s. 295-297.
336. Psaltis D., Mok F.: Holographic Memories. *Scientific American*, 1995, 273, 5, s. 70-76.
337. Purdin I. D. M., Schlichting R. D., Andrews G. R.: A File Replication Facility for Berkeley UNIX. *Software - Practice and Experience*, 1987, 17, 12, s. 923-940.

338. Quaterman J. S.: *The Matrix Computer Networks and Conferencing Systems Worldwide*. Bedford, MA, Digital Press 1990.
339. Quaterman J. S., Silberschatz A., Peterson J. L.: 4.2BSD and 4.3BSD as Examples of the UNIX Systems. *Computing Surveys*, 1985, 17, 4, s. 379-418.
340. Quaterman J. S., Hoskins H. C.: Notable Computer Networks. *Communications of the ACM*, 1986, 29, 10, s. 932-971.
341. Quinlan S.: A Cached WORM File System. *Software - Practice and Experience*, 1991, 21, 12, s. 1289-1299.
342. Rashid R. F.: From RIG to Accent to Mach: The Evolution of a Network Operating System. *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference*, 1986.
343. Rashid R., Robertson G.: Accent: A Communication Oriented Network Operating System Kernel. *Proceedings of the Eighth Symposium on Operating System Principles*, December 1981.
344. Raynal M.: *Algorithms for Mutual Exclusion*. Cambridge, MA, MIT Press 1986.
345. Raynal M.: A Simple Taxonomy for Distributed Mutual Exclusion Algorithms. *Operating Systems Review*, 1991, 25, 4, s. 47-50.
346. Redell D. D., Fabry R. S.: Selective Revocation of Capabilities. *Proceedings of the IRIA International Workshop on Protection in Operating Systems*, 1974, s. 197-210.
347. Reed D. P.: Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems*, 1983, 1, 1, s. 3-23.
348. Reed D. P., Kanodia R. K.: Synchronization with Eventcounts and Sequences. *Communications of the ACM*, 1979, 22, 2, s. 115-123.
349. Reid B.: Reflections on Some Recent Widespread Computer Break-Ins. *Communications of the ACM*, 1987, 30, 2, s. 103-105.
350. Ricart G., Agrawala A. K.: An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 1981, 24, 1, s. 9-17.
351. Richards A. E.: A File System Approach for Integrating Removable Media Devices and Jukeboxes. *Optical Information Systems*, 1990, 10, 5, s. 270-274.
352. Richter (?): *Advanced Windows*. Wyd. 3. Redmond, WA, Microsoft Press 1997.
353. Ritchie D.: The Evolution of the UNIX Time-Sharing System. *Language Design and Programming Methodology. Lecture Notes on Computer Science*, 79. Berlin, Springer-Verlag 1979.
354. Ritchie D. M., Thompson K.: The UNIX Time-Sharing System. *Communications of the ACM*, 1974, 17, 7, s. 365-375; późniejsza wersja ukazała się w *Bell System Technical Journal*, 1978, 57, 6, s. 1905-1929.
355. Rivest R. L., Shamir A., Adleman L.: On Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 1978, 21, 2, s. 120-126.
356. Rosen S.: Electronic Computers: A Historical Survey. *Computing Surveys*, 1969, 1, 1, s. 7-36.
357. Rosenblum M., Ousterhout J. K.: The Design and Implementation of a Log-Structured File System. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, s. 1-15.
358. Rosenkrantz D. J., Stearns R. E., Lewis II P. M.: System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 1978, 3, 2, s. 178-198.
359. Ruemmler C., Wilkes J.: Disk Shuffling. *Technical Report HPL-CSP-91-30*, Palo Alto, CA, Hewlett-Packard Laboratories, October 1991.
360. Ruemmler C., Wilkes J.: Unix Disk Access Patterns. *Proceedings of the Winter 1993 USENIX Conference*, January 1993, s. 403-420.

361. Ruemmler C., Wilkes J.: An Introduction to Disk Drive Modeling. *IEEE Computer*, 1994, 27, 3, s. 17-29.
362. Ruschitzka M., Fabry R. S.: A Unifying Approach to Scheduling. *Communications of the ACM*, 1977, 20, 7, s. 469-477.
363. Rushby J. M.: Design and Verification of Secure Systems. *Proceedings of the Eighth Symposium on Operating System Principles*, December 1981, s. 12-21.
364. Rushby J., Randell B.: A Distributed Secure System. *Computer*, 1983, 16, 7, s. 55-67.
365. Russell D., Gangemi G. T., Sr.: *Computer Security Basis*. Sebastopol, CA: O'Reilly and Associates, Inc. 1991.
366. Saltzer J. H., Schroeder M. D.: The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 1975, 63, 9, s. 1278-1308.
367. Samson S.: *MVS Performance Management*. New York, NY: McGraw-Hill 1990.
368. Sandberg R.: *The Sun Network File System: Design, Implementation, and Experience*. Mountain View, CA: Sun Microsystems, Inc. 1987.
369. Sandberg R., Goldberg D., Kleiman S., Walsh D., Lyon B.: Design and Implementation of the Sun Network Filesystem. *Proceedings of the 1985 USENIX Summer Conference*, June 1985, s. 119-130.
370. Sanguineti J.: Performance of a Message-Based Multiprocessor. *Computer*, 1986, 19, 9, s. 47-56.
371. Sargent M., Shoemaker R.: *The Personal Computer from the Inside Out*. Wyd. 3. Reading, MA: Addison-Wesley 1995.
372. Sarisky L.: Will Removable Hard Disks Replace the Floppy? *Byte*, March 1983, s. 110-117.
373. Satyanarayanan M.: *Multiprocessors: A Comparative Study*. Englewood Cliffs, NJ: Prentice-Hall 1980.
374. Satyanarayanan M.: Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, 1989, 7, 3, s. 247-280.
375. Satyanarayanan M.: Scalable, Secure, and Highly Available Distributed File Access. *Computer*, 1990, 23, 5, s. 9-21.
376. Sauer C. H., Chandy K. M.: *Computer Systems Performance Modeling*. Englewood Cliffs, NJ: Prentice-Hall 1981.
377. Schell R. R.: A Security Kernel for a Multiprocessor Microcomputer. *Computer*, 1983, 16, 7, s. 47-53.
378. Schlichting R. D., Schneider F. B.: Understanding and Using Asynchronous Message Passing Primitives. *Proceedings of the Symposium on Principles of Distributed Computing*, 1982, s. 141-147.
379. Schneider F. B.: Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 1982, 4, 2, s. 125-148.
380. Schrage L. E.: The Queue M/G/1 with Feedback to Lower Priority Queues. *Management Science*, 1967, 13, s. 466-474.
381. Schroeder M. D., Gifford D. K., Needham R. M.: A Caching File System for a Programmer's Workstation. *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, s. 25-32.
382. Schultz B.: VM: The Crossroads of Operating Systems. *Datamation*, 1988, 34, 14, s. 79-84.
383. Schwartz J. I., Weissman C.: The SDC Time-Sharing System Revisited. *Proceedings of the ACM National Meeting*, August 1967, s. 263-271.
384. Schwartz J. I., Coffman E. G., Weissman C.: A General Purpose Time-Sharing System. *Proceedings of the AFIPS Spring Joint Computer Conference*, April 1964, s. 397-411.
385. Seawright L. H., MacKinnon R. A.: VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 1979, 18, 1, s. 4-17.

386. Seely D.: Password Cracking: A Game of Wits. *Communications of the ACM*, 1989, **32**, 6, s. 700-704.
387. Shore J. E.: On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies. *Communications of the ACM*, 1975, **18**, 8, s. 433-440.
388. Srivastava S. K., Panzieri F.: The Design of a Reliable Remote Procedure Call Mechanism. *IEEE Transactions on Computers*, 1982, **C-31**, 7, s. 692-697.
389. Silberschatz A., Korth H. F., Sudarshan S.: *Database System Concepts*. Wyd. 3. New York, NY, McGraw-Hill 1997.
390. Silverman J. M.: Reflections on the Verification of the Security of an Operating System Kernel. *Proceedings of the Ninth Symposium on Operating Systems Principles*, October 1983, **17**, 5, s. 143-154.
391. Simmons G. J.: Symmetric and Asymmetric Encryption. *Computing Surveys*, 1979, **11**, 4, s. 304-330.
392. Sirerbox G. T. (ed.): *Selected Papers on Holographic Storage*. Bellingham, WA, SPIE Optical Engineering Press, SPIE Milestone Series, MS 95, 1994.
393. Singhal M.: Deadlock Detection in Distributed Systems. *IEEE Computer*, 1989, **22**, 11, s. 37-48.
394. Singhania R. P., Tonge F. M.: A Parametric Disk Scheduling Policy. *Proceedings of the 14th Hawaii International Conference on System Sciences*, January 1981, s. 288-297.
395. Smith A. J.: Cache Memories. *ACM Computing Surveys*, 1982, **14**, 3, s. 473-530.
396. Smith A. J.: Disk Cache-Miss Ratio Analysis and Design Consideration. *ACM Transactions on Computer Systems*, 1985, **3**, 3, s. 161-203.
397. Smith A. J.: A Survey of Process Migration Mechanisms. *Operating Systems Review*, July 1988.
398. Spafford E. H.: The Internet Worm: Crisis and Aftermath. *Communications of the ACM*, 1989, **32**, 6, s. 678-687.
399. Spector A. Z., Schwarz P. M.: Transactions: A Construct for Reliable Distributed Computing. *ACM SIGOPS Operating Systems Review*, 1983, **17**, 2, s. 18-35.
400. Stallings W.: Local Networks. *Computing Surveys*, 1984, **16**, 1, s. 1-41.
401. Stallings W.: *Operating Systems*. New York, Macmillan 1992.
402. Stankovic J. S.: Software Communication Mechanisms: Procedure Calls Versus Messages. *Computer*, 1982, **15**, 4.
403. Stankovic J. S., Ramamrithan K.: The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *Operating Systems Review*, July 1989.
404. Staunstrup J.: Message Passing Communication versus Procedure Call Communication. *Software - Practice and Experience*, 1982, **12**, 3, s. 223-234.
405. Stein D., Shaw D.: Implementing Lightweight Threads. *Proceedings of the Summer 1992 USENIX Conference*, June 1992, s. 1-9.
406. Stephenson C. J.: Fast Fits: A New Method for Dynamic Storage Allocation. *Proceedings of the Ninth Symposium on Operating Systems Principles*, December 1983, s. 30-32.
407. Stevens W. R.: *UNIX Network Programming*. Englewood Cliffs, NJ, Prentice-Hall 1990^{*}.
408. Stevens W. R.: *Advanced Programming in the UNIX Environment*. Reading, MA, Addison-Wesley 1992.
409. Strachey C.: Time Sharing in Large Fast Computers. *Proceedings of the International Conference on Information Processing*, June 1959, s. 336-341.

* Istnieje polskie tłumaczenie pt. *Programowanie zastosowań sieciowych w systemie Unix*, Warszawa, WN1 1998. – Przyp. red.

410. Sun Microsystems: *Network Programming Guide*. Mountain View, CA, Sun Microsystems, Inc. 1990, s. 168-186.
411. Svobodowa L.: *Computer Performance Measurement and Evaluation*. New York, NY, Elsevier North-Holland 1976.
412. Svobodowa L.: File Servers for Network-Based Distributed Systems. *ACM Computing Surveys*, 1984, **16**, 4, s. 353-398.
413. Tabak D.: *Multiprocessors*. Englewood Cliffs, NJ, Prentice-Hall 1990.
414. Tanenbaum A. S.: *Computer Networks*. Wyd. 2. Englewood Cliffs, NJ, Prentice-Hall 1988.
415. Tanenbaum A. S.: *Structured Computer Organization*. Wyd. 3. Englewood Cliffs, NJ, Prentice-Hall 1990.
416. Tanenbaum A. S.: *Modern Operating Systems*. Englewood Cliffs, NJ, Prentice Hall 1992.
417. Tanenbaum A. S., Van Renesse R.: Distributed Operating Systems. 1985. *ACM Computing Surveys*, **17**, 4, s. 419-470.
418. Tanenbaum A. S., Woodhull A. S.: *Operating Systems Design and Implementation*. Wyd. 2. Englewood Cliffs, NJ, Prentice Hall 1997.
419. Tanenbaum A. S., Van Renesse R., Van Staveren H., Sharp G. J., Mullender S. J., Jansen J., Van Rossum G.: Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 1990, **33**, 12, s. 46-63.
420. Tay D. H., Ananda A. L.: A Survey of Remote Procedure Calls. *Operating Systems Review*, 1990, **24**, 3, s. 68-79.
421. Teorey T. J., Pinkerton T. B.: A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM*, 1972, **15**, 3, s. 177-184.
422. Tevanian A. Jr., Rashid R. F., Golub D. B., Black D. I., Cooper E., Young M. W.: Mach Threads and the UNIX Kernel: The Battle for Control. *Proceedings of the Summer 1987 USENIX Conference*, July 1987.
423. Tevanian A. Jr., Rashid R. F., Young M. W., Golub D. B., Thompson M. R., Boosky W., Sanzi R.: A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach. *Technical Report*, Carnegie-Mellon University, Pittsburgh, PA, July 1987.
424. Tevanian A. Jr., Smith B.: Mach: The Model for Future Unix. *Byte*, November 1989.
425. The UNIX System. *The Bell System Technical Journal*, 1984, **63**, 8, Cz. 2.
426. Thompson K.: UNIX Implementation. *The Bell System Technical Journal*, 1978, **57**, 6, Cz. 2, s. 1931-1946.
427. Thurber K. J., Freeman H. A.: Updated Bibliography on Local Computer Networks. *Computer Architecture News*, 1980, 8, s. 20-28.
428. Fraiger I. L., Gray J. N., Galicieri C. A., Lindsay B. G.: Transactions and Consistency in Distributed Database Management Systems. *ACM Transactions on Database Systems*, 1982, **7**, 3, 323-342.
429. Tucker A., Gupta A.: Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.
430. UNIX Time-Sharing System. *The Bell System Technical Journal*, 1978, **57**, 6, Cz. 2.
431. USENIX Association: *Proceedings of the Mach Workshop*. Burlington, VT, October 1990.
432. USENIX Association: *Proceedings of the USENIX Mach Symposium*. Monterey, CA, November 1991.

* Znacznie poszerzoną część tej książki dotyczącą systemów rozproszonych można odnaleźć w polskim przekładzie nowszej książki tegoż autora pt. *Rozproszone systemy operacyjne* wydanej przez Wydawnictwo Naukowe PWN, Warszawa 1997. – Przyp. tłum.

433. USENIX Association: *Proceedings of the File Systems Workshop*. Ann Arbor, MI, May 1992.
434. USENIX Association: *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*. Seattle, WA, April 1992.
435. Vahalia U.: *Unix Internals: The New Frontiers*. Englewood Cliffs, NJ, Prentice Hall 1996.
436. Vuillemin A.: A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 1978, **21**, 4, s. 309-315.
437. Wah B. W.: File Placement on Distributed Computer Systems. *Computer*, 1984, **17**, 1, s. 23-32.
438. Walmer L. R., Thompson M. R.: A Programmer's Guide to the Mach System Calls. *Technical Report*, Carnegie-Mellon University, Pittsburg, PA, December 1989.
439. Weizer N.: A History of Operating Systems. *Datamation*, January 1981, s. 119-126.
440. Wood P., Kochan S.: *UNIX System Security*. Hasbrouck Heights, NJ, Hayden 1985.
441. Woodside C.: Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers. *IEEE Transactions on Software Engineering*, 1986, SE-12, 10, s. 1041-1048.
442. Wong C. K.: Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 1980, **12**, 2, s. 167-178.
443. Worthington B. L., Ganger G. R., Patt Y. N.: Scheduling Algorithms for Modern Disk Drives. *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1994, s. 241-251.
444. Worthington B. L., Ganger G. R., Patt Y. N., Wilkes J.: On-Line Extraction of SCSI Disk Drive Parameters. *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, May 1995, s. 146-156.
445. Wulf W. A.: Performance Monitors for Multiprogramming Systems. *Proceedings of the Second ACM Symposium on Operating System Principles*, October 1969, s. 175-181.
446. Wulf W. A., Levin R., Harbison S. P.: *Hydra/C mmp: An Experimental Computer System*. New York, NY, McGraw-Hill 1981.
447. Zahorjan J., McCann C.: Processor Scheduling in Shared Memory Multiprocessors. *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, May 1990.
448. Zayas E. R.: Attacking the Process Migration Bottleneck. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, October 1987, s. 13-24.
449. Zhao W. (ed.): Special Issue on Real-Time Operating Systems. *Operating Systems Review*, July 1989.
450. Zobel D.: The Deadlock Problem: A Classifying Bibliography. *Operating Systems Review*, 1983, **17**, 4, s. 6-16.

CREDITS

Fig. 3.11 from Iacobucci, *OS/2 Programmer's Guide*, ©1988, McGraw-Hill, Inc., New York, New York. Fig 1.7, p 20. Reprinted with permission of the publisher.

Fig. 5.8 from Khanna/Sebec/Zolnowsky, "Realtime Scheduling in SunOS 5.0," Proceedings of Winter USENIX, January 1992, San Francisco, California. Derived with permission of the authors.

Fig. 8.28 from 80386 Programmer's Reference Manual, Fig. 5-12, p. 5-12. Reprinted by permission of Intel Corporation. Copyright/Intel Corporation 1986.

Fig. 9.15 reprinted with permission from *IBM Systems Journal*, Vol. 10, No. 3, ©1971, International Business Machines Corporation.

Fig. 11.7 from Leffler/McKusick/Karls/Quartermann, *The Design and Implementation of the 4.3BSD UNIX Operating System*, ©1989 by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Fig. 7.6, p. 196. Reprinted with permission of the publisher.

Fig. 12.4 from *Pentium Processor User's Manual. Architecture and Programming Manual*, Volume 3, Copyright 1993. Reprinted by permission of Intel Corporation.

Fig. 18.2 from Silberschatz/Korth, *Database System Concepts, Second Edition*, ©1991, McGraw-Hill, Inc., New York, New York. Fig 15.8, p. 506. Reprinted with permission of the publisher.

Fig. 19.1 From Quartermann/Wilhelm, *UNIX, POSIX and Open Systems: The Open Standards Puzzle*, ©1993, by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Fig. 2.1, p. 31. Reprinted with permission of the publisher.

Sections of chapter 6 and 18 from Silberschatz/Korth, *Database System Concepts, Third Edition*, Copyright 1997, McGraw-Hill, Inc., New York, New York. Section 13.5 p. 451-454, 14.1.1 p. 471-472, 14.1.3. p. 476-479, 14.2, p. 482-485, 15.2.1 p. 512-513, 15.4.. p 517-518, 15.4.3. p. 523-524, 18.7. p. 613-617 18.8. p. 617-622. Reprinted with permission of the publisher.

Timeline information for the back end papers was assembled from a variety of sources which include "The History of Electronic Computing", compiled and edited by Marc Reitig, Association for Computing Machinery, Inc (ACM), New York, New York, and Sheldrake/Hutto/Fromm, Understanding Computers, ©1992, Vivid Publishing, distributed by SYBEX, San Francisco, California.

SKOROWIDZ

Abstakcyjny typ danych 709

adres fizyczny 284

- internetowy 575

- logiczny 284

- wirtualny 284

algorytm bankiera 256, 685

- C-SCAN 515

- deszyfrowania 755

- elekcji 694

- FCFS 154

- głosowania 681

- LFU 362

- LRU 356

- MFU 362

- „najpierw najkrótsze zadanie” 155-158

- piekarni 191-192

- pierścieniowy 696

- planowania priorytetowego 159, 287

- - rotacyjnego 161

- - rekonstrukcji 225

- - rotacyjny 796

- - sąsiednich stert 857-858

- SCAN 513

- SJF 155-158

- szyfrowania 755

- tyrania 695

- wykrywanie zakleszczenia 689, 691

- -, rozproszony 692

- -, secentralizowany 690

- zastępowania stron 351-363, 801

algorytm przydziału ramek 365-366

- zarządzania pamięcią 277, 329

alokator zasobów 4

anomalia Belady'ego 354

architektura von Neumanna 39

ARPANET 823-824

automata 618

Bezpieczeństwo 707, 737-764, 882-886, 922

- - poziomy 757

biblioteki przyłączane dynamicznie 281

bit ochrony 307

- - poprawności 307, 339

- - udostępnienia 383

blok indeksowy 445

bloki kontrolne procesu 793

blokowanie na użytkownik wspólny 682

- - - - - wyłączny 682

- - - - - nieskończone 201

bląd braku strony 339

boczne wejście 745

brama 577

bufor 490

- - translacji adresów stron 305

buforowanie 135, 815

C-lista 818

CSMA/CD 580

czas cyklu przetwarzania 152

- dostępu do pamięci, efektywny 307

czekanie cykliczne 244, 251
 częstotliwość braków stron 373
 czytanie z wyprzedzeniem 457

DCE 601

DCL 928

DFS 755

deskryptory bezpieczeństwa 761

– pliku 436, 781

– segmentu 382

diagram Gantta 154, 159

DMA 37-38, 478-481

DNS 574

domeny bezpieczeństwa 753

– ochrony 709

dostęp do pamięci, bezpośredni 38, 478-481

– plików zdalny 620-626

– rodzaje 422

drzewo katalogów 618, 619, 620

DSF 613

dynamiczny przydział pamięci 440

dysk, budowa 41-42

– czas szukania 510

– dostęp 510-517

– operacja zapisu 528

– opóźnienie obrótowe 511

– paskowanie 526

– przepłot 526

– struktura 509-510

– WORM 539

dyski wirtualne 92

– organizacja RAID 526

– tworzenie cienia 527

dyspozytor stron 857

dystrybucja klucza 755

dystrybutor zasobów 4

dziennik kontroli 752

– transakcji 675

Efekt Kera 538

– konwejów 155

ekspedytor 151, 287, 897

Ethernet 568

FAT 443

faza procesora 148

– wejścia-wyjścia 148

formatowanie dysku 518

– bloki uszkodzone 520-521

formatowanie fizyczne 518

– logiczne 519

fragmentacja 295, 302, 322, 329, 440, 441,

524, 800

FTP 592

funkcja jednokierunkowa 742

– systemowa 31, 55, 71-81

funkcje systemowe, komunikacja 80

– , operacje na plikach 79

– , rodzaje 73, 74

– , zarządzanie urządzeniami 79

Głodzenie 160, 166, 200-201, 206, 250, 266, 513, 671

gniazda, typy 820-821

gniazdo 820, 929

graf oczekiwania 260, 686, 687-690

– , globalny 688, 689

– , konstruowany 689

– , lokalny 687, 688, 691

– , rzeźwisty 689, 691

Hasła 740

– algorytmiczne 743

– dobrane parami 743

– jednorazowe 743

– postarzanie 741

– sposoby odgadywania 740

– szyfrowane 742

hierarchia nazewnicza 617

– urządzeń 617

Identyfikatory plików niezależnych od położenia 620

indeks pliku 407

interfejs klienta 614

– międzymasztowy 614

– użytkownika 777

– wejścia-wyjścia 481

interpretator polecen 83

IP 583

i-węzły 437, 804, 811

Jądro 5

– podsystem wejścia-wyjścia 489

– struktury danych 494

– synchronizacja 851-854

jednostka centralna 29

– składowa 615

- jednostka zarządzania pamięcią 284
 jednoznaczność nazw 618, 620
 język C 768
- Kanal wejścia-wyjścia** 501
 – bieżący 415
 – lokalny 618
 – zdalny 618
 – dowiązanie 417
- katalogi plików, tabela haszowania 453
 – efektywność 454
 – graf 417, 420
 – implementacja 452
 – lista liniowa 453
 – nazwa ścieżki 412
 – struktura 393, 408
 – ścieżka wyszukiwania 413
 – wydajność 456
- klient 614
- klucz jawny 756
 – prywatny 756
 – tajny 755
- kod ECC 518
 – korygujący 518
- kolejka komunikatów, implementacja 135
 – procesów gotowych 153, 160, 163, 287
 – modele obsługi 174
 – wejściowa 278
- kolejki wielopoziomowe 166
- komputery główne 15
 – osobiste 14
- komunikacja 69
 komunikacja międzyprocesowa 130-136, 819-826, 939
 – asynchroniczna 135
 – bezpośrednia 131
 – buforowanie 135
 – implementacja połączenia 130
 – kolejka komunikatów 135
 – komunikaty 130
 – łącze jednokierunkowe 131
 – podstawowe operacje 130
 – pośrednia 133
 – synchroniczna 136
 – zawodna 698
- komutowanie komunikatów 579
 – łączy 579
 – pakietów 579
 konsolidację 281
- konsolidatory 82
 kontekst, przełączanie 116
 koń trojaski 714
 koordynator 671, 674, 677, 680, 694
 – wykrywanie zakleszczeń 688
 kopia lokalna 630
 kopia podstawowa 630, 682
 – przeterminowana 630
- Licznik rozkazów** 108, 110, 123
- linia zgłoszenia przerwań 473
- Linux, *patrz* system Linux
- Locus, *patrz* system Locus
- lokalna pamięć podręczna 619
- LPC 909
- LRU 621, 815
- Mach.**, *patrz* system Mach
- macierz dostępów 715
 – implementacja 720
 – tabela globalna 720
- mapa bitowa 450
 – wymiany 525
- maszyna wirtualna 91-96
 – systemu DOS 914
- mechanizm zamka-klucza 722
- metoda FCFS 511
 – przekazywanie zettonu 673
 – SSTF 512
 – windy 514
- mikrojądro 86
- mobilność klienta 644
- użytkownika 636
- model bezpieczeństwa NT 759-762
- komunikacji 81
 – warstwowy ISO 582-583, 823
- moduł sterujący 37
 – wejścia-wyjścia 481
- MS-DOS, *patrz* system MS-DOS
- MULTICS, *patrz* system MULTICS
- mutant 896
- N**adprzydział pamięci 347
- nakładki 282
- namiastka procedury 281
- nazewnictwo 615-620
- nazwy strukturalne 620
- nazywanie 615
- NetBEUI 928

NetBIOS 928
 Newcastle Connection 632
 NFS 618, 623, 634
 niepodzielność 674-678
 niezależność położenia 616
 NSFNET 823

Obiekty 709, 721, 902

obiekty nazewnicze 903
 obsługa doglądana 627
 - niedoglądana 627
 - zakleszczeń 680, 681
 - zdalna 621, 625
 obszar wymiany 522-526
 ochrona 53, 70, 330, 422, 707
 - na poziomie języka programowania 729-734
 - zasadzie uprawnień 726-729
 - system Cambridge CAP 728-729
 - Hydra 726
 - wykaz dostępów 423

odbiór niejawny 600
 odliczanie czasu 604, 698, 797
 odpływanie 33
 odwołanie do systemu 31, 477, 777
 odwrócona tablica stron 312, 377
 odwzorowywanie nazw 616, 933
 ONC+ 618, 634

operacje idempotentne 628
 - konfliktowe 228
 - niepodzielne 196
 - plikowe 614
 - wejście-wyjścia 34, 69

OS/2, *patrz* system OS/2

OS/360, *patrz* system OS/360

Pamięć, algorytm upakowania 297

- wymiany 286
- dostęp bezpośredni 478-481
- efektywna szybkość przesyłania 552
- fragmentacja, *patrz* fragmentacja
- nośniki wymienne 537-541
- nazywanie plików 543
- obszary 291
- operacyjna 38, 40
- zarządzanie 63
- podrzędna 41, 492, 621
- bloków dyskowych 456
- dyskowa 622
- przepisywalna 623

pamięć podręczna, urniejscowienie 622

- zarządzanie 46
- pomocnicza 39, 66
- przydzielanie 290
- dynamiczne 294
- najgorsze dopasowanie 295
- najlepsze dopasowanie 294
- pierwsze dopasowanie 294
- segmentacja, *patrz* segmentacja
- stronicowanie, *patrz* stronicowanie
- strumieniowa, szybkość przesyłania 552
- wirtualna 14, 335-391, 859-863, 904-906
- wydajność 545
- zrzuć 76

PBC 109, 112

PIN 744

pisanie opóźniane 624
 - przy zamykaniu 624
 planista 114, 799
 - długoterminowy 114
 - krótkoterminowy 114
 - przydziału procesora 149, 798
 - średnioterminowy 115
 planowanie priorytetowe 159
 - przydziału procesora 11, 147-182, 796-798,
 898

- algorytmy 154-167
- głodzenie 160
- implementacja algorytmów 176
- ocena algorytmów 172
- symulacja 175

- rotacyjne 797

- zadań 9, 10

plik 64

- atrybuty 395
- blok indeksowy 445
- licznik otwarć 398
- metody dostępu 404-408
- ochrona 422
- przydział miejsca na dysku 438-449
- rodzaje 394
- typy 401
- wskaźnik plikowy 398

pliki, manipulowanie 82

- zarządzanie 64

pobieranie z wyprzedzeniem 57

podstawowy sieciowy system wejścia-wyjścia 928

podsystem bezpieczeństwa 917

- podsystem OS 2 917
 - POSIX 916
 - rejestracji 917
 - wejścia-wyjścia 468
 - , zadania 495-496
- port wejścia-wyjścia 40, 469, 471
- porządek całkowity 667, 670
 - częściowy 667, 668
- porządkowanie według znaczników czasu 684
 - zdarzeń 667
- postarzanie 160
- potok 791, 819
 - powłoka Bourne'a 789
 - C 789
 - Korna 789
 - poziom zaufania 758
- PPTP 928
- prawa dostępu 715
 - dostępu, cofanie 724
 - kopiowania 717
- prawo kontroli 718
 - właściciela 718
- priorytet planowania 796
- problem bizantyjskich generalów 698
 - ograniczonego buforowania 121, 183, 202, 210
 - palacy tytoniu 236
 - spójności pamięci podręcznej 621
 - śpiącego golibrody 236
 - zamknięcia 719
- procedura leniwej wymiany 338
 - obsługi przerwania 32, 474
- proces 13, 62, 107-146
 - blok kontrolny 109, 112, 793-796
 - demon 80, 493, 712
 - identyfikator 118, 793
 - init 31
 - koncepcja 107
 - konsument 184
 - kończenie 119, 137
 - , kaskadowe 119
 - lekki 123
 - macierzysty 116
 - nadzorowanie 75
 - niezależny 120
 - potomny 117
 - producent 184
 - stany 109
 - tworzenie 116
- proces Venus 643, 646-649
 - Vice 643
 - współpracujący 120
 - wykoranie 148
 - zarządzanie 62
- procesor 29
 - czolowy 19
 - Pentium, przerwania 476
 - planowanie przydziału 147-182
 - , kryteria 152
 - , wywlaściaczające 150
- procesy, drzewo 117
 - kolejka 112, 113
 - , komunikacja międzyprocesowa 129-143
 - pierwszoplanowe 162
 - planowanie 111, 854-856
 - wadliwe 699
 - współpraca 120-122
- program 62
 - diagnostyczny 75
 - komplikacja 278
 - ładowanie 280
 - rozruchowy 30, 519
 - sterujący 5
 - szceregujący 114, patrz: planista
 - uruchomieniowy 793
 - wykonywanie 280
- programowane wejście-wyjście 478
- programowanie powłokowe 792
- programy systemowe, rodzaje 82
- protokóły blokowania zasobów 679
 - sieciowe, podział na warstwy 582
- protokół 2PC 675, 676
 - AppleTalk 929
 - ARP 585
 - blokowania 229
 - bloku komunikatów serwera 927
 - dziedziczenia priorytetów 171
 - internetowy 583
 - montowania 637
 - NFS 637
 - sterowania liczbem danych 928
 - tendencjacyjny 682
 - użgadniania 472
 - Virtue 643
 - większośćowy 681, 682
 - wyzajeczenia tras 578
 - zatwierdzania 675
 - dwufazowego 675-676

- przechowywanie podręczne 621, 625
 przeciąganie sektorów 522
 przegrodki na komunikaty 581
 przekazywanie parametrów 73
 – żetonu 580
 przełączanie kontekstu 54
 przepisywalność 623
 przepustowość 17, 152
 przerwanie 31, 32, 35, 473-478, 797, 900
 – programowe 477
 –, priorytety 32
 przezroczystość położenia 616
 –, statyczna 617
 przydzielanie zasobów 69, 91
 –, graf 244-247, 255
 przywołanie 625
 PTLR 309
 pulapka 477
 punkt montażu 137
 punkty wywłaszczeń 170
- R**AID 526
- ramka pamięci 299
 region krytyczny, warunkowy 209, 210
 rejestr długości tablicy stron 309
 – przemieszczenia 285
 – transakcji 224, 225
 rejestrowanie z wyprzedzeniem 224
 rejesty procesora 110
 relacja uprzedniości zdarzeń 667, 668
 replika 629
 rezerwacja zasobów 169
 robaki 746-750
ROM 617
 rokazy sterowania pamięcią 38
 – uprzedniościowe 54
 rozliczanie 70
 rozproszone środowisko obliczeniowe 601
 rozszerzony interfejs użytkownika NetBIOS 928
RPC 136, 595, 597-600, 621, 930
- S**chemat „czekanie albo śmieci” 686
 „zranienie albo czekanie” 686
 schematy nazewnictwa 618
SCSI 469, 534
 segmentacja 315-322
 – dzielenie segmentów 321
 – fragmentacji 322
 segmentacja na żądanie 337
 –, tabela segmentów 317, 318
 – ze stronicowaniem 323
 sekcja krytyczna 186-208, 221, 235, 670
 sektory zapasowe 522
 semafor binarny 201
 – zliczający 201
 semafory 196-201, 208
 –, implementacja 197
 serwer 614
 – bezstanowy 627, 638
 sieci komunikacyjne 66
 – lokalne, rodzaje łączy 570
 – rozległe, adresowanie 573
 –, mechanizm tłumaczenia nazw 574
 –, metody wyznaczania tras 576-577
 –, rodzaje łączy 571
 –, wzorcowe modele warstwowe 825
 sieciowa pamięć wirtualna 621
 sieciowy system plików 618, 634-642
 sieć ARM 824
 skalowalność 606, 644
 skrypty powłokowe 792
 skrzynka pocztowa 134
SMB 927
 Solaris, patrz system Solaris
 spooling 8, 9, 25, 26, 57, 492
 spojność 427, 624
 sprawdzanie rzetelności 758
 – tożsamości 739
 stacja robocza 17
 stan bezpieczny 253, 257
 standard szyfrowania danych 755
STBR 319
 sterowanie współbieżnością 679
 –, algorytmy 227-233
 sterownik wejścia-wyjścia 469
 sterowniki urządzeń 30, 33
STLR 319
 stos procesu 108
 strona pamięci 299
 stronicowanie 299-314, 800-803, 861-862
 – na żądanie 337, 338, 800
 –, blokowanie stron 379
 –, brak strony 345
 –, czas obsługi braku strony 346
 –, czyste 342
 –, efektywny czas dostępu 344, 346
 –, lokalność odniesień 342

- stronicowanie na żądanie, obszar wymiany 342
 - , struktura programu 378
- odwrócona tabela stron 312
- rozmiar strony 375
- strony dzielone 314
- tabela haszowania 314
- stron 304, 312
- wstępne 374
- struktura domenowa 709
- strumieni 498
- surowe wejście-wyjście 484, 519, 817
- synchronizacja, języki wysokiego poziomu 212
 - monitor 212-219
 - problemy klasyczne 202
 - środki sprzętowe 193
- system Andrew 616, 620, 622, 624, 643-649
 - semantyka spójności 428
 - Apollo Domain 620
 - Atlas 952-954
 - CAL 723
 - CTS 957
 - czasu rzeczywistego 22, 25
 - Ibis 618, 620
 - interakcyjny 11
 - komputerowy, architektura 54-55
 - ochrona 67
 - zasoby 4, 242, 267
 - Linux 831-889
 - bezpieczeństwo 882-886
 - biblioteki systemowe 840
 - historia 831-837
 - jądro 832-834, 850
 - komunikacja międzyprocesowa 877-879
 - kontrolowanie dostępu 884
 - mechanizm rozwiązywania konfliktów 842, 845
 - moduły jądra 841
 - pakiet dystrybucyjny 832
 - planowanie 851-856
 - procesy i wątki 849-851
 - rejestracja modułów sterujących 842, 844
 - składowe systemu 839-841
 - struktura sieci 879-882
 - systemy plików 866-872
 - wejście i wyjście 872-877
 - zarządzanie modułami 842-843
 - zarządzanie pamięcią 857-866
 - procesami 846-851
 - Locus 620, 623, 629, 656-664
 - system Mach 961-962
 - komunikacja międzyprocesowa 138
 - Macintosh OS 963
 - MCP 963
 - MS-DOS, działanie 77
 - struktura 85
 - MULTICS 713, 723, 958
 - algorytmy przydzielania pamięci 324
 - NFS 623, 634
 - architektura 639
 - operacyjny 1, 3, 61, 68-104
 - implementacja, 98
 - interpretowanie poleceń 67
 - konfigurowanie 99
 - modele 92
 - ochrona 53, 70
 - projektowanie 96
 - struktura 84-91
 - tryby pracy 49-50
 - usługi 68
 - warstwy 86-90
 - OS/2, struktura systemu 90
 - zarządzanie pamięcią 326
 - OS/360 959-960
 - pamięci 44-45
 - spójność 47
 - plików, budowa 433-436
 - implementacja 433-463
 - lista wolnych obszarów 449-452
 - manipulowanie 69
 - odtwarzanie 458, 459
 - rozproszony 613-666
 - odwzorowywanie nazw 616
 - przeszroczysty 614, 615
 - wydajność 615
 - składowanie 459
 - spójność 458
 - wydajność 448
 - RC 4000 956
 - rozproszony 20, 66, 560, 613
 - powody budowy 561-563
 - RSX 963
 - secentralizowany 667
 - SCOPE 963
 - Solaris 2, synchronizacja 220
 - Solaris, wątki 127-129
 - Sprite 623, 625, 649-655
 - strumieniowego wejścia-wyjścia 768
 - Tenex 963

- system THE 955
 - struktura 88
 - TOPS-20 713
 - UNIX 630, 711, 767-830
 - 4.3 BSD, struktura 778
 - dostęp do informacji systemowych 787
 - historia 774
 - język poleceń 789
 - katalog 805
 - komunikacja międzyprocesowa 819-826
 - nadzorowanie procesów 782-784
 - ochrona katalogów 426
 - planowanie przydziału procesora 796-798
 - powłoka 767, 789
 - procedury biblioteczne 787
 - semantyka spójności 427
 - standardowe wejście-wyjście 790
 - struktura katalogów 780
 - systemu 86
 - struktury dysku 808-810
 - sygnały 784-786
 - system plików 778, 803-813
 - wejścia-wyjścia 495, 814-819
 - United 630, 631
 - zarządzanie pamięcią 798-803
 - procesami 792-798
 - Venus, struktura 89
 - VMS 963
 - wieloprocesorowy 17
 - Windows 963
 - Windows NT 759, 891-944
 - bezpieczeństwo 922
 - domeny 932
 - egzekutor 901-913
 - gniazda 929
 - historia 892
 - jądro 896-901
 - komunikacja międzyprocesowa 141, 939
 - obiekty jądra 934
 - podsystemy środowiskowe 913-917
 - praca w sieci 926-934
 - protokły 927
 - przerwania 900
 - przetwarzanie rozproszone 929
 - schemat blokowy 895
 - system plików 918-926
 - warstwa abstrakcji sprzętu 895
 - zarządcę wejścia-wyjścia 910-913
 - zarządzanie pamięcią 940-942

- system Windows NT, zarządzanie procesami 937-939
 - - - zwołnictwa i serwery 931
 - XDS-940 954
 - systemy czasu rzeczywistego, lagodne 169
 - - - rygorystyczne 169
 - heterogeniczne 168
 - homogeniczne 168
 - operacyjne, wczesne 945-952
 - tolerując awarie 18
 - sytuacje wyjątkowe 476, 797
 - szamotanie, model strefowy 369
 - - - zbioru roboczego 371
 - szeregowalność 227, 229
 - szifrowanie 754-756
 - z kluczem jawnym 756
 - szyna 469, 470

Ścieżka 618

- środowisko 16-bitowego systemu Windows 915
– systemu MS-DOS 914-915
– Win32 916

Tabelle FAT 443-918

- montaż 497, 809
 - otwieranych plików 436
 - punktów montażu 618
 - stanów urządzeń 35, 36
 - stron 304, 312, 794
 - tras 576
 - tekst czysty 755**
 - zaszyfrowany 755
 - telnet 591**
 - TLB 305**
 - tłumaczenie nazwy ścieżki 641**
 - tolerowanie uszkodzeń 606**
 - topologia sieci 563**
 - tożsamość procesu 847**
 - transakcja 222**
 - niepodzielna 221, 674
 - wycofana 223
 - zatwierdzanie 675
 - zatwierdzona 223
 - translatory 82**
 - tunelowy protokół od punktu do punktu 928**

Uaktualnianie 623, 629

- uchwyt plikowy 436
uniwersalne zamknięcie, algorytm 256

- UNIX, *patrz* system UNIX
 upakowywanie 926
 uporządkowanie według znaczników czasu 231, 686
 uprawnienie 721
 uprzedniość zdarzeń 667, 668
 urządzenia wejścia-wyjścia, blokowe 484, 814, 873, 874-876
 — sieciowe 485, 873
 — właściwości 487
 — znakowe 815, 873, 876-877
 usługa 614
 UI/CP 573
 uwierzytelnianie 739-741, 883
 uzgadnianie 603
- V-węzeł** 639, 641
- Warunek uporządkowania całkowitego** 669
wątek 122-129
 — wymagane zasoby 129
wątki, biblioteka 127
wczesne zwalnianie 457
wektor przerwań 32
wędrowka plików 616
wieloprogramowanie 9, 10, 147
wieloprogramowość 114
wieloprzetwarzanie 18, 19, 856
wielozadaniowość 11
Windows NT, *patrz* system Windows NT
wirtualna przestrzeń adresowa 793
wirtualny system plików 639
wirująca blokada 197, 856
wirusy 16, 750-751
wsad 5
współbieżność 668
współczynnik trafień 306
wydajność wejścia-wyjścia, zasady 502
wykaz dostępów 423, 721, 722
 — uprawnien 721, 722
wykorzystanie procesora 152
wykrywanie błędów 69
wykrywanie i eliminacja sierot 628
wymiana 115, 330, 799, 861
wywłaszczenie 150, 151, 240, 244, 250, 265, 266, 686, 898
wywołanie procedur zdalnych, *patrz* RPC
 — systemowe 31, 55, 71-81, 777
wzajemne wykluczanie 186, 243, 249, 670-674
wzór Little'a 174
- Zagrożenia, nadzorowanie** 751-754
 — programowe 744
 — systemowe 746
zakleszczenie 200, 206, 241-273, 685-694, 896
 — charakterystyka 243-247
 — likwidowanie 264-266
 — unikanie 248, 252-260
 — wykrywanie 260-264, 687-694
 — zapobieganie 248, 249-252, 685-687
zamawianie zasobów, algorytm 258
zamki 722, 896
zamówienia wejścia-wyjścia 496, 499
zapewnianie niepodzielności 223-233
zapora ogniowa 753
zarządcą obiektów 902
 — pamięci wirtualnej 904
 — procesów 908
zarządzanie pamięcią 798-803, 940-942
zasada wiedzy koniecznej 709
zasoby lokalne 614
 — zdalne 559, 614
zastępowanie stron 349
zastępowanie stron, algorytmy 351-363, 801
 — bit modyfikacji 349
 — — zabrudzenia 349
zdarzenia współbieżne 669
zmiastość danych 622
znacznik czasu 669, 683, 686
 — globalnie jednoznaczny 683
 — lokalny 683
 — niepowtarzalny 683
zwielokrotnianie 619
 — na zadanie 630
 — plików 616, 629
 — schemat 630
 — sterowanie 629
zwrótka 931
- Żelion** 673

Ewolucja systemów komputerowych

- 1834 Babbage, Lovelace - projekt urządzenia pn. „Analytical Engine”**
- 1854 Boole opublikował *Laws of Thought* (algebra booleana i logika)**
- 1928 Maszyna szyfrująca ENIGMA, Niemcy**
- 1930 Model 1, komputer elektromechaniczny, Bell Labs**
- 1938 COLOSSUS - Alan Turing, Flowers i Newman - pierwsze, w pełni elektroniczne urządzenie liczące**
- 1941 Kalkulator elektromechaniczny Konrada Zuse, Austria - 64 słowa pamięci, mnożenie w czasie 3 s**
- 1944 MARK I, komputer elektromechaniczny, Howard Aiken, Harvard University**
- 1945 ENIAC, Electrical Numerical Integrator and Computer, J. W. Mauchly i P. J. Eckert = Pierwsza „pluskwa” (cma) znaleziona przez Grace Hopper**
- 1948 Pierwszy tranzystor, Bell Labs: Bardeen, Brattain i Shockley**
- 1949 EDSAC, maszyna zbudowana przez Alana Turinga, akustyczne lampy pamięciowe, wyświetlacz oscyloskopowy, pierwsza biblioteka podprogramów = Pierwszy język asemblera dla komputera UNIVAC I**
- 1951 EDVAC pod klucz, pierwszy program zapamiętyany w komputerze - maszyna opracowana przez Johna von Neumanna i jego zespół**
- 1952 Pierwszy komercyjny kompilator = Maurice Wilkes oznajmia o możliwości mikroprogramowania**
- 1954 UNIVAC I, pierwszy komputer sprzedany Ministerstwu Obrony USA (zbudowany w Harvardzie) = MATH-MATIC - pierwszy kompilowalny język (komputer UNIVAC I) = Język FORTRAN opracowany w IBM = Pierwszy asembler rodem z IBM = Komputer IBM 650 - pierwszy komputer produkowany masowo**
- 1955 TRIDAC - pierwszy komputer na tranzystorach**
- 1957 Powstanie firmy DEC = IPL - Information Processing Language**
- 1958 ALGOL58 - język algorytmiczny = Komputer Atlas z pamięcią wirtualną - University of Manchester, Anglia**
- 1959 Układ scalony, opracowany przez Noyce'a i Moore'a = Komputer PDP-1 firmy DEC, z monitorem i klawiaturą**
- 1962 CTSS - Compatible Time-Sharing System, opracowany przez Corbato w MIT**
- 1964 Języki PL/I i APL = Komputer DEC PDP-8 - pierwszy masowo produkowany minikomputer = System komputerowy IBM/360**
- 1965 Control Data 6600 - pierwszy superkomputer, który odniósł sukces rynkowy = Język programowania BASIC = XDS-940 - system z podziałem czasu, opracowany w University of California w Berkeley = Język Simula = Sieć ARPAnet = Pamięć podręczna opracowana przez Wilkesa**

- 1966 System OS/360 = MULTICS** - system operacyjny z podziałem czasu, zaprojektowany w MIT
- 1968 System operacyjny THE, Holandia; struktura warstwowa i przetwarzanie współbieżne**
- 1969 Drukarka laserowa = UNIX** - Thompson i Ritchie, AT&T
- 1970 Język Pascal = komputer RC 4000 skonstruowany przez firmę Regenentalen, rdzeń - czyli jądro systemu operacyjnego**
- 1971 Intel - pierwszy komercyjny mikroprocesor (4004): 4-bitowy, 0,06 MIPS, 300 \$**
- 1972 Język C = Język Smalltalk = Minikomputer DEC PDP-11 = 8-bitowy mikroprocesor Intel 8008**
- 1973 Sieć Ethernet firmy Xerox PARC = Dysk twardy Winchester**
- 1975 Altair 8800 - komputer dla hobbystów: Intel 8080, 1 KB RAM, 375**
- 1976 MCP - wieloprocesorowy system operacyjny = SCOPE - system wieloprocesorowy**
- 1977 Komputery osobiste: Apple II, Radio Shack TRS80, Commodore PET = System operacyjny CPM**
- 1978 Komputer DEC VAX z systemem operacyjnym VMS**
- 1979 UNIX 3BSD = Język Ada = Pakiet Visicalc**
- 1981 IBM PC, 16 KB RAM = Xerox Star - pierwsza stacja robocza, graficzny interfejs użytkownika, Ethernet, myszka, Smalltalk**
- 1982 Compaq - pierwszy komputer przenośny = TurboPascal = Modula 2**
- 1983 Globalna sieć Internet**
- 1984 Apple Macintosh - pierwszy komputer osobisty z graficznym interfejsem użytkownika = System TrueBASIC = System operacyjny SunOS = Standard Postscript**
- 1985 Język C++ = Microsoft Windows = Superkomputer Cray-2 = Maszyna**
- 1987 System operacyjny OS/2 = Układ scalony zawierający 4 Mbitów pamięci DRAM = Początek sieci lokalnych (LAN - Local Area Networks) w wielkich instytucjach**
- 1988 Komputer NeXT - uniksowa stacja robocza, system obiektowy, graficzny interfejs użytkownika = Robak internetowy**
- 1989 Motif - standard graficznego interfejsu użytkownika dla uniksowych stacji roboczych = Układ Intel 80486**
- 1990 Windows 3.0 = Język Modula 3 = Prototyp usługi WWW autorstwa Bernersa i Lee**
- 1992 Wielowątkowy, wielozadaniowy, uniksowy system operacyjny Solaris firmy Sun = DEC Alpha - pierwszy 64-bitowy układ RISC**
- 1993 Windows NT = Procesor PowerPC firm IBM, Apple i Motorola = Procesor Intel Pentium**
- 1995 Język Java = System Microsoft Windows 95**
- 1996 Procesor Intel Pentium Pro**